# Image Processing Based Control of Mobile Robotics

by

Jesus Aldaco Lopez

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved August 2016 by the
Graduate Supervisory Committee:

Armando A. Rodriguez, Chair
Panagiotis K. Artemiadis
Spring M. Berman

ARIZONA STATE UNIVERSITY

December 2016

# ABSTRACT

Toward the ambitious long-term goal of a fleet of cooperating Flexible Autonomous Machines operating in an uncertain Environment (FAME), this thesis addresses various control objectives for ground vehicles. There are two main objectives within this thesis, first is the use of visual information to control a Differential-Drive Thunder Tumbler (DDTT) mobile robot and second is the solution to a minimum time optimal control problem for the robot around a racetrack.

One method to do the first objective is by using the Position Based Visual Servoing (PBVS) approach in which a camera looks at a target and the position of the target with respect to the camera is estimated; once this is done the robot can drive towards a desired position $(x_{ref}, z_{ref})$. Another method is called Image Based Visual Servoing (IBVS), in which the pixel coordinates $(u, v)$ of markers/dots placed on an object are driven towards the desired pixel coordinates $(u_{ref}, v_{ref})$ of the corresponding markers. By doing this, the mobile robot gets closer to a desired pose $(x_{ref}, z_{ref}, \theta_{ref})$.

For the second objective, a camera-based and noncamera-based $(v, \theta)$ cruise-control systems are used for the solution of the minimum time problem. To set up the minimum time problem, optimal control theory is used. Then a direct method is implemented by discretizing states and controls of the system. Finally, the solution is obtained by modeling the problem in AMPL and submitting to the nonlinear optimization solver KNITRO. Simulation and experimental results are presented.

The DDTT-vehicle used within this thesis has different components as summarized below: (1) magnetic wheel-encoders/IMU for inner-loop speed-control and outer-loop directional control, (2) Arduino Uno microcontroller-board for encoder-based inner-loop speed-control and encoder-IMU-based outer-loop cruise-directional-control, (3)

Arduino motor-shield for inner-loop speed-control, (4) Raspberry Pi II computer-board for outer-loop vision-based cruise-position-directional-control, (5) Raspberry Pi 5MP camera for outer-loop cruise-position-directional control.

Hardware demonstrations shown in this thesis are summarized: (1) PBVS without pan camera, (2) PBVS with pan camera, (3) IBVS with 1 marker/dot, (4) IBVS with 2 markers, (5) IBVS with 3 markers, (6) camera and (7) noncamera-based $(v, \theta)$ cruise control system for the minimum time problem.

*To God*

*To my parents, Vicente and Rosa Maria and my brothers, Vicente, Roberto and Manuel.*

# ACKNOWLEDGMENTS

To God because without Him I am nothing.

I would like to express my gratitude to my advisor Professor Armando A. Rodriguez for his knowledge, enthusiasm, motivation and continuous support during the MS program. To Dr. Artemiadis and Dr. Berman for being part of my defense committee.

To my extended family living in Phoenix,AZ for making me feel comfortable and supporting me throughout my studies.

To my classmates who helped me in every possible way during my research, Venkatraman, Zhichao, Xianlong, Nikilesh, Zhenyu, Michael and Victoria.

I would like to thank the Consejo Nacional de Ciencia y Tecnologia (CONACYT) for supporting me during the second year of my studies.

Last but not least, I want to thank my parents for their love and the opportunity they gave me to come here to continue my studies and my brothers for their support and for being an example to me.

I could not have done it without you.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

# Chapter 1

# INTRODUCTION AND OVERVIEW OF WORK

## 1.1 Introduction and Motivation

Robotic systems need the ability to understand their workspace to behave autonomously. Vision is a useful robotic characteristic since it mimics the human sense of vision and allows for noncontact measurement of the environment [9], [11]. New technologies (e.g. Arduino, Raspberry Pi with compatible interfaces, software and actuators/sensors) now permit people to perform very complicated tasks. Within this thesis low-cost ground vehicles are used for robotics research.

Two central objectives of the thesis were how to use visual information to develop an outer loop position controller for the Differential-Drive Thunder Tumbler vehicle (used in [58]) and how to make the vehicle go around a racetrack in minimum time with the help of a cruise control system.

The work presented here is a step toward the longer-term goal of achieving a fleet of *Flexible Autonomous Machines operating in an uncertain Environment (FAME).* Such a fleet can involve multiple ground and air vehicles that work collaboratively to accomplish coordinated tasks. Such a fleet may be called a swarm [60]. Potential applications can include: remote sensing, mapping, intelligence gathering, intelligence-surveillance-reconnaissance (ISR), search and rescue, manufacturing, teleoperation and much more. It is this vast application arena as well as the ongoing technological revolution that continues to fuel robotic vehicle research.

For the vehicle used within this thesis both kinematic and dynamical models are examined. Here, differential-drive means that the speed of each of the rear wheels

are controlled independently by separate dc motors. This vehicle is non-holonomic; i.e. the two (2) $(x, z)$ or $(v, \theta)$ controllable degrees of freedom are less than the three (3) total $(x, z, \theta)$ degrees of freedom. This fundamentally limits the ability of a single continuous (non-switching) control law to "precisely park the vehicle" (see discussions below based on work of [1], [2], [4]).

This chapter attempts to provide a fairly comprehensive literature survey - one that summarizes relevant literature and how it has been used. This is then used as the basis for outlining the central contributions of the thesis.

## 1.2 Literature Survey: Robotics - State of the Field

In an effort to shed light on the state of ground robotic vehicle image based control design, minimum time optimal control problem, modeling and hardware, the following literature survey is offered.

- **Image Based Control of Mobile Ground Vehicles.**

  Hutchinson et.al. in [11] provide an introduction to vision-based control (visual servo control) of robotic manipulators. It is also presented basic coordinate transformations, velocity representations as well as the geometric aspects of the image formation process. More recently in[6] it is presented the basic concepts for the development of both image and position-based robot control that are explored within this thesis, i.e. not only for robotic manipulators but for mobile ground robots as well. This work provides a foundation for research within robotic systems and the use of visual information to control them.

  Again in [6] the author describes the Position Based Visual Servoing. The pose of the target with respect to the camera is estimated. The geometry of the target is known, i.e. the position of a number of points $(X_i, Y_i, Z_i)$, $i \in [1, N]$

on the target with respect to the target's coordinate frame $\{T\}$ (in this thesis a chessboard is used as the target). The camera intrinsic parameters must be known. An image is captured and the corresponding image plane coordinates $(u_i, v_i)$ are determined by using image processing techniques. Estimating the pose using $(u_i, v_i)$, $(X_i, Y_i, Z_i)$ and camera intrinsic parameters is known as the Perspective-n-Point problem (PnP). Gao and Zhang in [7] provide a formal definition of the PnP problem. Given a set of non-collinear 3D coordinates of reference points $\mathbf{p}_i = (X_i, Y_i, Z_i)^T \, i = 1, ..., n \, n \geq 3$ expressed in an object-space coordinates and the corresponding pixel coordinates $\mathbf{u}_i = (u_i, v_i, 1)^T$ the following relationship is obtained: $w_i \, \mathbf{u}_i = C(R\mathbf{p}_i + \mathbf{t})$ (where $C$ is the camera intrinsic parameter matrix, $w_i$ is a scalar projective parameter denoting the depth of a feature point in the camera coordinate system, $R$ is a rotation matrix from object to camera coordinate frame and $\mathbf{t}$ is the translation vector from the camera to the object coordinate frame). Thus in the PnP problem $R$ and $t$ must be found. In this thesis the OpenCV function *solvePnp* is used which implements an iterative procedure based on the Levenberg-Marquardt algorithm to find $R$ and $t$. Once this estimation is performed, a mobile ground robot can be driven towards a goal position $(x_{ref}, z_{ref})$.

Ebata, Ito and Shibata presented in [8] and [9] a relationship between camera velocity $v_{cam} = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$ and a mobile ground robot velocity $(v, \omega)$ is given. This relationship is used within this thesis to obtain the control law for the Image Based Visual Servoing method. Also in these two works a switching control strategy was used. First the desired linear velocity of the mobile robot and the desired pan angle of the camera are computed to regulate pixel coordinates of feature points in the image plane to the desired pixel coordinates. Then the orientation of the mobile robot platform is driven towards that of the

camera. In [15] the authors discuss different ways to approximate the interaction or jacobian matrix $J$ that relates the variation of pixel coordinates in the image plane with the camera linear and angular velocities $(v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$. One way to approximate the jacobian matrix, which was used here, is to just compute it with the desired pixel coordinates of feature points which results in a constant matrix used throughout the control task. The local minima problem, among others, is addressed within [16]. The authors describe that local minima is defined such that the commanded velocity to a robotic system is zero $\dot{r} = 0$ and the robotic system is not at the desired position $q \neq q_{ref}$.

- **Minimum Time Optimal Control Problem.**

  A study on the different applications of optimal control such as the energy-optimal trajectories for unmanned aerial vehicles (UAVs) equipped with solar cells, the minimal fuel thrust for the terminal phase of a lunar soft-landing mission, and optimization of the racing line for a hybrid vehicle around a close race track is given within [35].

  Casanova's PhD thesis [23] adresses the minimum lap time maneuvring with the use of a direct method to solve the optimal control problem for a Formula One car. In [24] it is showed how to use a scaling factor $\alpha$ to transform the time dependent system $\dot{x} = \frac{dx}{dt} = f(x(t), u(t))$ into a distance dependent system $\frac{dx}{ds} = \alpha\, f(x(t), u(t)) = \bar{f}(x(s), u(s))$. This is helpful since $s_f$ is known, i.e. the distance of the racetrack instead of $t_f$. In [23] and [25] a discretization of the control signal is performed to convert the optimal control problem into a Nonlinear Programming Program (NLP).

  In [26] a real-time control of autonomous vehicles under minimum travelling time objective is studied. The control inputs for the vehicle are computed from

a nonlinear model predictive control (MPC) scheme. A system transformation from time-dependent to spatial coordinates-dependent is made to make time an optimization variable. Simulation and experimental results (using miniature race cars) are presented. Implementation of this method uses a camera placed on top of the racetrack, i.e. the image processing is not performed on-board.

Limebeer and Perantoni in [27] studied the optimal control of a Formula One car on a three-dimensional track and presented the solution based on a direct numerical method. Velenis et.al. in [28] solved a minimum-time cornering problem along a 90 deg corner for a rear-wheel drive vehicle using two of the most common rally racing maneuvers, the Trail-Braking and Pendulum-Turn. They obtained the solution numerically by employing a tool called EZOPT, which is a direct optimization software. It uses collocation to transcribe an optimal control problem to a nonlinear programming problem; this in turn provides an interface to NPSOL, a nonlinear optimization program.

Zhang in [36] presents a tutorial for solving optimal control problems with a proposed DMOC (Discrete Mechanics and Optimal Control) methodology to solve optimal control problems.

Within [41] and [42] it is described the AMPL programming language which is used within this thesis. A description of the NEOS server, which is a free internet-based service for solving numerical optimization problems is described in [37], [38], [39] and [40].

Desineni in [31] writes about different optimal control problems, their formulation, classification and solution.

Jorge Nocedal in [33] provides an extensive analysis of numerical optimization among other about constrained optimzation (KKT conditions), unconstrained optimization, line search methods.

In [34] it is described the Nonlinear Programming solver that was used within this thesis, namely KNITRO.

- **Robot Modeling.** Siciliano's book [61] addresses modeling for both robotic manipulators and mobile robots. Within this thesis, the focus is on differential-drive.

- **Differential-Drive Robot Modeling.** Within this thesis, differential-drive (Thunder Tumbler) ground vehicles are used. Here, differential-drive means that there are two rear wheels - each with an independent torque generating armature controlled dc motor on it [55]. As such, these dc motors can be used to independently control the speed of the rear wheels. Nominally, the assumption that the motors are identical is made. The motor inputs (vehicle controls) are voltages. The sum of these voltages is used to control the vehicle's speed $v$. The difference is used to control the direction $\theta$ of the vehicle.

  - *Kinematic Model.* A kinematic model for differential-drive robot (ignoring dynamic mass-inertia effects) is presented within [46], [45]. Within this kinematic model, it is assumed that the translational and angular velocities $(v, \omega)$ of the robot are realized instantaneously.

    While the kinematic model is controllable from a nonlinear geometric (Lie bracket) point of view [3]; i.e. the vehicle can be "parked;" locally, it can lose linear controllability.(See Section 4.1 on page 61 for more complete nonlinear and linear controllability argument details).

It must be noted that this (linear/local) loss of controllability is a direct consequence of the fact that the vehicle cannot move sideways to park.

– *Dynamical Model.* A dynamical model can take the torques applied to the robot wheels as inputs (controls) to the system. This is done within [47], [50]. The model presented within these works incorporates dynamic (acceleration constraining) mass-inertia effects as well as friction, wheel slippage etc. Within [5], a two-input two-output (TITO) linear time invariant (LTI) model - including dc motor dynamics as well as vehicle mass-inertia effects - is presented for a differential-drive ground vehicle. This model was exploited within [57] for control design. It is very important to note that the vehicle model becomes nonlinear when one considers the planar $(x, z)$ coordinates of the vehicle.

– *Non-Holonomic Differential Drive.* Non-holonomic differential-drive vehicle modeling and control is addressed within [2]. The paper relies on the fundamental nonlinear controls work within [1] to address non-smooth stabilization for differential-drive vehicles. Here, non-holonomic implies that the two controllable degrees of freedom $(x, z)$ or $(v, \theta)$ is less than the three total degrees of freedom $(x, z, \theta)$. Astolfi (1994) exploits the work of Brockett (1983) to show that the classic parking stabilization objective $(x_{ref}, z_{ref}, \theta_{ref})$ cannot be achieved with a continuous control law; i.e. to park the vehicle, one must switch between continuous control laws.

An underlying consequence of the above is that the linearized vehicle position model for the differential-drive and vehicle is uncontrollable [48] - an obvious fact since the vehicle (differential-drive) cannot move sideways to park the vehicle. Despite this, it is well known that this vehicle is control-

lable from a nonlinear geometric (Lie bracket) point of view [3]; i.e. the vehicles can be "parked."

  - *Nonlinear Controllability.* Nonlinear (Lie bracket-based) controllability for differential-drive vehicles is addressed within the text [3]. Within this text, it is shown that while the differential-drive vehicle is locally (linearly) uncontrollable (discussed above and in [48]) the vehicle is actually globally (nonlinearly) controllable. Lie brackets are used to prove the latter.

- **Classical Controls.** Classical control design fundamentals are addressed within the text [55]. Internal model principle ideas - critical for command following and disturbance attenuation - are presented within [51], [55]. General PID (proportional plus integral plus derivative) control theory, design and tuning are addressed within the text [53]. Fundamental performance limitations are discussed with [52],[55].

- **Nonlinear Control.** Fundamental theory addressing the existence of a continuous stabilizing control laws for nonlinear systems was introduced within the ground breaking work [1]. This work was used within [2] and [4] to address the classic parking problem for differential-drive vehicles (see discussion above). A nonlinear control law for the parking problem is also presented within [46] - the stability of the control law based upon Lyapunov ideas.

- **Robot Inner-Loop Control.** A proportional-plus-integral-plus-derivative (PID) inner-loop control design is addressed within [66], [54]. A PI controller is used for inner-loop control within [67], [57]. In Chapter 3, PI inner-loop speed $(\omega_r, \omega_l)$ control law is examined for the differential-drive vehicle.

- **Robot Outer-Loop Control.** Within this thesis, various outer-loop control laws are examined. When relevant, existing work in the literature was exploited.

  – *Cartesian Stabilization and Parking Problem.* Viera et. al. in [4] show how to use linear controllers to address the classic posture and Cartesian stabilization problems. The posture (or parking) problem addresses arriving at a desired point $(x_{ref}, z_{ref})$ with a specified posture angle $\theta_{ref}$. The Cartesian stabilization problem addresses moving a vehicle from one planar $(x, z)$ coordinate to another coordinate $(x_{ref}, z_{ref})$. Within [4], the authors show that a (smooth) linear control law (involving longitudinal distance to the target $(x_{ref}, z_{ref})$ and the angle between the vehicle and target) can be used to get arbitrarily $\epsilon$-close to a desired planar $(x_{ref}, z_{ref})$ and to a desired parking target (posture) $(x_{ref}, z_{ref}, \theta_{ref})$. Again, based on the work of [1], one must switch control laws in order to reach the desired $(x_{ref}, z_{ref}, \theta_{ref})$ parking target.

    The parking problem is related to the so-called *Cartesian stabilization problem* which addresses achieving a desired $(x, z)$ point.

  – *Cruise Control.* Cruise control is a fundamentally important feature for a ground robotic system. Within this thesis, an encoder-camera based (PD with roll off) outer-loop $(v, \theta)$ control law is developed that permits cruise control along a camera visible line/path.

    The map from the reference commands ($v_{ref}$, $\omega_{ref}$) to the actual velocities ($v$, $\omega$)looks like a simple diagonal system (e.g. $diag(\frac{a}{s+a}, \frac{b}{s+b})$) at low frequencies - a consequence of a well-designed inner-loop control system. (See inner-loop work within Chapter 3; outer-loop work in Chapter 5). The outer-loop $\theta$ controller therefore "sees" $\frac{b}{s(s+b)}$. From classical root lo-

cus ideas [55], a proportional controller is therefore justified - provided that the gain is not too large. If the gain is too large, oscillations (or even limit cycle behavior) are expected in $\theta$. A PD controller with roll off would help with this issue. (See work within Chapter 5).

- **Vision Algorithms.** In this thesis different image processing ideas were used. As mentioned above, the PnP problem deals with finding the pose of a target with respect to the camera by using a set of 3D coordinates of reference points $(X, Y, Z)$ and their corresponding pixel coordinates $(u, v)$ in the image plane. Relevant theory is presented within [6], [7] and [22].

  Threshold of a gray image is implemented to detect black tape on the ground for the line tracking behavior; also morphology operations such as dilation and erosion to remove noise. Contours are then used to compute the pixel coordinate of the center of the black tape in the image plane. This information is within [17]. Also the thresholding of a color image (hsv image) is exploited from [20] for the filtering of markers within the Image Based Visual Servoing outer loop control. The open source computer vision library OpenCV is greatly used in this thesis. Useful information for using this library is presented within [17].

- **Cameras.** Within this research, a Raspberry Pi camera (2592 $\times$ 1944 pixel or 5 MP static images; 1080p30 (30 fps), 720p60 and 640$\times$480p60/90 MPEG-4 video) is used. It connects directly to the Raspberry Pi II's GPU (graphical processing unit). It is capable of 1080p full HD video. Because the camera is directly connected to the GPU, there is very little impact on the CPU (central processing unit). This makes the CPU available for other processing tasks [71]. Within this thesis, cameras are used for outer-loop control law implementation (e.g. $(v, \theta)$, $(x, z)$).

- **Arduino.** Within this thesis, a great use of the Arduino Uno microcontroller board (16MHZ ATmega328 processor, 32KB Flash Memory, 14 digital I/O pins, 6 analog inputs, $25) is done. More detailed specifications for the Arduino Uno board are presented within [68]. It is used to implement inner- and outer-loop control laws for the differential-drive Thunder Tumbler vehicle.

- **Raspberry Pi II.** Within this thesis, a great use of the Raspberry Pi II computer board (900 MHz quad-core ARM Cortex-A7 CPU, 1GB SDRAM, 40 GPIO pins, camera interface, $35) is done. Introductory and technical details for the Raspberry Pi II are discussed within [69]. The Raspberry PI II us used to implement outer-loop $(x, z)$, $(v, \theta)$ control laws within this thesis.

- **Actuators and Sensors.** Actuators and sensors are addressed within the text [62].

- **DC Motors.** Simple armature controlled dc motor modeling concepts are addressed from a controls perspective within [55]. DC motor modeling for wheeled robot applications is addressed within [63]. In this paper, nonlinear effects are neglected. Nonlinear modeling and identification for dc motors is addressed within [64], [65]. Also, see detailed discussion presented above on the TITO LTI vehicle-motor model presented within[5].

- **Encoders.** Magnetic encoders consist of magnets and a hall effect sensor. They are inherently rugged and operate reliably under shock, vibration and high temperature [70]. Rotary optical encoders are the most widely used encoder design. They consist of an LED light source, light detector, code disc, and signal processor [70]. Within this thesis, magnetic encoders are used on the wheels of the differential-drive Thunder Tumbler ground vehicles.

## 1.3 Contributions of Work: Questions to be Addressed

Within this thesis, the following fundamental questions are addressed. The answers to these questions are important to move toward the longer-term *FAME* goal.

1. **How can a differential-drive mobile robot be controlled using visual information?**

   Position Based Visual Servoing (PBVS) and Image Based Visual Servoing (IBVS) are two basic methods to control a robotic system using visual information. In this thesis both methodologies are explored for the Differential-Drive Thunder Tumbler mobile robot. In PBVS the position of the vehicle with respect to a chessboard is estimated (using a known geometric model of a cheessboard-/target and its visual features). This information is later used to drive the robot to a desired position $(x_{ref}, z_{ref})$. In IBVS the control task is defined in the image plane, i.e. the pixel coordinates $(u, v)$ of markers/dots placed on an object are driven towards the desired pixel coordinates $(u_{ref}, v_{ref})$ of the corresponding markers. By doing this, the mobile robot gets closer to a desired pose $(x_{ref}, z_{ref}, \theta_{ref})$.

2. **How can a differential-drive mobile robot go around a racetrack in minimum time?**

   To make a differential-drive mobile ground robot go around a racetrack in minimum time, optimal control theory is used in this thesis.

   The basic control system used to accomplish this task is a speed-directional $(v, \theta)$ outer loop control.

   A camera-based and a noncamera-based implementation of the outer loop are presented (both with simulation and experimental results). The camera-based

method uses an encoder-camera based $(v, \theta)$ outer loop control; the noncamera-based method uses an encoder-IMU based $(v, \theta)$ outer loop control.

To set up the minimum time problem, optimal control theory is used. Then a direct solution method is implemented by discretizing states and controls of the system. This results in a Nonlinear Programming (NLP) problem. This problem is written in AMPL modeling language which then is interfaced with the nonlinear optimization solver KNITRO. Finally a solution to the NLP problem is obtained.

3. **What are typical outer-loop objectives?** For the vehicle applications considered within this thesis, three (3) outer-loop objectives are examined:
   (1) Position Based Visual Servoing by estimating the position of the robot using a front-facing camera with respect to a chessboard, thus driving the vehicle from an initial position $(x_0, z_0)$ to a desired position $(x_{ref}, z_{ref})$,
   (2) Image Based Visual Servoing by controlling the position (in the image plane) of the pixel coordinates $(u, v)$ of several markers/dots placed on an object which the robot sees with the front-facing camera and by doing this driving the robot closer to a desired pose $(x_{ref}, z_{ref}, \theta_{ref})$
   (3) speed-direction $(v, \theta)$ cruise control by using encoders to measure speed information and both a camera and an IMU for measuring directional information for the solution of the minimum time optimal control problem.

4. **What is a suitable outer-loop model?** If the inner-loop is designed well, after it is closed it can yield a system (seen by the outer-loop controller) that is very simple looping (e.g. $diag(\frac{a}{s+a}, \frac{a}{s+a})$, looks like identity at low frequencies).

5. **What is a suitable outer-loop control structure?** Suppose that an inner-loop control system has beed designed and it looks like $\frac{a}{s+a}$. If position is concerned, then a system that looks like $\left[\frac{a}{s(s+a)}\right]$ is obtained. Given this, classical control (root locus) concepts [55] can be used to motivate an outer-loop control structure $K_o = g(s+z)$. To attenuate the effect of high frequency sensor noise, roll-off can be introduced; e.g. $K_o = g(s+z)\left[\frac{b}{s+b}\right]^n$ where $n = 2$ or greater.

6. **What is a suitable outer-loop processor/microcontroller?** Both Arduino Uno and Raspberry Pi II are used for different outer-loop controller implementations.

   Arduino Uno is used for both encoder-based speed inner-loop control and encoder-IMU-based speed-directional outer-loop control. The Raspberry Pi II is used for Position Based Visual Servoing, Image Based Visual Servoing and the encoder-camera-based speed-directional outer loop control.

When taken collectively, the contributions of this thesis are of importance especially to those interested in conducting robotics/*FAME* research.

## 1.4 Organization of Thesis

The remainder of the thesis is organized as follows.

- Chapter 2 (page 17) presents an overview for a general *FAME* architecture describing candidate technologies (e.g. sensing, communications, computing, actuation).

- Chapter 3 (page 21) describes the model for the differential-drive Thunder Tumbler as well as the inner loop controller design. The controller design here is fundamental for the work done in chapters 4 and 5.

- Chapter 4 (page 61) presents two methods to control the differential-drive Thunder Tumbler using information extracted from a camera. Simulation as well as hardware results are presented.

- Chapter 5 (page 109) describes the optimal control minimum time problem along a race track for the differential-drive Thunder Tumbler. An inner loop controller as well as an outer loop controller are used within this chapter.

- Chapter 6 (page 157) summarizes the thesis and presents directions for future robotics/*FAME* research. While much has been accomplished in this thesis, lots remains to be done.

- Appendix A (page 165) contains MATLAB m files and simulink models used to generate the results for this thesis.

- Appendix B (page 184) contains Arduino program files used to generate inner- and outer-loop results for this thesis.

- Appendix C (page 201) contains Python program files (for Raspberry Pi II Model B) used to generate inner- and outer-loop results for this thesis.

- Appendix D (page 224) contains AMPL files used to generate the simulation results of the minimum time optimal control problem for the differential-drive mobile robot around a racetrack.

## 1.5 Summary and Conclusions

In this chapter, an overview of the work presented in this thesis and the major contributions have been provided. A central contribution of the thesis is the use of low-cost multi-capability differential-drive Thunder Tumbler robotic ground vehicle for the design and implementation of inner and outer loop controllers that can be used for robotics/*FAME* research.

Chapter 2

# OVERVIEW OF GENERAL FAME ARCHITECTURE AND $C^4S$ REQUIREMENTS

## 2.1 Introduction and Overview

In this chapter, a general architecture for the general *FAME* research is described. The architecture described attempts to shed light on command, control, communications, computing ($C^4$), and sensing ($S$) requirements needed to support a fleet of collaborating vehicles. Collectively, the $C^4S$ and $S$ requirements are referred to as ($C^4S$) requirements.

## 2.2 FAME Architecture and $C^4S$ Requirements

In this section, a candidate system-level architecture that can be used for a fleet of robotic vehicles[1] is described. The architecture can be visualized as shown in Figure 2.1. The architecture addresses global/central as well as local command, control, computing, communications ($C^4$), and sensing ($C^4S$) needs. Elements within the figure are now described.

- **Central Command: Global/Central Command, Control, Computing.** A global/central computer (or suite of computers) can be used to perform all of the very heavy computing requirements. This computer gathers information from a global/central (possibly distributed) suite of sensors (e.g. GPS, radar, cameras). The information gathered is used for many purposes. This includes temporal/spatial mission planning, objective adaptation, optimization, decision

[1]Here the term robotic vehicle can refer to a ground, air, space, sea or underwater vehicle.

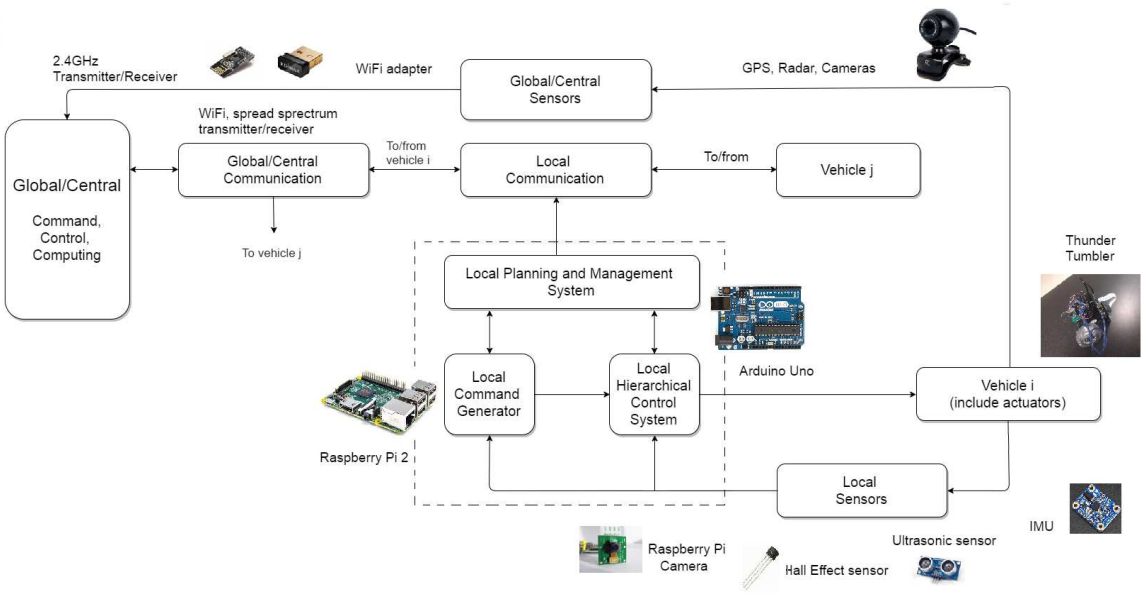


Figure 2.1: FAME Architecture to Accommodate Fleet of Cooperating Vehicles

making (control), information transmission/broadcasting and the generation of commands that can be issued to members of the fleet.

- **Global/Central Sensing.** In order to make global/central decisions, a suite of sensors should be available (e.g. GPS, radar, cameras). This suite provides information about the state of the fleet (or individual members) that can be used by central command.

- **Global/Central Communications.** In order to communicate with members of the fleet, a suite of communication devices must be available to central command. Such devices can include (wideband) spread spectrum transmitters/receivers, WiFi/Bluetooth adapters, etc.

- **Fleet of Vehicles.** The fleet of vehicles can consist of ground, air, space, sea or underwater vehicles. Ground vehicles can consist of semi-autonomous or au-

tonomous robotic vehicles (e.g. differential-drive, rear-wheel drive, etc.). Here, autonomous implies that no human intervention is involved (a longer-term objective). Semi-autonomous implies that some human intervention is involved. Air vehicles can consist of quadrotors, micro/nano air vehicles, drones, other air vehicles and space vehicles. Sea vehicles can consist of a variety of surface and underwater vehicles. Within this thesis the focus is on ground vehicles (e.g. enhanced Thunder Tumbler differential-drive).

- **Local Computing.** Every vehicle in the fleet will (generally speaking) have some computing capability. Some vehicles may have more than others. Local computing here is used to address command, control, computing, planning and optimization needs for a single vehicle. The objective for the single vehicle, however, may (in general) involve multiple vehicles in the fleet (e.g. maintaining a specified formation, controlling the inter-vehicle spacing for a platoon of vehicles). Local computing can consist of a computer, microcontroller or suite of computers/microcontrollers. Within this thesis, Arduino Uno microcontroller (16MHZ ATmega328 processor, 32KB Flash Memory, 14 digital I/O pins, 6 analog inputs, $25) [68]and Raspberry Pi II (900 MHz quad-core ARM Cortex-A7 CPU, 1GB SDRAM, 40 GPIO pins, camera interface, $35) [69] computer boards for local computing on a vehicle are exploited. They are low-cost, well supported (e.g. some high-level software development tools  Arduino IDE and Raspberry Pi II IDLE), and easy to use.

- **Local Sensing.** Local sensing, in general, refers to sensors on individual vehicles. As such, this can involve a variety of sensors. These can include encoders, IMUs (containing accelerometers, gyroscopes, magnetometers), ultra-

sonic range sensors, Lidar, GPS, radar, and cameras. Within this thesis, magnetic encoders(A3144 Hall effect sensor, VELLEMAN 8 mm $\times$ 3 mm magnet, 8 per wheel) [70], IMUs to measure vehicle rotation ( 9DOF, Accelerometer $\pm$ 2,4,8,16g. Gyro $\pm$ 125 $-$ 2000°/*sec*. Compass $\pm$ 13 and $\pm$ 25 Gauss) [72], and Raspberry Pi cameras(2592 $\times$ 1944, 30 fps, 150 MPs, MPEG-4) [71] are used. Lidar, GPS and radar are not used.

- **Local Communications.** Here, local communications refers to how fleet vehicles communicate with one another as well as with central command.

## 2.3 Summary and Conclusions

In this chapter, a general (candidate) *FAME* architecture for a fleet of cooperating robotic vehicles was described. Of critical importance to properly assess the utility of a *FAME* architecture is understanding the fundamental limitations imposed by its subsystems (e.g. bandwidth/dynamic, accuracy/static) [58].

Chapter 3

# INNER LOOP SPEED CONTROL DESIGN FOR DIFFERENTIAL-DRIVE MOBILE GROUND ROBOT

## 3.1 Introduction and Overview

In this chapter the inner loop speed controller for the differential drive mobile ground robot is designed. Also, the hardware used in this thesis for the development of the inner and outer loop controllers is described, for example magnetic wheel encoders for estimating translational speeds, an Inertial Measurement Unit for vehicle posture $\theta$ estimation, camera, Arduino board (used to implement inner loop controller) and Raspberry Pi 2 for more intense computations. Kinematic and dynamic models for the Enhanced Thunder Tumbler are also presented.

## 3.2 Description of Hardware

One central objective of [58] was to show how to take low-cost remote control "toy" vehicles and convert them into intelligent multi-capability robotic platforms. In this thesis the Enhanced Thunder Tumbler robots developed within [58] are used. In this section each component on the robot is described.

1. **Differential-Drive Thunder Tumbler.** It is a differential-drive vehicle with two dc motors - one on left wheel, one on right wheel. Figure 3.1 shows the vehicle used in this thesis.

   More specifically, differential-drive Thunder Tumbler vehicle was augmented with the following: Arduino Motor Shield, Arduino Uno microcontroller board,

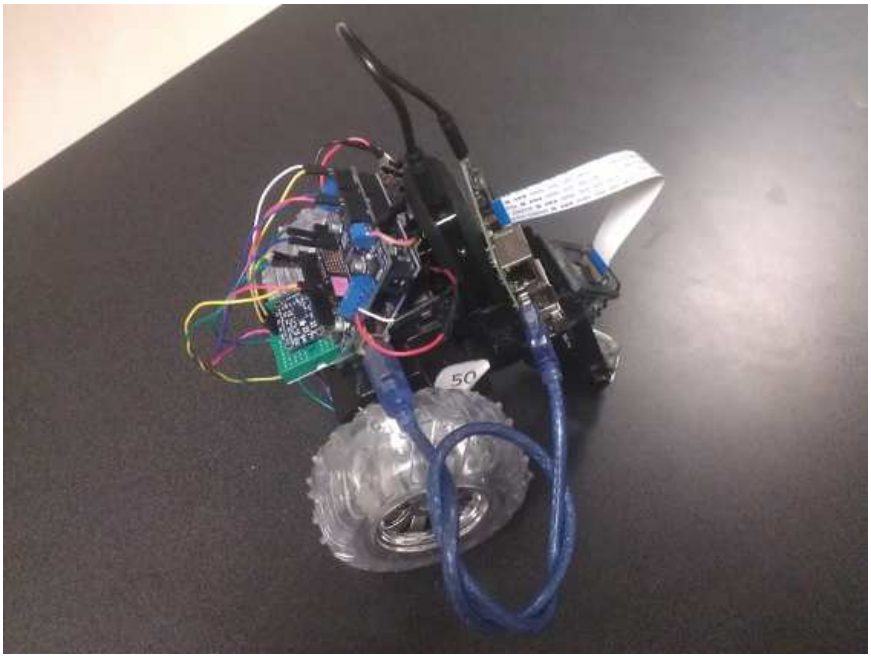

Figure 3.1: Visualization of Fully-Loaded (Enhanced) Thunder Tumbler

Magnetic wheel encoders, Inertia Measurement Unit (IMU), Raspberry Pi 2, Raspberry 5MP camera module,

2. **DC Motors.** Two 6V brushed armature controlled dc motors are on each differential-drive Thunder Tumbler vehicle. The dc motors receive voltage signals from an Arduino motor shield and apply the required torques to each of the Thunder Tumbler's wheels. DC motor parameter values were taken from [59].

3. **Arduino Motor Shield.** An Adafruit Motor/Stepper/Servo Shield for Arduino v2 Kit (v2.3) was used in this thesis (70 × 55 × 10 mm or 2.7" × 2.1" × 0.4," see http://www.adafruit.com/products/1438). It uses a TB6612 MOSFET driver with 1.2 A per channel and 3 A peak current capability, fully dedicated pulse width modulation (PWM) driver chip onboard, polarity protection FET on the power pins, and the serial I2C (inter-integrated circuit) 7-bit ad-

dress computer bus (selectable via jumpers). It can run motors on 4.5V-13.5V dc. Motors are automatically disabled on power up. Five address-select pins permits stacking of 32 shields.

The motor shield receives commands from the Arduino Uno microcontroller board. The shield directly drives the two dc motors - translating Arduino Uno control commands into voltage signals to each dc motor. PWM is used to generate the voltage signal to each dc motor. An 8-bit PWM output (up to 1.6 kHz or about 9600 rad/sec) is provided by the motor shield. Figure 3.2 shows the Adafruit Motor Shield v2.3.

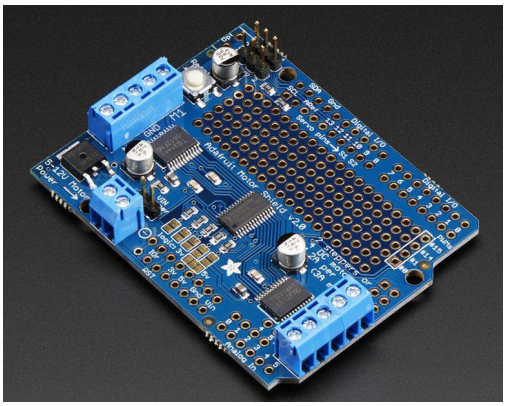

Figure 3.2: Adafruit Motor Shield for Arduino v2.3 - Provides PWM Signal to DC Motors

4. **Arduino Uno Open-Source Microcontroller Board.**

   Arduino Uno microcontroller Board attributes include:

   - 16 MHZ ATmega328 processor, 32 KB Flash Memory, 2 KB SRAM (static random access memory[1], conventionally volatile but exhibits data rema-

[1]SRAM is faster and more expensive than DRAM (dynamic RAM). SRAM is typically used for CPU cache. DRAM must be periodically refreshed and is typically used for a computer's main memory.

nence), 1 KB EEPROM (electrically erasable programmable read only memory), 14 digital I/O pins of which 6 provide PWM output, 6 analog inputs, 8 bit bus, 5V operating voltage, 7-12 V recommended input voltage, 20 mA per I/O pin, 50 mA for 3.3V pin, 68.6 × 53.4 mm, 25 g, USB connection, ICSP (in circuit serial programming) header, power jack, reset button

The Arduino Uno can be seen in Figure 3.3.

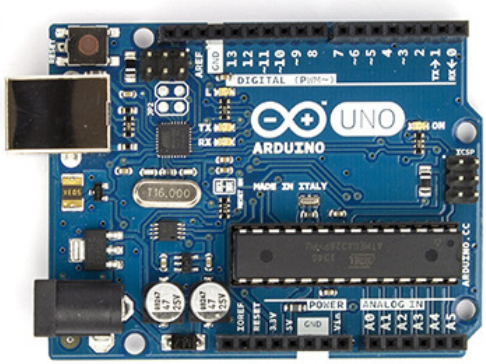


Figure 3.3: Arduino Uno Open-Source Microcontroller Development Board

**Software Support for Arduino.** The Arduino Uno board uses the Arduino open-source IDE (integrated development environment) to write, compile, upload and run code. The Arduino IDE is often called the Arduino Programmer. It runs on various platforms (Windows, Mac OS X, and Linux).

Some of the key Arduino IDE components are as follows:

(1) *Editor.* The editor helps create and edit the text of the sketch (i.e. edit the project code). It actively highlights keywords in order to reduce typing errors.

(2) *Verification System.* The verification system runs through the entire program, verifies that there are no errors, and then compiles the source code into

machine language instructions that can be uploaded to the Arduino board over USB cable.

(3) *Upload System.* The upload system communicates with the Arduino board over the USB cable. It uploads the program into the Arduino's memory.

(4) *Serial Monitor.* The serial monitor allows to send and receive messages from programs running on the Arduino board. This is helpful for testing and debugging.

(5) *Example Sketches.* Example sketches (or project codes) illustrates how to use many different devices.

(6) *Library System.* The library system is a resource which contains, and permits access to, pre-written sketches.

(7) *File System.* The file system is used to save and retrieve sketches.

(8) *Help.* Help includes the complete reference document.

**Arduino Actuation (D-to-A) and Sampling (A-to-D)).** The Arduino actuation and sampling rate is 10Hz or about $60 rad/sec$. Given this, the widely used factor-of-ten rule yields maximum control bandwidth of 6 rad/s. The associated Arduino D-to-A zero order hold (ZOH) has a classic half-sample time delay. This, in turn, places a right half plane (non-minimum phase) zero at $\frac{2}{\Delta} = \frac{2}{0.5T} = \frac{2}{0.5(0.1)} = 40$. The widely used factor-of-ten rule [55], [52] then yields the ZOH-based 4 rad/sec bandwidth constraint.

5. **Magnetic Wheel Hall Effect Sensor-Based Encoders.** A Hall effect sensor (A3144) and magnets (VELLEMAN 8 mm $\times$ 3 mm, 8 per wheel) are used as wheel encoders. Wheel encoders are used for (dead-reckoning) speed/position control. The wheel encoders count the times that a magnet rotates past the

Hall effect sensor. This information is sent to the Arduino Uno which can then calculate/estimate vehicle velocity and translational displacement, vehicle angular velocity and angular displacement. Figure 3.4 shows the Hall effect sensor as well as the magnets.

Figure 3.4: Magnetic Wheel Encoders - Hall Effect Sensors on Left, Magnets on Right

**Magnetic Wheel Encoder Bandwidth Constraint.** The number of samples (or counts) per sec can be obtained as follows by noting that the vehicle speed is related to the wheel angular velocity via $v = r_{wheel}\omega$:

$$\left(8\frac{samples}{rev}\right)\left(\frac{rev}{2\pi\ rad}\right)\omega\ \frac{rad}{sec} = \left(8\frac{samples}{rev}\right)\left(\frac{rev}{2\pi\ rad}\right)\left(\frac{v}{r_{wheel}}\right)\frac{rad}{sec} \quad (3.1)$$

$$= \left(8\frac{samples}{rev}\right)\left(\frac{rev}{2\pi\ rad}\right)\left(\frac{v}{0.05}\right)\frac{rad}{sec} \quad (3.2)$$

$$= 25.46v\ \frac{samples}{sec}\ Hz \quad (3.3)$$

This is the same as $160v\frac{rad}{sec}$. Using the factor-of-ten rule then gives $BW_{encoder_{limit}} = 0.1(160v) = 16v\frac{rad}{sec}$ where the vehicle speed $v$ is measured in m/sec. Note that $16v$ will be larger than the limit of 4 rad/sec (due to half sample D-to-A zero order hold effect with $T = 0.1$) if $v > 0.25\frac{m}{sec}$.

6. **Inertial Measurement Unit (IMU).** In this thesis, the IMU mainly collects the angular velocity information of the robot and sends the information to Arduino Uno. An (Adafruit BNO055 9dof) inertial measurement unit (IMU) is used for directional control (see Figure 3.5).

   The IMU includes 3 acceleration channels, 3 angular rate channels and 3 magnetic field channels. Range features are as follows: ± 2/4/8/16 g acceleration

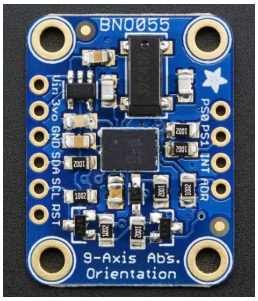


Figure 3.5: Adafruit 9DOF Inertial Measurement Unit (IMU)

full scale, ± 13 gauss (x-, y-axis) and 25 gauss (z-axis) magnetic full scale and ± 125 to 2000 degree/sec angular rate. The rate at which the IMU output angular velocity readings is 100 Hz or approximately 600 rad/sec. The factor-of-ten rule gives a bandwidth constraint of 60 rad/sec.

7. **Servo Motor.** In this thesis, a servo motor is exploited to enable the camera to rotate horizontally (panning) in the implementation of Position Based Visual Servoing. Figure 3.6 shows the servo motor used.


Figure 3.6: Servo Motor

The servo has an operating speed of about 0.1 seconds/60 deg, an operating voltage of 5 volts [73].

8. **Raspberry Pi II Single Board Computer.** The Thunder Tumbler has an onboard Raspberry Pi II Model B single board computer (see Figure 3.7).


Figure 3.7: Raspberry Pi 2 Model B Open-Source Single Board Computer

Raspberry Pi II Model B characteristics include:

- Broadcom BCM2836 with a 900 MHz quad-core ARM Cortex-A7 32-bit CPU and VideoCore IV GPU (see below), 1GB SDRAM (bus synchronous dynamic RAM) at 450 MHz (shared with GPU),
- 40 GPIO (general purpose input/output) pins,
- full HDMI (high definition multimedia interface, EIA/CEA-861) 1.4 port offering 14 HDMI resolutions from 640 × 350 (0.22 MP) to 1920 × 1200 (2.3 MP)),
- Ethernet port (for local area networking based on IEEE 802.3 at 100 Gbps, 400 Gbps by 2017; twisted pair or fiber optic; can surf internet),
- 4 USB 2.0

  (via onboard 5-port USB hub, 480 Mbps, half duplex[2]; can connect keyboard and mouse) ports,
- display interface, slot for micro SDHC (secure digital high capacity) card (15 × 11 × 1 mm, 0.5 grams, minimum sustained read/write speed 17.6 Mbps),

[2]Half-duplex implies "walkie-talkie" like, one-direction-at-a-time, communications. In contrast, full-duplex is bi-directional or "phone like."

- Broadcom VideoCore IV 3D graphics core GPU (250 MHz) with OpenGL ES 2.0 (24 GFLOPS),
- 15-pin MIPI camera interface connector (used with Raspberry Pi camera),
- combined 3.5 mm audio jack and composite video (PAL and NTSC, digital audio via HDMI, integrated interchip sound ($I^2S$, serial bus interface standard for connecting digital audio devices),

The Raspberry Pi II Model B is a full computer with a GPU and 1080p full HD video capability.

**Software Support for Pi.** Software support is important in order to minimize an often significant amount of low-level programming overhead that most embedded system developers would prefer to avoid.

- **Python IDLE.** Raspberry Pi uses the open-source Python IDLE (Integrated DeveLopment Environment) to write, upload and run code. IDLE is coded in Python using the so-called "tkinter" GUI toolkit. It works on standard platforms such as Windows, Unix, and Mac OS X.
- **Interpreted Python.** The Python shell window implements an interactive interpreter. An interpreter is a computer program that executes instructions without previously compiling them into native machine language for the host CPU. Python, like Perl and MATLAB, translate source code into some efficient intermediate representation that is immediately executed. As expected, interpreted programs run more slowly and less efficiently than compiled programs.

  *Interpreter Advantages.* Interpreted languages often offer the following advantages over compiled implementations: (1) platform independence,

(2) reflection (ability to modify structure and behavior of code at runtime), (3) dynamic typing (verifies type safety of program at runtime), (4) smaller executable program size, and (5) dynamic scoping (used by few modern languages).

*Interpreter Deficiencies.* Interpreters have the following deficiencies: (1) no static type-checking (as done by compilers at compile time) and hence less reliable code, (2) susceptible to code injection attacks, (3) slower execution compared to direct native machine code execution on the host CPU, and (4) source code can be read and copied.

- **Key Python Attribute: Facilitates Good Code Writing.** Python is a widely used, general-purpose, high-level programming language that emphasizes code readability. Its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. Python provides constructs to facilitate the writing of clear small or large programs.

- **Communication Between Pi and Arduino During Robot Operation.** Python and USB (Serial) communication were used between the Pi and Arduino. There are many ways to establish communication between the Raspberry Pi and the Arduino such as using the GPIO and Serial pins or using I2C communication (using the SCL-clock and SDL-data pins). The easiest way to get the two devices talking is to use the micro USB cable that comes with the Arduino Uno. By using the PySerial library package, Python installed on the Pi can be used to read from and write to Arduino's serial port.

**Summary of Arduino and Raspberry Pi Use.** Arduino was used to implement the $(\omega_r, \omega_l)$ inner-loop control law. Section 3.4 on page 48 describes the relevant theory. The associated Arduino code can be found on page 185. The Raspberry Pi II used the $M^{-1}$ matrix to translate $(v_{ref}, \omega_{ref})$ commands into $(\omega_{r_{ref}}, \omega_{l_{ref}})$ commands for the inner-loop. The associated Python code can be found on page 201. It thus follows that the Pi was used to implement some inner-loop functionality. Arduino was used to implement outer-loop functions as well as the low-level inner-loop feedback control functions discussed above.

(1) The Raspberry Pi was used to implement outer-loop 1: Position Based Visual Servoing. Section 4.4 on page 69 describes the relevant theory. The associated Python code can be found on page 201.

(2) The Raspberry Pi was used to implement outer-loop 2: Image Based Visual Servoing. Section 4.5 on page 76 describes the relevant theory. The associated Python code can be found on page 201.

(3) The Raspberry Pi and Arduino were used to implement outer-loop 3: camera-based and noncamera-based $(v, \theta)$ cruise control for the minimum time optimal control problem respectively. Section 5.2.4 on page 129 describes the relevant theory. The associated Python code can be found on page 219. The associated Arduino code can be found on page 195.

Which computing unit is used depends on the demo being conducted and hence on the sensors being used. The Arduino is involved in all demos because it implements the $(\omega_r, \omega_l)$ inner-loop control law (see Section 3.4 on page 48). As discussed above, the Pi is always involved to generate the required $(\omega_r, \omega_l)$ commands to the inner-loop. Whenever the camera is used within an outer-loop, the Pi was used. When the camera is not used within an outer-loop, the Pi plays no outer-loop role.

**Raspberry 5MP Camera Module.** The Enhanced Thunder Tumbler has an onboard Raspberry Pi 5MP camera. The camera contains a 5MP Omnivision 5647 sensor in a fixed focus module which enables 2592×1944 pixel static images. It also supports 1080p30 (30 fps), 720p60 and 640×480p60/90 MPEG-4 video. The camera module plugs directly into the Pi's 15 pin MIPI (MIPI Alliance) camera serial interface (CSI) via 15 pin ribbon cable. The CSI bus supports very high data rates to carry data directly to the Pi's Broadcom VideoCore4 BCM2835 system on a chip (SoC) processor (GPU) which uses a 32 bit RISC (reduced instruction set computing) ARM1176 (700 MHz) core/processor. The camera module collects image information and sends it to the onboard Raspberry Pi 2. The Raspberry Pi camera can support the following frame rates: up to 15 fps at a resolution of 2592 × 1944 (5 MP), 30 fps at 1980 × 1080 (2.1 MP, this is 1080p30), 42 fps at 1296 × 972 (1.3 MP), and 60 fps at 640 × 480 (0.31 MP).

A hardware components list for an enhanced Thunder Tumbler is given in Table 3.1.

| **Product** | **Quantity** | **Price ($)** |
|---|---|---|
| Thunder Tumbler Vehice | 1 | $10 |
| Raspberry Pi 2 Model B | 1 | $40 |
| Arduino Uno | 1 | $12.19 |
| Adafruit Motor Shield | 1 | $22.50 |
| Raspberry Pi 5MP Camera | 1 | $25 |
| Camera Holder | 1 | $5 |
| Power Supply for Raspberry Pi | 1 | $10 |
| Power Supply for Arduino | 4 | $6.75 |
| Magnetic Wheel Encoders | 2 | $4.40 |
| Magnets (Velleman) | 16 | $9.6 |
| BNO055 9dof IMU | 1 | $34.95 |
| Servo motor | 1 | $2.3 |
| Metal ball caster | 1 | $2.50 |
| **Total Price** | | **$185.19** |

Table 3.1: Hardware Components for Enhanced Differential-Drive Thunder Tumbler Robotic Vehicle

## 3.3 Differential-Drive Ground Robotic Vehicle Model

Many mobile robots use a differential-drive drive mechanism. Such a mechanism involves two rear wheels that are independently controlled via torque-generating dc motors. The inputs to the dc motors are voltages. Within this thesis, the motors are assumed to be identical in order to simplify the presentation. In practice, motor differences must be accounted for. This, in part, is addressed by the motor control laws being employed. Within this section, the TITO LTI model that was presented within [5] is examined. This model was used for control law design within the MS thesis [57]. The ground mobile robot kinematics are first discussed.

The robot dynamics are then examined - first without and then with the dc motor dynamics. It is useful to define key robot variables and parameters to be used throughout the section. This is done within Table 3.2.

| **Parameters** | **Definition** | **Nominal Values** |
|---|---|---|
| $m$ | Mass (Fully Loaded, Enhanced Vehicle) | 0.82 kg |
| $m_o$ | Mass (Not Loaded, Original Vehicle) | 0.55 kg |
| $I_z$ | Vehicle Moment of Inertia (Estimated using cuboid, $\frac{1}{12}m(width^2 + lenght^2)$) | 0.0047 $Kgm^2$ |
| $r$ | Wheel Radius | 0.05 m |
| $d_w$ | Distance between Two Rear Wheels | 0.14 m |
| $L_a$ | Armature Inductance | 261 $\mu$ H |
| $R_a$ | Armature Resistance | 0.86 $\Omega$ |
| $K_b$ | Back EMF Constant | 0.0031 V/(rad/sec) |
| $K_t$ | Torque Constant | 0.0031 Nm/A |
| $\beta$ | Speed Damping Constant | $8.15e-7$ Nms |
| $I$ | Moment of Inertia of Motor-Shaft System | $3.2e-6$ Nms |
| $v_{max}$ | Maximum Observed Speed (Enhanced Vehicle) | 2.3 m/sec |
| $v_{max_o}$ | Maximum Observed Speed (Original Vehicle) | 4.5 m/sec |
| $a_{max}$ | Maximum Acceleration (Enhanced) | 1.5 $m/sec^2$ |
| $\omega_{wheel_{max}}$ | Maximum Angular Wheel Velocity (Enhanced) | 46 rad/sec |
| $e_{a_{max}}$ | Maximum Motor Voltage | 6 V |

Table 3.2: Thunder Tumbler Nominal Parameter Values and Characteristics

### 3.3.1 Differential-Drive Robot Kinematics

Figure 3.8 can be used to understand the kinematics of a differential-drive ground robot [44].

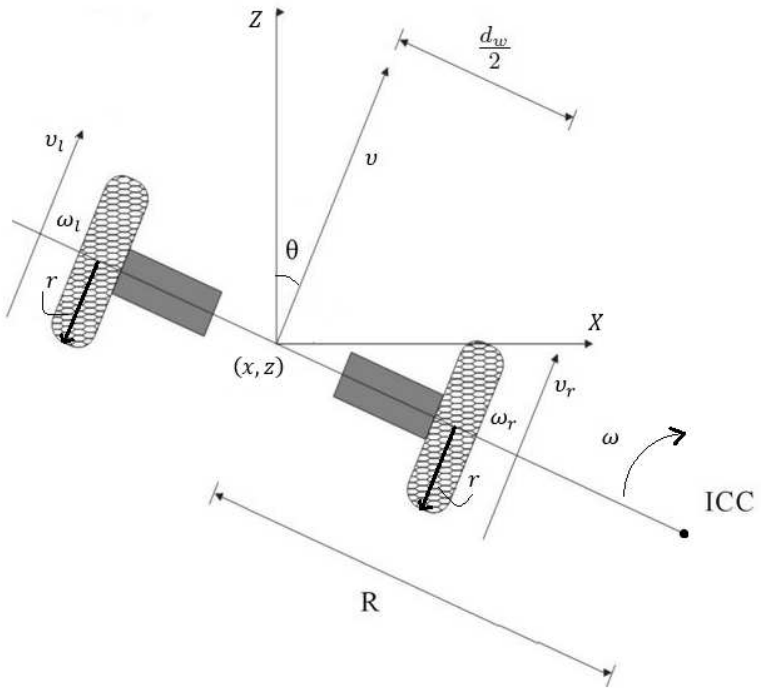


Figure 3.8: Visualization of Differential-Drive Mobile Robot

The point that the robot rotates about at a given instant in time is called the instantaneous center of curvature (ICC) [44]. If $(x, z)$ denotes the planar inertial coordinate of the robot and $\theta$ denotes the direction of the robot's longitudinal body axis with respect to the $z$-axis, then the following nonlinear kinematic model is obtained:

$$\begin{bmatrix} \dot{x} \\ \dot{z} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \sin\theta \\ \cos\theta \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega \tag{3.4}$$

where

$$v = \sqrt{\dot{x}^2 + \dot{z}^2} \tag{3.5}$$

denotes the translational speed of the robot and $\omega = \dot{\theta}$ denotes its angular speed. Within the above very simple (and intuitive) model, $v$ can $\omega$ can be thought of as inputs (or controls). This is not intuitive - especially to a controls person. Why? $v$ and $\omega$ cannot be instantaneously generated because of real-world mass-inertia effects. In practice, $v$ and $\omega$ are generated by applying voltages to the left and right wheel dc motors.

At this point, it is instructive to relate the $(v, \omega)$ to the angular velocities $(\omega_L, \omega_R)$ of the left and right rear wheels. Why? The idea here, is that if one can precisely control $(\omega_L, \omega_R)$ , then one will be able to precisely control $(v, \omega)$. The desired relationships are as follows:

$$v = \left[\frac{r(\omega_R + \omega_L)}{2}\right] \qquad\qquad \omega = \left[\frac{r(\omega_L - \omega_R)}{d_w}\right] \tag{3.6}$$

where $r$ denotes the wheel radius and $d_w$ denotes the distance between the rear wheels. Given the latter, it follows that the distance between the vehicle longitudinal body axis and the wheel longitudinal center lines is simply $L = \frac{d_w}{2}$ (as shown in Figure 3.8). Both $r$ and $d_w$ are assumed to be constant. Within Figure 3.8, the point vehicle coordinate $(x, z)$ is located on the vehicle's longitudinal body axis directly in between the two rear wheels.

To derive the above relationships, one can proceed as follows. Let $v_l$ and $v_r$ denote the left and right wheel translational speeds along the ground. If $R$ denotes the "signed" distance from the $(x, z)$ coordinate of the vehicle to the ICC, then it follows that

$$(R + d_w/2)\omega = v_l \qquad\qquad (R - d_w/2)\omega = v_r \tag{3.7}$$

From these equations, it follows (after some algebra) that

$$R = \frac{d_w}{2}\left[\frac{v_l + v_r}{v_l - v_r}\right] \qquad \omega = \left[\frac{v_l - v_r}{d_w}\right] \tag{3.8}$$

Next, note that

$$v = R\omega \qquad v_l = r\omega_L \qquad v_r = r\omega_R \tag{3.9}$$

Substituting $v_l = r\omega_L$ and $v_r = r\omega_R$ into $\omega = \frac{v_l - v_r}{d_w}$, yields the relation $\omega = \frac{r(\omega_L - \omega_R)}{d_w}$. Substituting $R = \frac{v}{\omega}$ and $\omega = \frac{v_l - v_r}{d_w}$ into $R = \frac{d_w}{2}\frac{v_l + v_r}{v_l - v_r}$ yields the relation $v = \frac{v_r + v_l}{2}$. Substituting $v_l = r\omega_L$ and $v_r = r\omega_R$ into this relation then yields the desired result $v = \frac{r(\omega_R + \omega_L)}{2}$. This completes the derivation.

It is convenient to rewrite the above relations in vector-matrix form as follows:

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = M \begin{bmatrix} \omega_R \\ \omega_L \end{bmatrix} \qquad M = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ -\frac{r}{d_w} & \frac{r}{d_w} \end{bmatrix} \tag{3.10}$$

Again, the importance of the above relation stems from the fact that if one can control $(\omega_L, \omega_R)$ well, one will be able to control $(v, \omega)$ well - the latter being the prime directive of this chapter.

### *3.3.2 Differential-Drive Robot Dynamics*

In order to more accurately represent the system, a dynamical model is considered - one that captures mass-inertia effects. The following intuitive representation of the model comes from [49]:

$$\begin{bmatrix} \dot{x} \\ \dot{z} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \sin\theta & 0 \\ \cos\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.11}$$

$$\begin{bmatrix} \dot{v} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} \frac{1}{m} & 0 \\ 0 & \frac{1}{I} \end{bmatrix} \begin{bmatrix} F \\ \tau \end{bmatrix} \tag{3.12}$$

$$\tan\theta = \frac{\dot{x}}{\dot{z}} \tag{3.13}$$

where $F$ represents the applied translational force along the vehicles longitudinal body axis, $\tau$ represents the applied torque about the vertical $z$ axis passing through the point $(x, z)$, $m$ denotes the mass of the vehicle and $I_z$ denotes its moment of inertia about the vertical $z$ axis passing through the point $(x, z)$. From the above, it is seen that the dynamical model consists of the following five equations: three kinematic model equations within the matrix-vector equation (3.11), two Newtonian dynamical equations within the matrix-vector equation (3.12), and the no slipping (non-holonomic) constraint within equation (3.13). It should be noted that in practice, the force $F$ and torque $\tau$ are generated by the two dc motors on the rear wheels. This shall become evident within the subsections that follow below.

As suggested above, the kinematic model neglects dynamic mass-inertia effects. As such, the kinematic model is just an approximation to the dynamic model. The kinematic model is a good approximation to the dynamical model when $(v, \omega)$ can be generated quickly. Intuitively, this occurs when $m$ and $I$ are sufficiently small or the inner $(v, \omega)$ loop has a sufficiently large bandwidth.

Finally, it is important to note the relationship between $(F, \tau)$ and the left-right motor torques $(\tau_l, \tau_r)$. The desired relationship is similar in form to the angular velocity relationships within equation 3.6 and is given by

$$F = \left[\frac{\tau_r + \tau_l}{r}\right] \qquad\qquad \tau = \left[\frac{d_w(\tau_l - \tau_r)}{2r}\right] \tag{3.14}$$

Here, $\tau_l$ and $\tau_r$ represent the torques acting on the left and right wheels, respectively.

Next, the motor (actuator) dynamics are discussed. Ultimately, the motors are responsible for producing the wheel torques $(\tau_l, \tau_r)$ and hence the associated pair

$(F, \tau)$. The latter, of course are directly responsible for producing the vehicle speeds $(v, \omega)$.

### *3.3.3 DC Motor (Actuator) Dynamics*

There are two classes of DC motors: (1) armature-current controlled and (2) field-current controlled [55]. Similar to [58], focus is made on the former in this thesis; i.e. armature-current controlled dc motors. The dynamics for a DC motor can be visualized as shown within Figure 3.9. The associated equations are as follows:

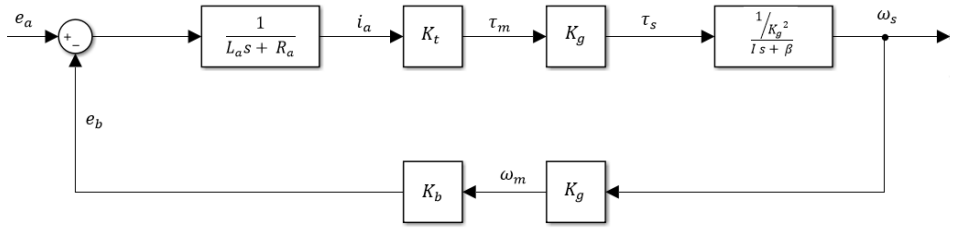


Figure 3.9: Visualization of DC Motor Speed-Voltage Dynamics

*Armature Equation*:

$$e_a = L_a \frac{di_a}{dt} + R_a i_a + e_b \tag{3.15}$$

*Back EMF Equation*:

$$e_b = K_b K_g \omega_s \tag{3.16}$$

*Torque Equation*:

$$\tau_s = K_t K_g i_a \tag{3.17}$$

*Load Equation*:

$$I\dot{\omega}_s + \beta\omega_s = \frac{\tau_s}{{K_g}^2} \tag{3.18}$$

Here, $e_a$ represents the applied armature voltage. This is the control input for an armature controlled DC motor. Other relevant variables are as follows: $i_a$ represents the armature current, $e_b$ represents the back emf, $\tau_s$ represents the torque exerted by the

motor on the motor shaft-load system, $\omega_s$ represents the motor shaft angular speed. Relevant motor parameters are as follows: $L_a$ represents the armature inductance (often negligibly small in many applications), $R_a$ represents the armature resistance, $K_b$ represents the back emf motor constant, $K_t$ represents the motor torque constant, $K_g$ represents the gearbox ratio of motor shaft-load system $\beta$ represents a load-motor speed rotational damping constant, and $I$ represents the moment of inertia of the motor shaft-load system.

From the above, one can obtain the transfer function from the input voltage $e_a$ to the angular speed $\omega_s$:

$$\frac{\omega_s}{e_a} = \left[ \frac{\frac{K_t}{K_g}}{(Is+\beta)(L_a s + R_a) + K_t K_b} \right] \tag{3.19}$$

Given the above, some observations are in order. The motor speed transfer function is generally second order. If the armature inductance $L_a$ is negligibly small (i.e. $\omega_s L_a << R_a$ over the operational bandwidth), then the motor speed transfer function becomes first order. In such a case, the following speed-voltage transfer function approximation is obtained:

$$\frac{\omega_s}{e_a} \approx \left[ \frac{\frac{K_t}{K_g}}{(Is+\beta)(R_a) + K_t K_b} \right] \left[ \frac{R_a}{L_a s + R_a} \right] \tag{3.20}$$

In such a case, the dominant motor pole becomes $s \approx -\left(\frac{R_a\beta + K_t K_b}{R_a I}\right) = -\frac{\beta}{I} - \frac{K_t K_b}{R_a I}$ and the associated inductance pole becomes large and given by $s \approx -\frac{R_a}{L_a}$. Given this, the motor response is faster for larger ($\beta$, $K_t$, $K_b$) and smaller ($I$, $R_a$). If the armature inductance is neglected, then the speed-voltage transfer function becomes first order. Generally, $K_t = K_b$.

### 3.3.4 Robot TITO LTI Model with Actuator Dynamics

In this section, the ideas presented above are combined in order to get state space representation TITO LTI model for the differential-drive vehicle. This model, taken from [5], was used within [57] and [58] for inner-loop control design. The TITO LTI model from motor voltages $(e_{aR}, e_{aL})$ to the wheel angular velocities $(\omega_R, \omega_L)$ can be visualized as shown within Figures 3.10-3.11.

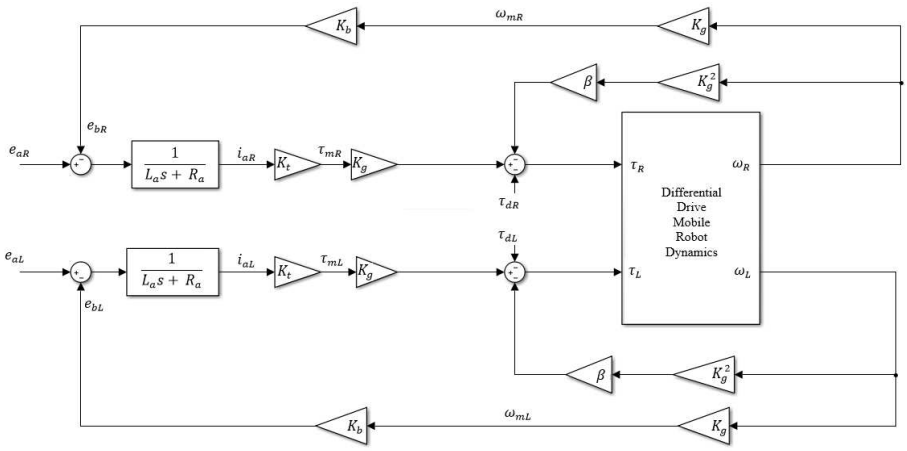


Figure 3.10: TITO LTI Differential-Drive Mobile Robot Dynamic Model with Actuators

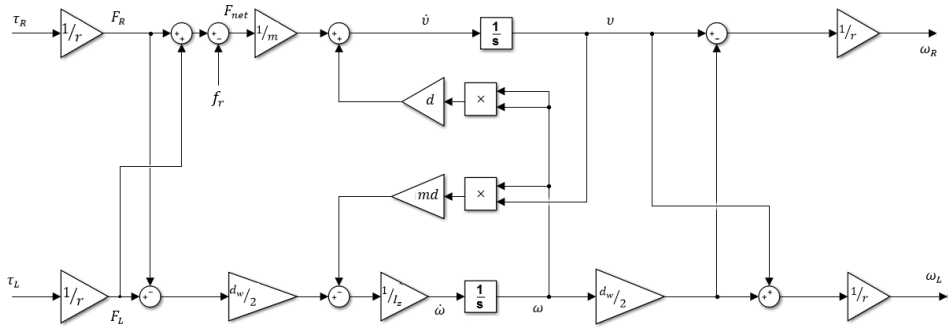


Figure 3.11: Differential-Drive Mobile Robot Dynamic Model

The associated fourth order TITO LTI state space representation is given by

$$\dot{x} = Ax + Bu \qquad\qquad y = Cx + Du \tag{3.21}$$

where $x = [\ v\ \ \omega\ \ i_{aR}\ \ i_{aL}\ ]^T$, $y = [\ \omega_R\ \ \omega_L\ ]^T$, $u = [\ e_{aR}\ \ e_{aL}\ ]^T$,

$$A = \begin{bmatrix} \frac{-2\beta K_g^2}{mr^2} & 0 & \frac{K_g K_t}{mr} & \frac{K_g K_t}{mr} \\ 0 & \frac{-\beta {K_g}^2 {d_w}^2}{2I_z r^2} & \frac{K_g K_t d_w}{2I_z r} & \frac{-K_g K_t d_w}{2I_z r} \\ \frac{-K_g K_b}{L_a r} & \frac{-K_g K_b d_w}{2L_a r} & \frac{-R_a}{L_a} & 0 \\ \frac{-K_g K_b}{L_a r} & \frac{K_g K_b d_w}{2L_a r} & 0 & \frac{-R_a}{L_a} \end{bmatrix} \tag{3.22}$$

$$B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{1}{L_a} & 0 \\ 0 & \frac{1}{L_a} \end{bmatrix} \tag{3.23}$$

$$C = \begin{bmatrix} \frac{1}{r} & \frac{d_w}{2r} & 0 & 0 \\ \frac{1}{r} & \frac{-d_w}{2r} & 0 & 0 \end{bmatrix} \tag{3.24}$$

$$D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{3.25}$$

Here, $(i_{aL}, i_{aR})$ represent left and right motor armature currents, $v$ is the vehicle's translational velocity (directed along the direction $\theta$), $\omega$ is the vehicle's angular velocity, $(\omega_L, \omega_R)$ represent left and right vehicle wheel angular velocities, $(e_{aL}, e_{aR})$ represent left and right motor armature voltage inputs. The latter are the robot's control inputs. Relevant system parameters are as follows: $m$ is the vehicle mass, $d_w$

is the distance between the wheels, $r$ is the vehicle wheel radius, $I_z$ is the vehicle's moment of inertia, $\beta$ represents a load-motor speed rotational damping constant, $K_b$ represents a back emf constant, $K_t$ represents a torque constant, $K_g$ represents the gearbox ratio of motor shaft-load system $R_a$ represents armature resistance, and $L_a$ represents armature inductance (often negligibly small). It should be noted that differences in the motor properties is a practical concern. This has not been captured in the above model. It shall not be addressed within this thesis. Addressing such uncertainty will be the subject of future work.

**Frequency Response Properties.** The singular values for the above system and the associated low frequency approximation are plotted within Figure 3.12 for the nominal parameter values given within Table 3.2 (taken from [59]). Note that the singular values at dc match one another. This is because from each input, the motor-vehicle $(\omega_R, \omega_L)$ system looks the same.

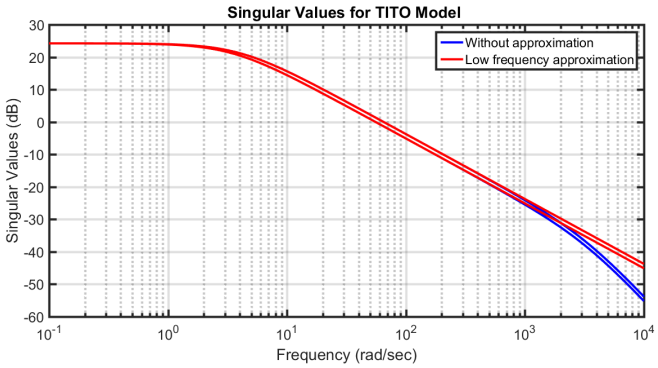


Figure 3.12: Robot Singular Values (Voltages to Wheel Speeds) - Including Low Frequency Approximation

The plot in Figure 3.12 suggests that the low frequency approximation (red, with a 20 dB/decade high frequency roll-off) is a good approximation for the system. The relatively high system gain at low frequencies will help achieve good low frequency

command following and low frequency disturbance attenuation (in principle, without too much control action).

To better examine the coupling in the $(\omega_R, \omega_L)$ system, its frequency response is plotted in Figure 3.13. The figure clearly shows that the off-diagonal elements peak around 4 rad/sec and that the coupling disappears at dc. This low frequency behavior, as well as the first order low frequency behavior of the diagonal elements, provides substantive motivation for a decentralized PI control law; i.e. the use of identical PI controllers for each motor.

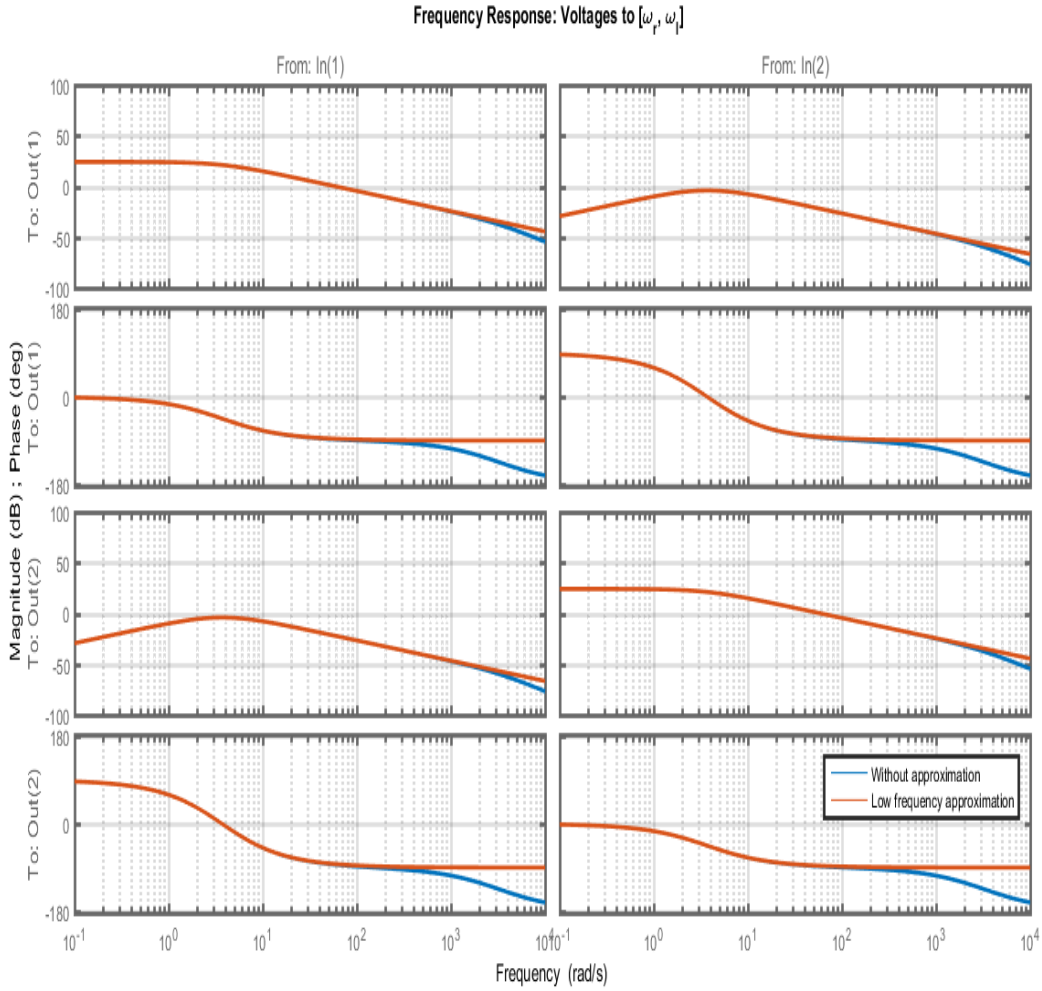


Figure 3.13: Robot Frequency Response (Voltages to Wheel Speeds) - Including Low Frequency Approximation

### Off-Ground Motor Step Response

In Figure 3.14 the DC motor off-ground step response is plotted when the input voltage is 1.5V. The ripple is due to the fact that the encoder resolution is approximately 3.93 rad/sec.

A transfer function corresponding to this hardware result reads as:

$$P_{offground} = \frac{\omega}{e} = 14.2 \left[\frac{4.5}{s + 4.5}\right] \tag{3.26}$$

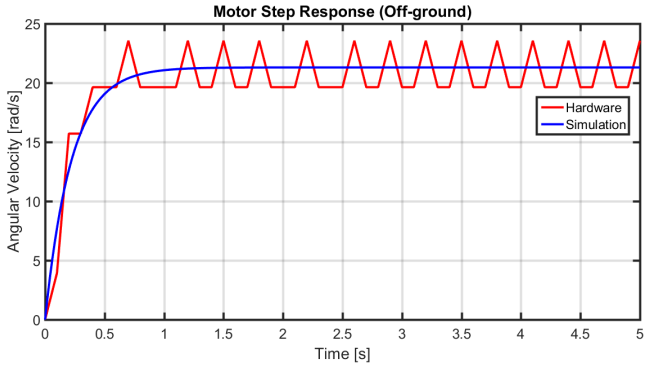


Figure 3.14: DC Motor Output $\omega$ Response to 1.5V Step Input

### Estimated Transfer Function for Differential-Drive Mobile Robot.

An on-ground test was carried out in order to estimate a transfer function for the vehicle. Figure 3.15 shows the hardware measured response to a 2.42 V input voltage.

According to the experimental result shown in Figure 3.15, the following estimated transfer function is obtained(from voltage to angular velocity):

$$P_{inner} = \frac{\omega}{e} = 5.4954 \left[\frac{1.73}{s + 1.73}\right] \tag{3.27}$$

The simulated step response for the plant $P_{inner}$ is shown in Figure 3.15. The two DC motors for the vehicle are assumed to be identical. The TITO LTI $(\omega_R, \omega_L)$

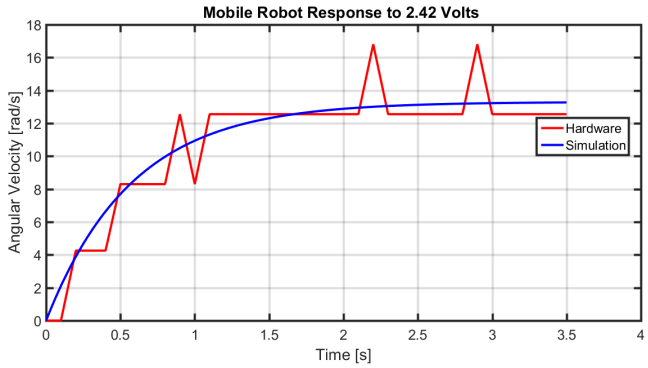


Figure 3.15: DC Motor Output $\omega$ Response to 2.42V Step Input

vehicle-motor model is assumed to be diagonal.

This model is different from the one shown in Figures 3.12-3.13, due to unmodeled dynamics such as stiction, backlash and deadzone. A more complete model shall be investigated in future work.

For the remainder of this chapter the following approximation for the inner loop vehicle-motor $(\omega_R, \omega_L)$ plant will be used:

$$P_{[\omega_R,\omega_L]} \approx 5.4954 \left[\frac{1.73}{s+1.73}\right] \times I_{2\times 2} \tag{3.28}$$

**Frequency Response Analysis for Diagonal (Decoupled) System.** Given the estimated model above in equation (3.27), the associated decoupled vehicle-motor frequency response is shown in Figure 3.16.

## 3.4 Inner-Loop Speed Control Design and Implementation

In this section, the $(\omega_R, \omega_L)$ and $(v, \omega)$ inner-loop control design for the differential drive Thunder Tumbler is described. For this basic inner-loop control modality,

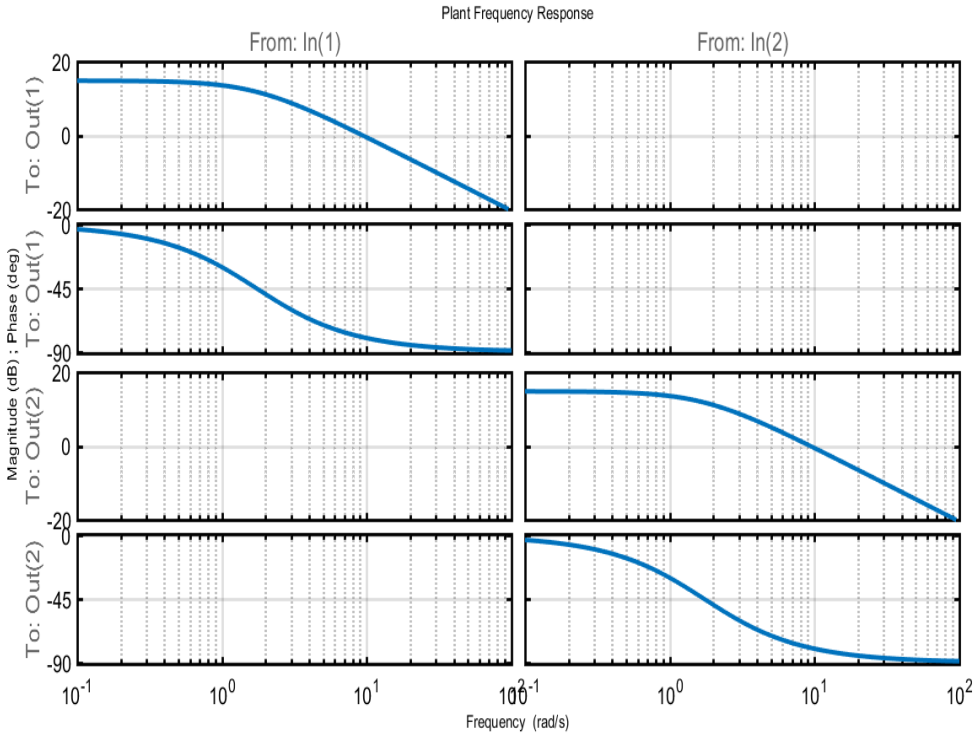


Figure 3.16: Magnitude Response for Vehicle-Motor - Decoupled $(\omega_R, \omega_L)$ Model

the angular velocity of each vehicle wheel is estimated/approximated by exploiting the magnetic pulse counts picked up by the two wheel encoders during a $T$ seconds sampling window. This results in the following estimate for $(\omega_R, \omega_L)$:

$$\omega_R \approx \frac{2\pi n_r}{8T} = 7.854\ n_r\ (rad/sec) \qquad \omega_L \approx \frac{2\pi n_l}{8T} = 7.854\ n_l\ (rad/sec) \quad (3.29)$$

where

- $T = 0.1$ sec (100 msec or 10 Hz) was the chosen sampling (and actuation) time.
- $n_r$ is the number of counts measured by the magnetic encoder (Hall effect sensor) on the right wheel (with 8 pulses/counts per rotation[3]),

[3]Actually in hardware, Arduino provides 16 pulses per rotation, i.e. it counts the rising and falling edges caused by the 8 magnets. Hence the resolution is reduced by half (0.5*7.854) = 3.927

- $n_l$ is the number of counts measured by the magnetic encoder (Hall effect sensor) on the left wheel (with 8 counts per rotation).

Note that as the number of magnets used on a wheel is increased, then the constant 7.854 would decrease. The vehicle translational and rotational velocities $(v, \omega)$ are then estimated from the above $(\omega_R, \omega_L)$ estimates as follows:

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = M \begin{bmatrix} \omega_R \\ \omega_L \end{bmatrix} \qquad M = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ -\frac{r}{d_w} & \frac{r}{d_w} \end{bmatrix} \qquad M^{-1} = \begin{bmatrix} \frac{1}{r} & \frac{-d_w}{2r} \\ \frac{1}{r} & \frac{d_w}{2r} \end{bmatrix} \tag{3.30}$$

and

$$v = \left( \frac{r(\omega_R + \omega_L)}{2} \right) \approx 0.392 \left( \frac{n_r + n_l}{2} \right) \; m/sec \tag{3.31}$$

$$\omega = \left( \frac{r(\omega_L - \omega_R)}{d_w} \right) \approx 2.805 \left( n_l - n_r \right) \; rad/sec \tag{3.32}$$

where $r = 0.05$ m is the radius of each wheel and $d_w = 0.14$ m is the distance between the rear wheels. The above suggests that a single missed count could result in a 0.3928 m/sec translation velocity error or a 2.805 rad/sec rotational velocity error. As the number of magnets used on a wheel is increased, these errors would decrease.

**Control Design: PI with One Pole Roll-Off and Command Pre-filter.** Based on the simple (decoupled first order) LTI model obtained in the previous section in equation (3.27)

$$P_{inner} = \frac{\omega}{e} = 5.4954 \left[ \frac{1.73}{s + 1.73} \right] \tag{3.33}$$

a PI controller with roll-off and pre-filter is designed. The controller has the form (PI plus roll-off):

$$K_{inner} = \frac{g(s + z)}{s} \left[ \frac{100}{s + 100} \right] \tag{3.34}$$

This $K_{inner}$ will be used to drive each dc motor [4] on the vehicle.

[4] Actually, the digital implementation of $K_{inner}$ will be used to drive the Arduino shield. The shield will then drive the dc motors.

Here, phase margin ($PM$) of 60 deg and a unity-gain crossover frequency ($\omega_g$) of $2\,rad/sec$ are used as design parameters. The open loop transfer function $L$ is given by

$$L = P_{inner}K_{inner} = \frac{g(s+z)}{s}\frac{9.507}{s+1.73}\left[\frac{100}{s+100}\right] \tag{3.35}$$

From the phase margin equation $PM = 180^\circ + \angle L(j\omega_g)$ the value of the zero $z$ is computed, i.e.

$$\begin{aligned}
PM &= 180^\circ - 90^\circ + \tan^{-1}\left(\frac{\omega_g}{z}\right) - \tan^{-1}\left(\frac{\omega_g}{1.73}\right) - \tan^{-1}\left(\frac{\omega_g}{100}\right) && (3.36)\\
&= 90^\circ + \tan^{-1}\left(\frac{\omega_g}{z}\right) - \tan^{-1}\left(\frac{\omega_g}{1.73}\right) - \tan^{-1}\left(\frac{1}{50}\right) && (3.37)\\
\tan^{-1}\left(\frac{\omega_g}{z}\right) &= PM - 90^\circ + \tan^{-1}\left(\frac{\omega_g}{1.73}\right) + \tan^{-1}\left(\frac{1}{50}\right) && (3.38)\\
&= 60^\circ - 90^\circ + 49.14^\circ + 1.145^\circ = 20.285 && (3.39)\\
\frac{\omega_g}{z} &= \tan(20.285) && (3.40)\\
z &= \frac{\omega_g}{\tan(20.285)} && (3.41)\\
z &= 5.411 && (3.42)
\end{aligned}$$

Now $g$ is obtained by knowing that $|L(j\omega_g)| = 1$.

$$\begin{aligned}
&\frac{g\sqrt{\omega_g^2+z^2}}{\omega_g}\frac{9.507}{\sqrt{\omega_g^2+1.73^2}}[1] = 1\\
&g = \frac{\omega_g\sqrt{\omega_g^2+1.73^2}}{9.507\sqrt{_g^2+z^2}}\\
&g = \frac{5.288}{54.84}\\
&g = 0.096
\end{aligned} \tag{3.43}$$

This values of $g$ and $z$ yields

$$\Phi_{actual}(s) \approx s(s+1.73) + 9.507g(s+z) = s^2 + 2.6426s + 4.938. \tag{3.44}$$

A reference command pre-filter

$$W = \frac{z}{s+z} \tag{3.45}$$

will ensure that the overshoot to a step reference command approximates that dictated by the second order theory. That is, the following single channel (SISO) map from commanded wheel speed to actual wheel speed is obtained: $T_{ry_{wheel\ speeds}} = W\left[\frac{P_{inner}K_{inner}}{1+P_{inner}K_{inner}}\right] \approx \frac{4.938}{s^2+2.6426s+4.938}$.

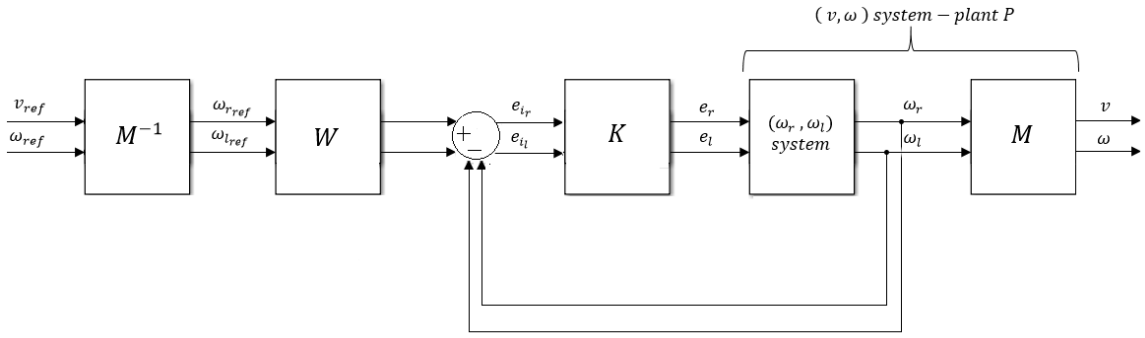


Figure 3.17: Visualization of $(v, \omega)$ and $(\omega_r, \omega_l)$ Inner-Loop Control

The inner-loop control system can be visualized as shown in Figure 3.17. $(v, \omega)$ are commanded but not directly measured. Within Figure 3.17, the matrix $M$ is a $2 \times 2$ vehicle-wheel speed map that relates the vehicle translational-rotational velocities $(v, \omega)$ to the wheel angular velocities $(\omega_R, \omega_L)$; i.e. see equation (3.30) (page 50). Although only the wheel encoder count information is fed back within the physical inner-loop hardware implementation, the system outputs $v$ and $\omega$ were estimated (computed) using wheel encoder counts in accordance with equations (3.31)-(3.32).

**Reference to Output $T_{ry}$ $(v, \omega)$ Map.** From Figure 3.17, it follows that one can use the relationships in equation (3.30) to get the model-based closed loop map from

references $(v_{ref}, \omega_{ref})$ to outputs $(v, \omega)$. Doing so yields the following TITO LTI map:

$$T_{ry} = MWPK(I + PK)^{-1}M^{-1} \approx \left[\frac{4.938}{s^2 + 2.6426s + 4.938}\right] I_{2\times 2} \tag{3.46}$$

where $P \approx P_{inner} I_{2\times 2}$ is a TITO LTI system and $K = K_{inner} I_{2\times 2}$ is a diagonal inner-loop controller.

**Inner-Loop Open Loop Singular Values: $(\omega_r, \omega_l)$ System.** The open loop singular values for the $(\omega_r, \omega_l)$ system are plotted in Figure 3.18. Note that the different gains have been used. The blue line corresponds to a controller with a unitary gain crossover frequency $\omega_g$ of 1 $\frac{rad}{s}$, the black line corresponds to an $\omega_g = 2\frac{rad}{s}$ (with its gains obtained above) and the red line corresponds to an $\omega_g = 4\frac{rad}{s}$.

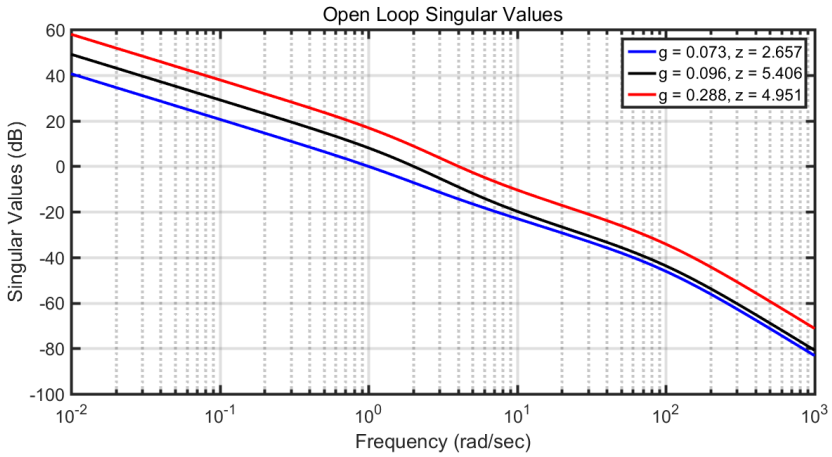


Figure 3.18: $L_o = PK$ Singular Values

Low frequency reference commands $r$ will be followed, low frequency output disturbances $d_o$ will be attenuated and high frequency sensor noise $n$ will be attenuated.

In [58] it was shown that the open loop singular values at the output/errors are the same as those at the controls/inputs. Also it was shown that the open loop singular values for for $PK$ and $MPKM^{-1}$ are identical.

**Sensitivity Singular Values.** The sensitivity singular values (at outputs/controls) for system $(\omega_r, \omega_l)$ are plotted in Figure 3.19.

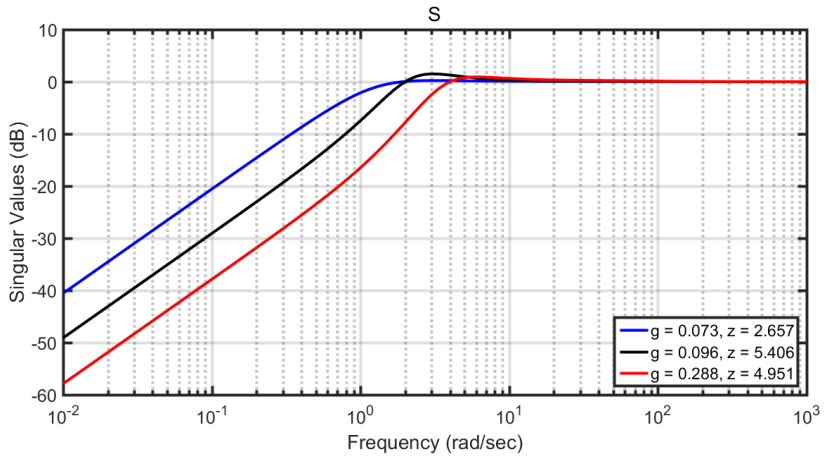


Figure 3.19: $S_o = (I + L_o)^{-1} = S_i$ Singular Values

Figure 3.19 shows that the system has good low frequency command following, good low frequency output disturbance attenuation and nominal stability robustness properties (i.e. little sensitivity peaking).

**Complementary Sensitivity Singular Values.** The complementary sensitivity singular values (at outputs/controls) for system $(\omega_r, \omega_l)$ are plotted in Figure 3.20.

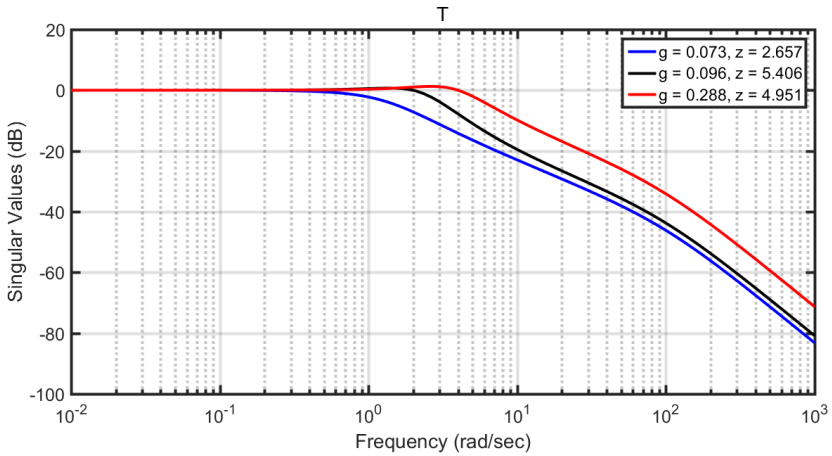


Figure 3.20: $T_o = I - So = T_i$ Singular Values

Figure 3.20 shows that low frequency reference commands will be followed. The plot also shows that high frequency sensor noise will be attenuated.

**Reference to Control Singular Values.** The reference to control singular values are shown in Figures 3.21-3.22. The latter shows the utility of the command pre-filter for reducing control effort.

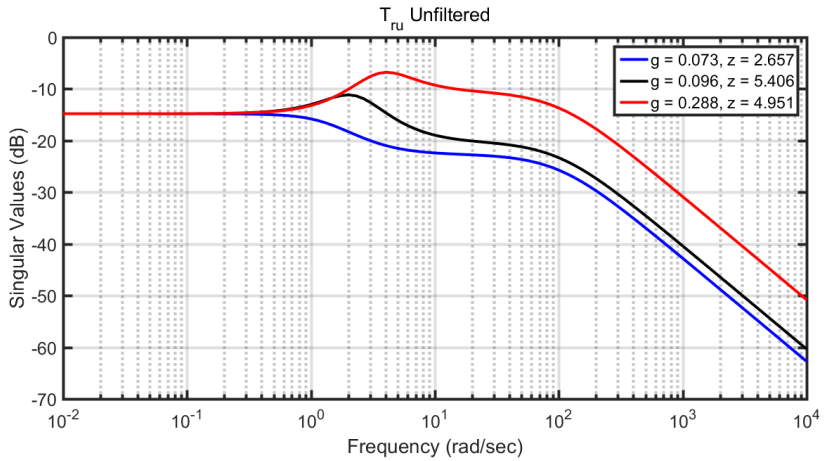


Figure 3.21: $T_{ru}$ Singular Values (No Pre-filter)

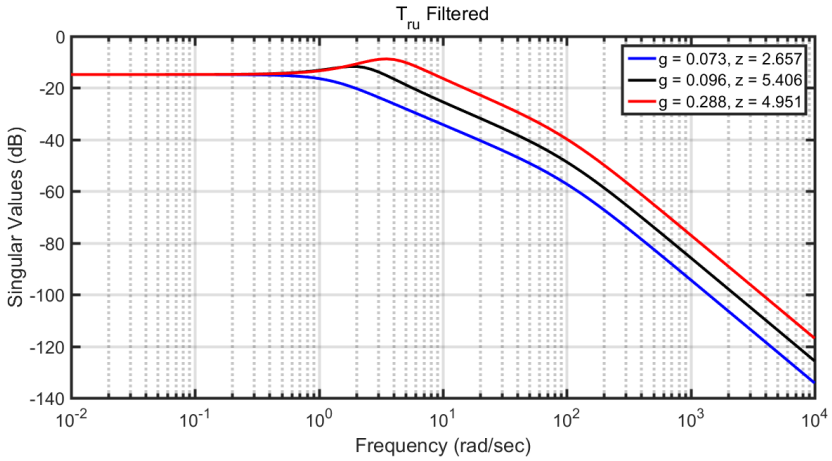


Figure 3.22: $WT_{ru}$ Singular Values (with Pre-filter)

Figures 3.21-3.22 suggest that reference commands $r$ will be attenuated to produce the necessary steady state control $u$.

In Figure 3.22 the peaks are reduced in comparison to the peaks Figure 3.21 since a pre-filter $W$ is used.

**Input Disturbance to Output $T_{d_iy}$ Singular Values.** The input disturbance to output singular values are shown in Figure 3.23.

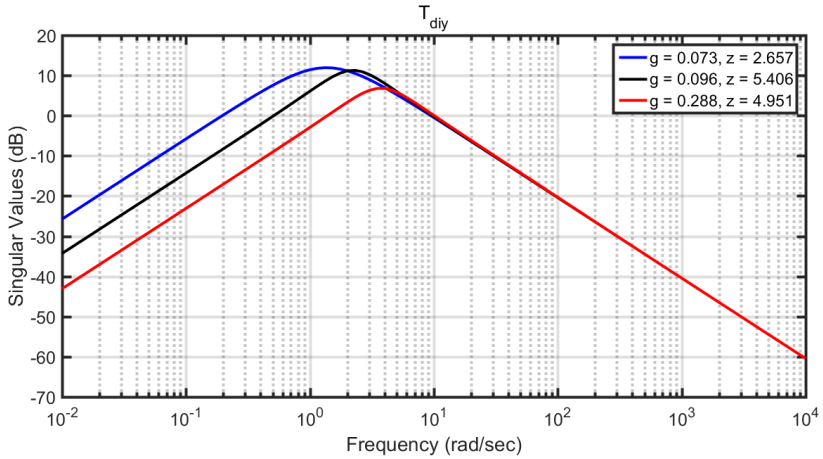


Figure 3.23: $T_{diy}$ Singular Values

The plot shows that as one increases the gains of the controller (i.e. higher bandwidth) the input disturbances will have little effect on the output.

For completeness in Figures 3.24-3.25 it is showed the singular values for $T_{diy}$ and $Tru$ (unfiltered) for the $(v, \omega)$ system.

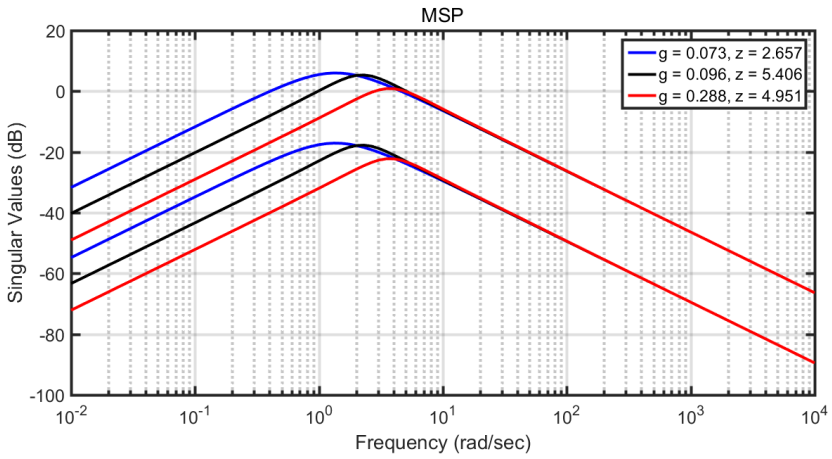


Figure 3.24: $MSP$ Singular Values

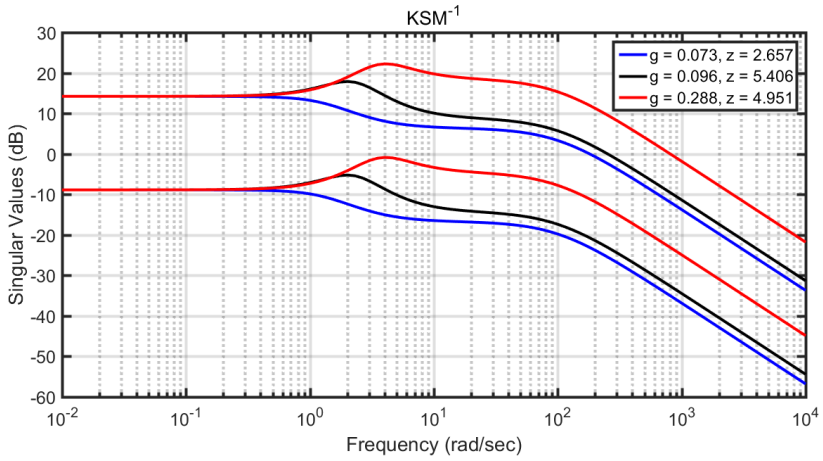


Figure 3.25: $KSM^{-1}$ Singular Values

From above, input disturbances for the $(v, \omega)$ have small impact on the output. Also, one needs to be careful when issuing commands for the $(v, \omega)$ system since the control action will be larger than the required for the corresponding $(\omega_R, \omega_L)$ system.

**Simulation and Experimental Step Response Analysis: Output Responses $(\omega_R, \omega_L)$.** The filtered step reference time responses for the $(\omega_R, \omega_L)$ inner loop control system are shown in Figure 3.26. The parameters for the controller are $g = 0.096$ and $z = 5.406$.

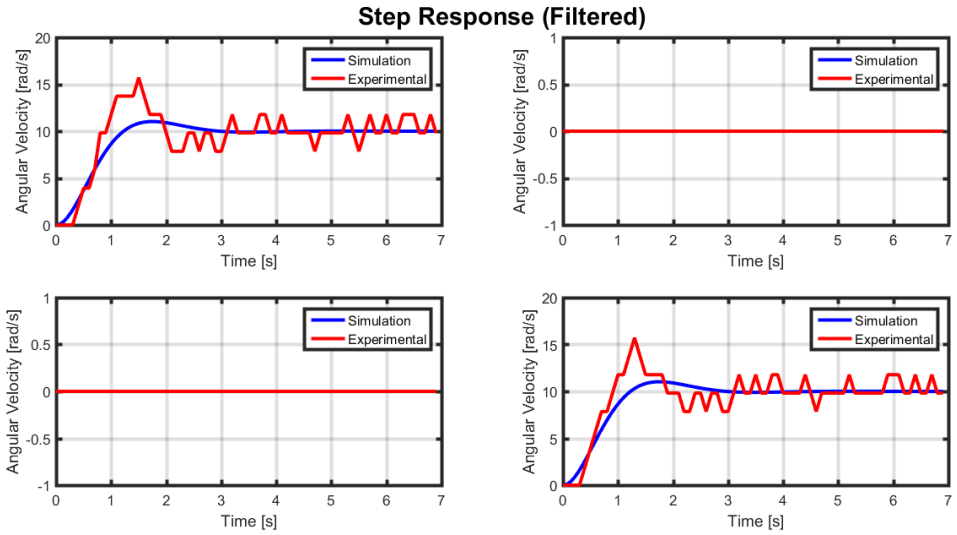


Figure 3.26: Inner-Loop [$\omega_R$, $\omega_L$] Filtered Step Response

The experimental result has overshoot due to deadzone and static friction. In the steady state both responses are close to each other. Note the encoder resolution is around 1.96 rad/s due to the fact that an average filter was used.

**Simulation and Experimental Step Response Analysis: Output Responses $(v, \omega)$.** The filtered step reference time responses for the $(v, \omega)$ system are shown in Figure 3.27. Here the encoder resolution for $v$ is around 0.098 m/s; the resolution for $\omega$ is approximately 0.7 rad/s.

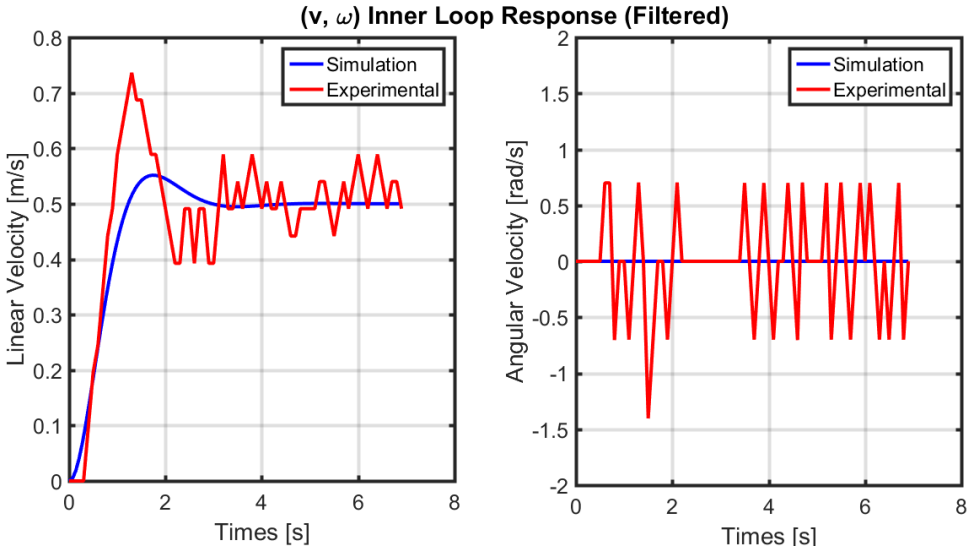


Figure 3.27: Inner-Loop $[v,\, \omega]$ Filtered Step Response

**Simulation and Experimental Step Response Analysis: Control Responses $(e_{aR}, e_{aL})$.** Next the filtered control responses for the inner loop (either for $(\omega_R, \omega_L)$ or $(v, \omega)$ systems) are presented in Figure 3.28.

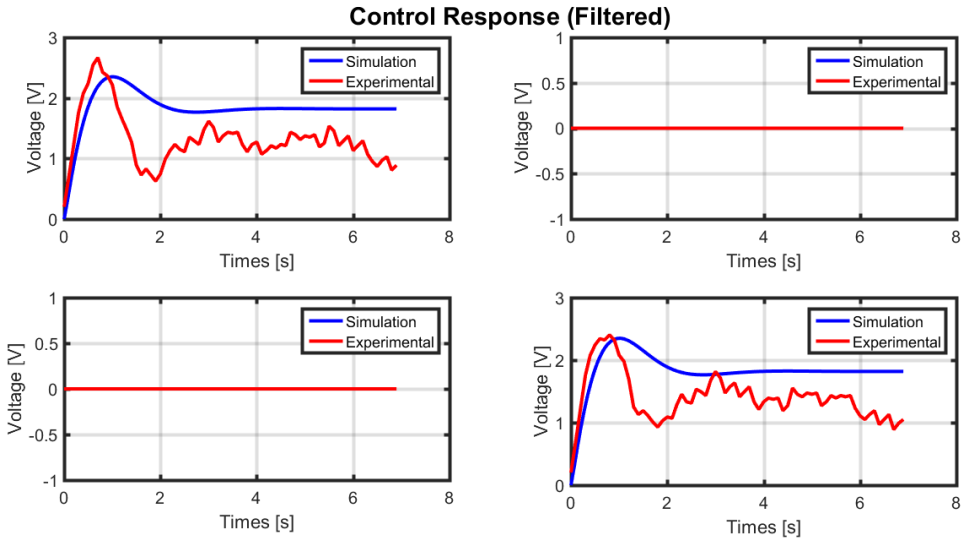


Figure 3.28: Control Step Response

**CODE: ARDUINO INNER-LOOP CONTROL LAW CODE.** The Arduino code used for implementing the $(\omega_r, \omega_l)$-$(v, \omega)$ inner-loop control law - a control law that is used by all of subsequent outer-loop control laws - can be found within Appendix B on page 185.

## 3.5 Summary and Conclusion

This chapter has provided a description of the hardware used within this thesis. The kinematics and dynamics for the differential-drive mobile robot were examined. Finally an inner loop speed control system was designed. This system is important since it is used in the subsequent chapters. Both simulation and hardware results were presented.

# Chapter 4

# VISION-BASED ROBOT CONTROL

## 4.1 Introduction and Overview

The aim in Vision-Based Robot Control or Visual Servoing is to use visual information to control the vehicles pose with respect to some landmarks [11]. By taking the image measurements of feature points/markers of an object, and comparing with the desired values of the features, the Visual Servoing control can then be designed [12]. The vision data may be acquired from a camera that is mounted directly on a robot manipulator or on a mobile robot, eye-in-hand, in which case motion of the robot induces camera motion, or the camera can be xed, eye-to-hand, in the workspace so that it can observe the robot motion from a stationary conguration. In this thesis an eye-in-hand configuration of the camera is used.

In this chapter, first the controllability properties of the differential-drive kinematic mobile robot model are examined. Then two (2) outer loop controllers are designed, namely Image Based Visual Servoing and Position Based Visual Servoing. Simulation as well as experimental results are presented and discussed.

## 4.2 Controllability of Nonlinear Kinematic Differential-Drive $(x, z, \theta)$ Model

In this section, the controllability properties of the $(x, z, \theta)$ differential-drive kinematic mobile robot model are examined - discussed within section 3.3.1. First the nonlinear model and then its linearization are examined. A system is said to be controllable if there exists a control law $u(\cdot)$ which can transfer the state of the system from any initial state $x_o$ to any final state $x_f$ within a finite amount of time.

**Controllability of Nonlinear Kinematic Differential-Drive $(x, z, \theta)$ Model.** The nonlinear kinematical model discussed within section 3.3.1 can be rewritten as follows:

$$\dot{x} = f(x) \; + \; g_1 u_1 \; + \; g_2 u_2 \tag{4.1}$$

where

$$f(x) = 0 \qquad g_1 = \begin{bmatrix} \sin\theta \\ \cos\theta \\ 0 \end{bmatrix} \qquad g_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{4.2}$$

The nonlinear (Lie-bracket based) controllability matrix for this system can be formed and its rank can be checked as follows [3]:

$$rank\,(\; g_1 \;\; g_2,\; [\; g_1,\; g_2\;]\;) = rank \begin{bmatrix} \sin\theta & 0 & -\cos\theta \\ \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \end{bmatrix} = 3 \tag{4.3}$$

Here, the quantity $[\; g_1, g_2\;]$ is called the Lie-Bracket of $g_1$ and $g_2$. It is defined by the following relationship:

$$[\; g_1, g_2\;] = \frac{\partial g_2}{\partial \theta} g_1 - \frac{\partial g_1}{\partial \theta} g_2 \tag{4.4}$$

Since the (nonlinear) controllability matrix $(\; g_1 \;\; g_2,\; [\; g_1,\; g_2\;]\;)$ has full rank, it follows that the system (i.e. nonlinear differential-drive kinematic vehicle model) is controllable. This confirms the common physical experience that a mobile vehicle can be taken from any point $(x_1, z_1, \theta_1)$ to any other point $(x_2, z_2, \theta_2)$. More specifically, it can be "parked" at any point $(x, z)$ in any posture $\theta$.

**Controllability of Linearized Kinematic Differential-Drive $(x, y, \theta)$ Model.** Linearizing the above nonlinear kinematic model about the equilibrium $(x, z, \theta) = (0, 0, 0)$ yields the following linear system

$$\begin{bmatrix} \dot{x} \\ \dot{z} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{4.5}$$

The controllability matrix for this LTI system is just the matrix given above. It has rank 2 which is less than the number of states or 3. Hence, this LTI system is uncontrollable. More precisely, since this system can be written as $\dot{x} = Ax + Bu$ with $A = 0_{3\times 3}$ and $B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$, it follows that the left eigenvector $[\ 1 \ \ 0 \ \ 0\ ]$ of $A$ lies in the left null space of $B$. By the PBH eigenvalue-eigenvector test [56] , the above LTI system is uncontrollable. Thus, in the process of linearizing the system controllability has been lost. This, fundamentally, is because the vehicle cannot move sideways!

**Brockett's Theorem.** Brocket's theorem is now presented. Brockett's theorem shows that no continuous control law can completely stabilize a system with a non-holonomic restriction.

**Theorem 4.1 (Brockett, 1983)**

*Suppose that (1) $\dot{q} = g(q)u$ is a continuously differentiable distribution in a neighborhood of $q_o$, (2) $g(q_o)u_o = 0$, (3) $g(q)$ is a distribution of constant rank in a neighborhood of $q_o$. Given the above, it follows that a continuously differentiable control law which makes $(q_o, u_o)$ asymptotically stable exists if and only if dim(q) = dim(u).*

In the case of non-holonomic mobile robots, $dim(q) = 3$ and $dim(u) = 2$. Thus no smooth control law exists which can stabilize the robot about a posture. This result requires that more sophisticated control schemes be used to stabilize non-holonomic mobile robots. These new schemes include time varying control laws, piece-wise continuous control or model transformation techniques. In short, to park a car the switching of control laws is needed. A single control law can get the robot close, but switching is required to achieve the target.

## 4.3 Image Formation

Image formation is the process where radiation emitted from objects is collected to form an image of the objects [18].

From images the size, shape and position of objects in the world can be deduced as well as other characteristics such as color and texture. In a digital camera a glass or plastic lens forms an image on the surface of a semiconductor chip with an array of light sensitive devices to convert light to a digital image.

Image formation, in an eye or in a camera, involves a projection of the 3-dimensional world onto a 2-dimensional surface. The depth information is lost and one can no longer tell from the image whether it is a large object in the distance or a smaller closer object. This transformation from 3 to 2 dimensions is known as perspective projection [6]. In computer vision it is common to use the central perspective imaging model shown in Figure 4.1.

The origin of the camera coordinate frame is at the center of projection of the camera (this is where the camera aperture is located). The z-axis is taken to be the optical axis of the camera (which points in front of the camera in the positive z direction).

The rays converge on the origin of the camera frame $C$ and a non-inverted image

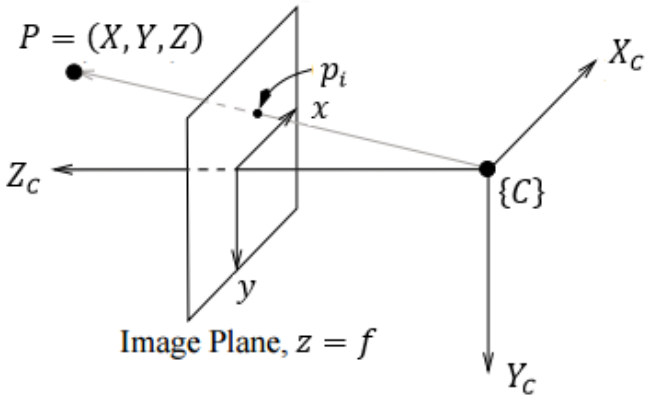


Figure 4.1: Perspective Model

is projected onto the image plane located at a distance $z = f$. Using similar triangles a point at the world coordinates $P = (X, Y, Z)$ is projected to the image plane $p_i = (x, y)$ by

$$x = f\frac{X}{Z} \quad y = f\frac{Y}{Z} \tag{4.6}$$

or in compact matrix form

$$p_i^{'} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{4.7}$$

where $p_i^{'} = (x^{'}, y^{'}, z^{'})$ and the image plane coordinates are obtained as follows, $x = x^{'}/z^{'}$ and $y = y^{'}/z^{'}$.

A perspective projection, from the world to the image plane and has the following characteristics:

- It performs a mapping from 3-dimensional space to the 2-dimensional image plane,

- Straight lines in the world are projected to straight lines on the image plane
- Parallel lines in the world are projected to lines that intersect at a vanishing point
- Conics in the world are projected to conics on the image plane. For example, a circle is projected as a circle or an ellipse
- The mapping is not one-to-one and a unique inverse does not exist. That is, given $(x, y)$ uniquely determining $(X, Y, Z)$ is not possible
- The transformation is not conformal, i.e. it does not preserve shape since internal angles are not preserved.

In a digital camera the image plane is a W $\times$ H grid of light sensitive elements called photosites that correspond directly to the picture elements (or pixels) of the image as shown in Figure 4.2.

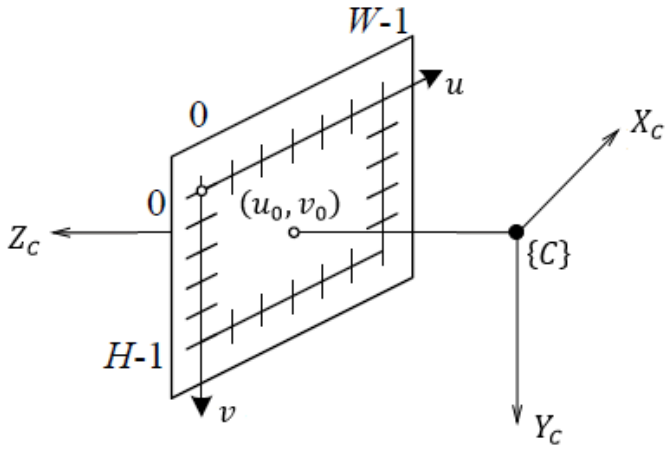


Figure 4.2: Pixels In A Digital Camera

The pixel coordinates are a 2-vector $(u, v)$ of non-negative integers and by convention the origin is at the top-left hand corner of the image plane. The pixels are

uniform in size and centered on a regular grid so the pixel coordinate is $(x, y)$ related to the image plane coordinate $(u, v)$ by the following expression

$$u = \frac{x}{\rho_w} + u_0 \quad v = \frac{y}{\rho_h} + v_0 \tag{4.8}$$

where $\rho_w$ and $\rho_h$ are the width and height of each pixel respectively, and $(u_0, v_0)$ is the principal point, i.e. the coordinate of the point where the optical axis intersects the image plane [6]. Equivalently one can express the previous equations in matrix form as follows:

$$p^{'} = \begin{bmatrix} u^{'} \\ v^{'} \\ w^{'} \end{bmatrix} = \begin{bmatrix} \frac{f}{\rho_w} & 0 & u_0 \\ 0 & \frac{f}{\rho_h} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = CP \tag{4.9}$$

From this the image plane coordinates expressed in pixels are found like this:

$$p = \begin{bmatrix} u \\ \\ v \end{bmatrix} = \begin{bmatrix} \frac{u^{'}}{w^{'}} \\ \\ \frac{v^{'}}{w^{'}} \end{bmatrix} \tag{4.10}$$

The matrix $C$ found in (4.9) is called camera intrinsic parameters matrix, i.e. its elements are innate characteristics of the camera and sensor.

### *4.3.1 Camera Calibration*

Camera calibration is the process of determining the cameras intrinsic parameters. Calibration techniques rely on sets of world points whose relative coordinates are known and whose corresponding image-plane coordinates are also known.

Twenty pictures of the chessboard from different angles were taken with the Raspberry Pi Camera and used in the OpenCV default calibration function. The size of each picture is 320x240. Figure 4.3 shows some examples of the pictures taken.

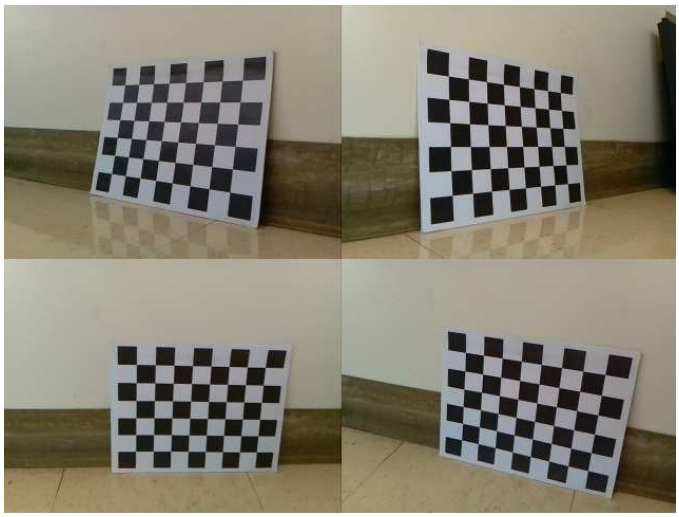

Figure 4.3: Pictures Used for Camera Calibration

The camera intrinsic parameters matrix $C$ were found to be:

$$C = \begin{bmatrix} 327.267 & 0 & 152.44 \\ 0 & 326.883 & 120.221 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.11}$$

## 4.4 Position Based Visual Servoing

Position Based Visual Servoing (PBVS) uses observed visual features, a calibrated camera and a known geometric model of the target to determine (estimate) the position of the target with respect to the camera. The robot then computes the error between desired and actual pose to generate the required control input to get to its destination. Good algorithms exist for pose estimation but it is computationally expensive and relies critically on the accuracy of the camera calibration and the model of the objects geometry [6].

### *4.4.1 Control Law Development*

PBVS operates on the task space, i.e. $X - Z$ plane, therefore the goal is to minimize the errors in position $(x_{ref} - x)$ and $(z_{ref} - z)$. Hence the outer loop design presented here is the same as the cartesian stabilization showed in [58]. The difference is the way the mobile robot position is estimated. In [58] IMU along with wheel encoders were used to get an estimate position; here only a camera is used.

The block diagram shown in Figure 4.4 shows the outer loop implementation of PBVS.

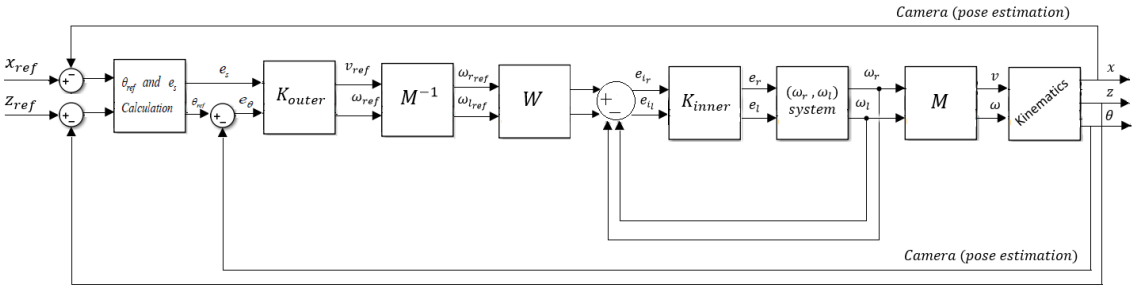


Figure 4.4: Position Based Visual Servoing Block Diagram

The use of a proportional gain controller is justified from the work in [4]. A simple control law $v = k_s e_s$, $\omega = k_\theta e_\theta$ results in an error dynamics matrix (after linearization) that is Hurwitz when $k_\theta > k_s > 0$ [4]. A drawback of this control law (consistent with the Brockett 1983 result [1]) is that it can only get the system arbitrarily close to the desired $(x_{ref}, z_{ref}, \theta_{ref})$ [4]. To precisely achieve the objective, one would have to switch control laws. These ideas are used to motivate a simple proportional control law for PBVS outer-loop that was implemented for the differential-drive vehicle. It is now useful to present some of the key ideas about cartesian stabilization within [4]. Let $e_s = \Delta\lambda$ denote the projection of the vehicle-to-target vector onto the longitudinal body axis of the vehicle. $\phi$ is defined as the angle which binds $(x_{ref}, z_{ref})$ and $(x, z)$. It is called the *pointing angle*.

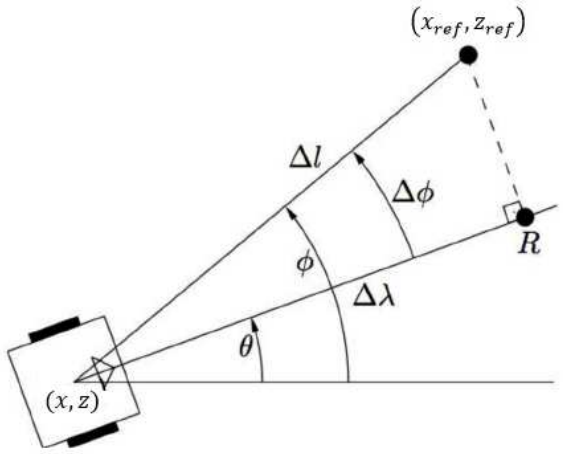


Figure 4.5: Visualization of Longitudinal Distance to Target $e_s = \Delta\lambda$ and Angular Error $e_\theta = \Delta\phi$

From Figure 4.5, the following expressions are obtained:

$$\phi = tan^{-1}(\frac{z_{ref} - z}{x_{ref} - x}) \tag{4.12}$$

$$e_\theta = \phi - \theta \tag{4.13}$$

$$e_s = \Delta\lambda = \Delta l \cos \Delta\phi \tag{4.14}$$

The structure of the control law used within [4] and which will be used here as well is as follows - a proportional control law:

$$v = k_s e_s \quad \omega = k_\theta e_\theta \tag{4.15}$$

In [58] the local stability of this closed loop system is proved by analizing the error dynamics. The *Cartesian stabilization* error dynamics and hence the *PBVS*, will be locally exponentially stable if $k_\theta > k_s > 0$.

How does the robot estimate its position with respect to some target using only visual information, i.e. with the camera?

First the target in this thesis is a chessboard as shown in the Figure 4.6.

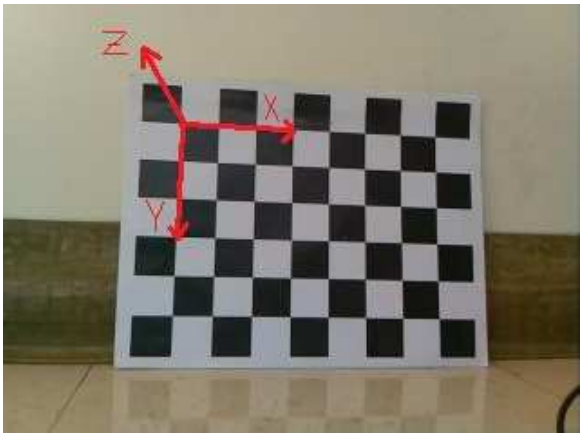


Figure 4.6: $7 \times 6$ Chessboard

The goal here is to determine the pose of the target coordinate frame, $\{T\}$ with respect to the camera. The geometry of the target is known, that is, the position of the points are known - in this case the corners - $(X_i, Y_i, Z_i)$, $i \in [1, N]$ on the target with respect to $\{T\}$. The distance between each corner in the chessboard is $2.8cm$. The intrinsic parameters of the camera are also known, as it was discussed in

4.3.1. An image is captured and the *corresponding* image plane coordinates $(u_i, v_i)$ are determined by using an OpenCV function that computes the corners of the squares in the chessboard.

As mentioned in Chapter 1, estimating the pose using $(u_i, v_i)$, $(X_i, Y_i, Z_i)$ and the intrinsic parameters of the camera is known as the Perspective-n-Point problem or PnP for short [6]. OpenCV provides a function which is called *SolvePnP* that takes as inputs the camera intrinsic parameters, the 2D image points $(u_i, v_i)$ and the corresponding 3D coordinates $(X_i, Y_i, Z_i)$ of the points and it returns the pose (rotation matrix, translation vector) of the target with respect to the camera.

### *4.4.2 Simulation Results*

In this part simulation results for the Position Based Visual Servoing are presented. The gains $k_\theta = 0.8$ and $k_s = 0.4$ were used. In Figure 4.7 the robot reaches the desired position $(x_{ref}, z_{ref}) = (0, 1)$ for three different initial conditions. The robot fails to get close to the desired position when $|x_{ref}| > 0.2$m.

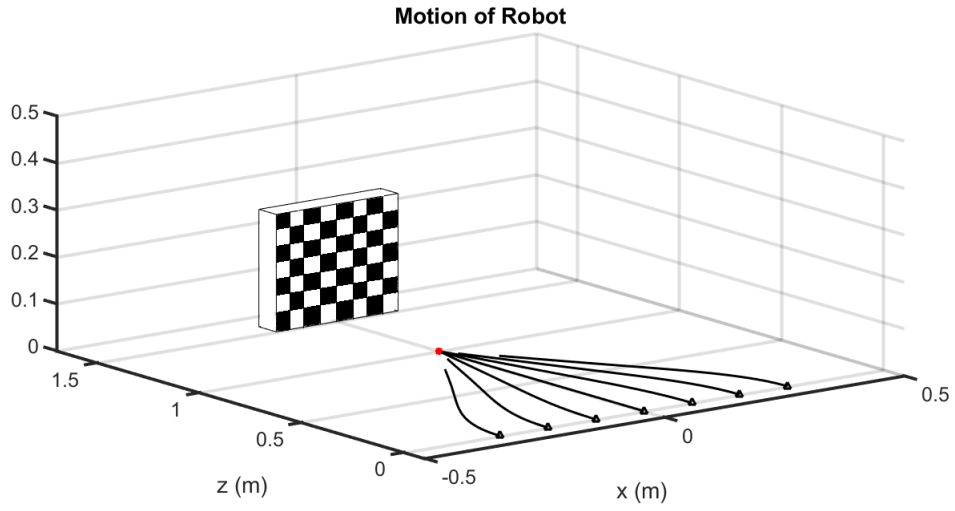


Figure 4.7: Motion of Robot Using Position Based Visual Servoing

This occurs because the chessboard leaves the camera Field of View (FOV), and once this happens the robot can't continue with its motion. A pan camera is then implemented, i.e. one that can be moved from side to side. This way the camera can keep the chessboard in its FOV. A simple algorithm for the control of the angle of the pan camera was used; when the middle point of the chessboard started to move to the left or right from the center of the image plane then the pan camera would turn either to the to the left or right respectively. Proportional and integral gains were used ($k_p = 0.001, k_i = 0.003$).

In Figure 4.8 the robot reaches the desired position $(x_{ref}, z_{ref}) = (0, 1)$ for all initial positions.

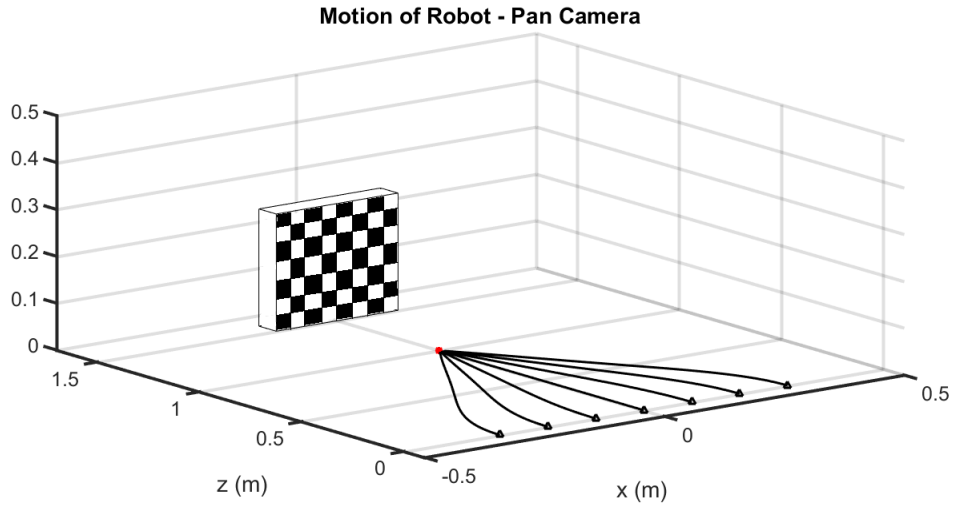


Figure 4.8: Motion of Robot Using Position Based Visual Servoing with Pan Camera

### *4.4.3 Experimental Results*

The experimental results for the PBVS outer loop control system are now presented. Figure 4.9 shows how the mobile robot moves in the $x-z$ plane from different initial conditions. In accordance with simulation results presented above, the robot cannot get to the desired position due to the chessboard leaving the FOV of the cam-

era. During this test it was observed that sometimes the algorithm failed to detect the chessboard and consequently failed to estimate position and the motion of the mobile robot was not smooth. A more robust algorithm and a different target can be later investigated to improve performance.

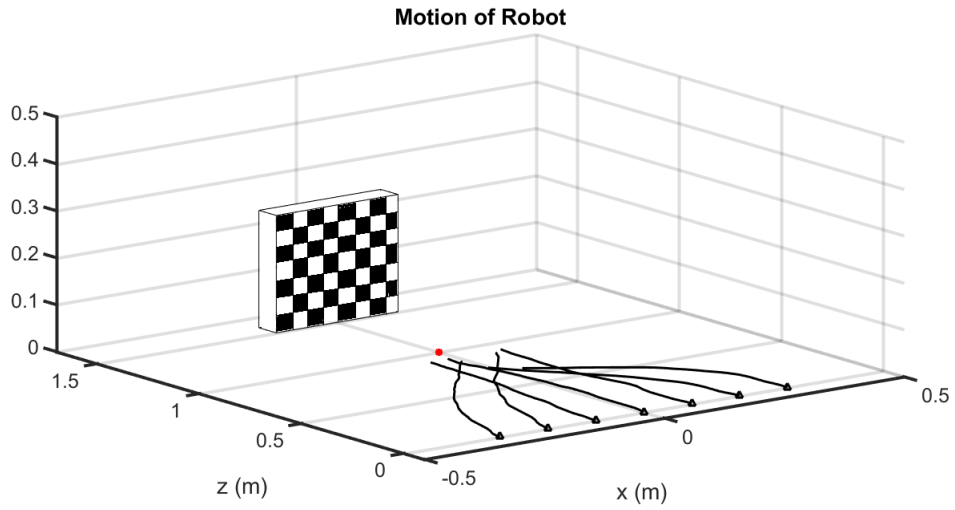


Figure 4.9: Motion of Robot Using PBVS - Experimental

Once a pan camera is used the robot gets close to the desired position as it can be seen in Figure 4.10.

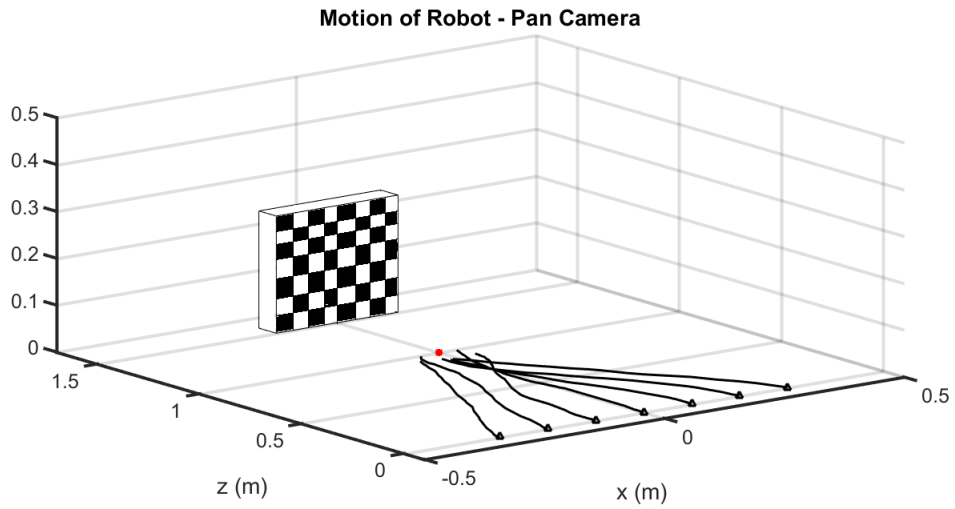


Figure 4.10: Motion of Robot Using PBVS with Pan Camera - Experimental

The accuracy of the estimation algorithm is around 75% along the x-axis (lateral accuracy). An accuracy of about 98% was observed on the z-axis (longitudinal accuracy). A different estimation algorithm can be investigated in order to improve this accuracy.

**CODE: PYTHON AND ARDUINO CODE.** The Python and Arduino code used for the Position-Based robot control can be found within Appendix C on page 213 and Appendix B on page 187.

## 4.5 Image-Based Visual Servoing

Image-based visual servoing (IBVS) uses the image features directly. The control is performed in image coordinate space, in other words image-based schemes define the reference signal in the image plane [10]. The desired camera pose is defined implicitly by the image feature values at the goal pose [6].

In IBVS control, an error signal is measured in the image and mapped directly to actuator commands [14]. A controller is designed to maneuver the image features toward a goal configuration. The approach is inherently robust to camera calibration and target modeling errors. Because of the above reasons IBVS has seen increasing popularity in recent years [13].

### *4.5.1 Control Law Development*

The aim of the IBVS scheme is to minimize an error $\mathbf{e}(t)$, which is defined by

$$\mathbf{e}(t) = \mathbf{p}^* - \mathbf{p} \tag{4.16}$$

where $\mathbf{p}^*, \mathbf{p}^* \in \mathbb{R}^{2k}$ are vectors that contain desired and current visual features. In this thesis image plane coordinates $(u, v)$ of $k$ points or dots are used as visual features. The target object is assumed to have these $k$ points.

Before going further, it is appropriate to define important characteristics about the wheeled mobile robot with the camera. In Figure 4.11 the camera coordinate frame is shown as well as the image plane and its origin, which is at the top right corner.

Figure 4.12 the relationships between different coordinate frames. Here $\theta$ is the angle between the world and robot frames Z-axis, $\phi_c$ is the angle formed between the robot and camera frames Z-axis (measured positive in the clockwise direction).

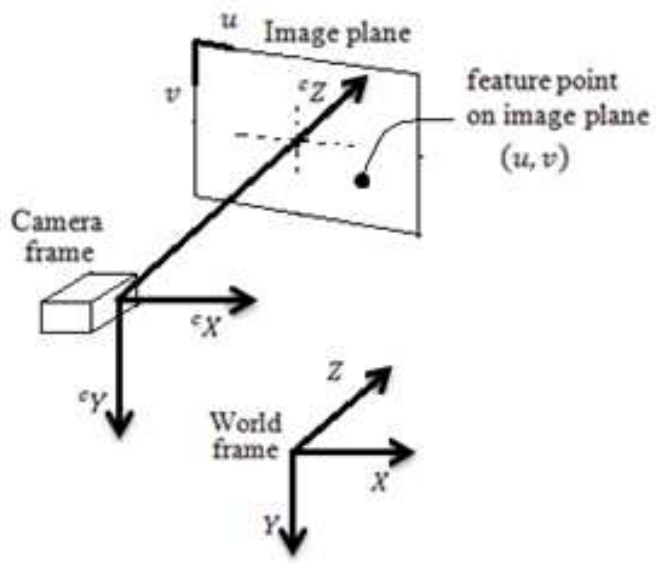


Figure 4.11: Camera Coordinate Frame and Image Plane

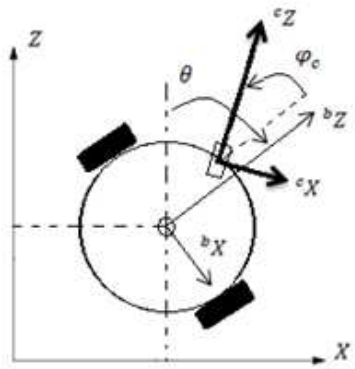


Figure 4.12: Complete System

Consider a camera moving with a body velocity $\mathbf{v_{cam}} = (\mathbf{v_c}, \omega_\mathbf{c}) = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$ in the world frame and observing a world point $P$ with camera relative coordinates $P = (X, Y, Z)$. The velocity of the point relative to the camera frame is given by

$$\dot{\mathbf{P}} = -\omega_\mathbf{c} \times P - \mathbf{v_c} \tag{4.17}$$

which can be written in scalar form as

$$\begin{aligned}\dot{X} &= Y\omega_z Z - \omega_y - v_x \\ \dot{Y} &= Z\omega_x - X\omega_z - v_y \\ \dot{Z} &= X\omega_y - Y\omega_y - v_z\end{aligned} \tag{4.18}$$

The perspective projection for normalized coordinates is given by

$$x = \frac{X}{Z} \quad y = \frac{Y}{Z} \tag{4.19}$$

and the derivative, using the quotient rule is

$$\dot{x} = \frac{\dot{X}Z - X\dot{Z}}{Z^2} \quad y = \frac{\dot{Y}Z - Y\dot{Z}}{Z^2} \tag{4.20}$$

Substituting $X = xZ$, $Y = yZ$ and (4.18), one can rewrite equations (4.20) in matrix form

$$\begin{bmatrix}\dot{x} \\ \dot{y}\end{bmatrix} = \begin{bmatrix}-\frac{1}{Z} & 0 & \frac{x}{Z} & xy & -(1+x^2) & y \\ 0 & -\frac{1}{Z} & \frac{y}{Z} & (1+y^2) & -xy & -x\end{bmatrix} \begin{bmatrix}v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z\end{bmatrix} \tag{4.21}$$

which relates the camera velocity to the velocity of the normalized image coordinates. The normalized image plane coordinates are related to the pixel coordinates by

$$u = \frac{f}{\rho_w}x + u_0 \quad v = \frac{f}{\rho_h}y + v_0 \tag{4.22}$$

which may be rearranged as

$$x = \frac{\rho_w}{f}\bar{u} \quad y = \frac{\rho_h}{f}\bar{v} \tag{4.23}$$

where $\bar{u} = u - u_0$ and $\bar{v} = v - v_0$ are the pixel coordinates relative to the principal point. Taking the derivative one obtains the following

$$\dot{x} = \frac{\rho_w}{f}\dot{\bar{u}} \quad \dot{y} = \frac{\rho_h}{f}\dot{\bar{v}} \tag{4.24}$$

Finally substituting (4.23), (4.24) in (4.21) leads to

$$\begin{bmatrix} \dot{\bar{u}} \\ \dot{\bar{v}} \end{bmatrix} = \begin{bmatrix} -\frac{f}{\rho_w Z} & 0 & \frac{\bar{u}}{Z} & \frac{\rho_w \bar{u}\bar{v}}{f} & -\frac{f^2+\rho_w^2\bar{u}^2}{\rho_w f} & \bar{v} \\ 0 & -\frac{f}{\rho_h Z} & \frac{\bar{v}}{Z} & \frac{f^2+\rho_h^2\bar{v}^2}{\rho_h f} & -\frac{\rho_h \bar{u}\bar{v}}{f} & -\bar{u} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{4.25}$$

One can write this in concise matrix form as

$$\dot{p} = J_{img}\mathbf{v_{cam}} \tag{4.26}$$

where $J_{img}$ is called the image jacobian matrix for a point feature or a dot. In the previous equation only one marker was being considered, however if $k$ markers are used then $p = [\bar{u_1}, \bar{v_1}, ..., \bar{u_k}, \bar{v_k}]^T \in \mathbb{R}^{2k}$, and $J_{img} \in \mathbb{R}^{2k\times 6}$. The image jacobian matrix does not depend at all on the world coordinates $X$ or $Y$, only on the image plane coordinates $(u, v)$ and the depth $Z$. In [15] an approximation to the image jacobian matrix is described; in this thesis $J_{img}$ is constant, i.e. the desired pixel coordinates $(u_{id}, v_{id})$ of the markers and a constant value for the depth $Z$ (0.4) will be used.

There is a relationship between the camera velocity $\mathbf{v_{cam}}$ and the linear and angular velocity of the robot, $\mathbf{v} = [v, \omega]^T$ [9], which is given by the next equation

$$\mathbf{v_{cam}} = \begin{bmatrix} -\sin\phi_c & b_{X_c}\sin\phi_c + b_{Z_c}\cos\phi_c \\ 0 & 0 \\ \cos\phi_c & -b_{X_c}\cos\phi_c + b_{Z_c}\sin\phi_c \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} = J_R\mathbf{v} \tag{4.27}$$

where $b_{X_c}$ is the distance from the robot to the camera along the robot's $X$ axis, $b_{Z_c}$ is the distance from the robot to the camera along the robot's $Z$ axis.

Having a fixed camera on the wheeled mobile robot means that $\phi_c$ is constant, specifically in this work it will be equal to zero degress, i.e. the $Z$ axis of the camera is coincident with the $Z$ axis of the robot. The parameters $b_{X_c}$ and $b_{Z_c}$ are equal to 0 and 10 cm, respectively. Given this equation (4.27) becomes

$$\mathbf{v_{cam}} = \begin{bmatrix} 0 & b_{Z_c} \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{4.28}$$

Using (4.28) along with (4.26), the following is obtained

$$\dot{p} = J_{img} J_R \begin{bmatrix} v \\ \\ \omega \end{bmatrix} = J_{vis} \begin{bmatrix} v \\ \\ \omega \end{bmatrix} \tag{4.29}$$

where $J_{vis} \in \mathbb{R}^{2k \times 2}$. Solving for the linear and angular velocities of the robot, the next equation is obtained

$$\begin{bmatrix} v \\ \\ \omega \end{bmatrix} = J_{vis}^{+} \dot{p} \tag{4.30}$$

The matrix $J_{vis}$ is the pseudo-inverse of $J_{vis}$ and is described as

$$J_{vis}^{+} = \begin{cases} J_{vis}^{-1} & : k = 1 \\ \\ (J_{vis}^{T} J_{vis})^{-1} J_{vis}^{T} & : k > 1 \end{cases} \tag{4.31}$$

The control objective is to drive each feature point $p_i$ to the desired one $p_i^*$ $(i = 1, , k)$. To do this, a proportional controller is used

$$\dot{p} = \lambda (p^* - p) \tag{4.32}$$

Combining equation (4.32) with equation (4.30), the control law is obtained

$$\begin{bmatrix} v_{ref} \\ \\ \omega_{ref} \end{bmatrix} = \lambda J_{vis}^{+} (p^* - p) \tag{4.33}$$

The way $p^*$ is obtained in this thesis is by taking the robot to the desired position and then taking a picture of the target object in which the markers are located (learning phase).

Figure 4.13 shows a block diagram of the outer loop implementation of IBVS.

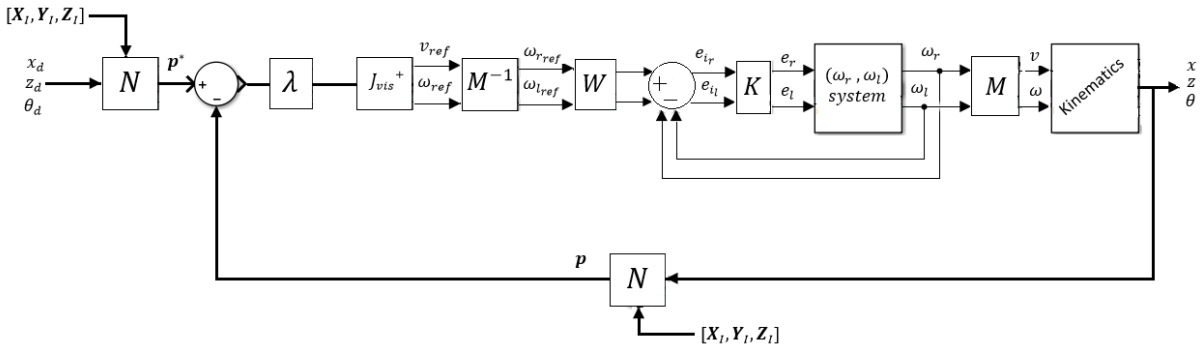


Figure 4.13: Image Based Visual Servoing Block Diagram

Here $N$ is a nonlinear transformation that takes the position of the robot $(x, z, \theta)$ and of the dots $(X_I, Y_I, Z_I)$ in the world/inertial frame and produces the pixel coordinates $(u, v)$ in the image plane of each of those dots. This mapping consists of the following:

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = C\, T_{cb}^{-1}\, T_{bI}^{-1} \begin{bmatrix} X_I \\ Y_I \\ Z_I \end{bmatrix} \tag{4.34}$$

$$u = \frac{u'}{w'} \quad v = \frac{v'}{w'}$$

where the $C$ matrix is the camera intrinsic parameters matrix, $T_{cb}$ and $T_{bI}$ are transformation matrices (from camera coordinate frame to robot coordinate frame and from robot coordinate frame to world/inertial frame) and are defined as follows:

$$C = \begin{bmatrix} \frac{f}{\rho_w} & 0 & u_0 & 0 \\ 0 & \frac{f}{\rho_h} & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad T_{cb} = \begin{bmatrix} \cos\phi_c & 0 & \sin\phi_c & b_{X_c} \\ 0 & 1 & 0 & b_{Y_c} \\ -\sin\phi_c & 0 & \cos\phi_c & b_{Z_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.35}$$

where $b_{X_c}$, $b_{Y_c}$, $b_{Z_c}$ are the distances from the mobile robot to the camera along each robot axis $b_X, b_Y, b_Z$.

$$T_{bI} = \begin{bmatrix} \cos\theta_d & 0 & \sin\theta_d & x_d \\ 0 & 1 & 0 & y_d \\ -\sin\theta_d & 0 & \cos\theta_d & z_d \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.36}$$

**Image processing.** In this thesis some image processing techniques were used for implementing IBVS. Image processing is a computational process that transforms one or more input images into an output image. Image processing is frequently used to enhance an image for human viewing or interpretation, for example to improve contrast [6].

- Image Segmentation. The term image segmentation refers to the partition of an image into a set of regions. The goal in many tasks is for the regions to represent meaningful areas of the image, such as the crops, urban areas, and forests of a satellite image. A binary image can be obtained from a color image through an operation that selects a subset of the image pixels as foreground pixels, the pixels of interest in an image analysis task, leaving the rest as background pixels to be ignored. The selection operation can be as simple as the thresholding operator that chooses pixels in a certain subspace of color space or it may be a complex

classification algorithm. In a number of applications binary images can be used as the input to algorithms that perform useful tasks. These algorithms can handle tasks ranging from very simple counting tasks to much more complex recognition, localization, and inspection tasks [20]. In this work a simple color thresholding was used to identify/detect the colored markers (which are on the target object) on the image.

- Morphology. The word morphology refers to form and structure; in computer vision it can be used to refer to the shape of a region. The most common binary image operations are called morphological operations, since they change the shape of the underlying binary objects [22]. The operations of binary morphology input a binary image $B$ and a structuring element $S$, which is another, usually much smaller, binary image. The structuring element represents a shape; it can be of any size and have arbitrary structure. However, there are a number of common structuring elements such as rectangle of specified dimensions, or a circular region of specified diameter. The purpose of the structuring elements is to act as probes of the binary image. One pixel of the structuring element is denoted as its origin; this is often the central pixel of a symmetric structuring element. Some of the basic morphology operations, which are used in this thesis, are dilation and erosion. A dilation operation enlarges a region, while erosion makes it smaller. These operations arise in a wide variety of contexts such as removing noise, isolating individual elements, and joining disparate elements in an image. Dilation is a convolution of some image $B$, with some kernel, or structuring element $S$. As the kernel $S$ is scanned over the image, the maximal pixel value overlapped by $S$ is computed and the value of the image pixel under the origin is replaced with that maximal value. This causes

bright regions within an image to grow. Erosion is the converse operation. The action of the erosion operator is equivalent to computing a local minimum over the area of the kernel. As the kernel $S$ is scanned over the image, the minimal pixel value overlapped by $S$ is computed and the value of the image pixel under the origin is replaced with that minimal value [17]. Some examples of the dilation and erosion operations are shown in Figure 4.14.

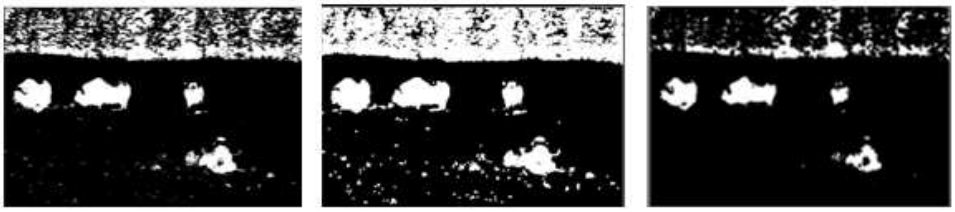

Figure 4.14: From Left to Right. Original Image, Dilated Image, Eroded Image

### *4.5.2 Simulation Results*

Simulation results are presented for the Image Based Visual Servoing. Figure 4.15 shows the motion of the robot using one marker on the target.

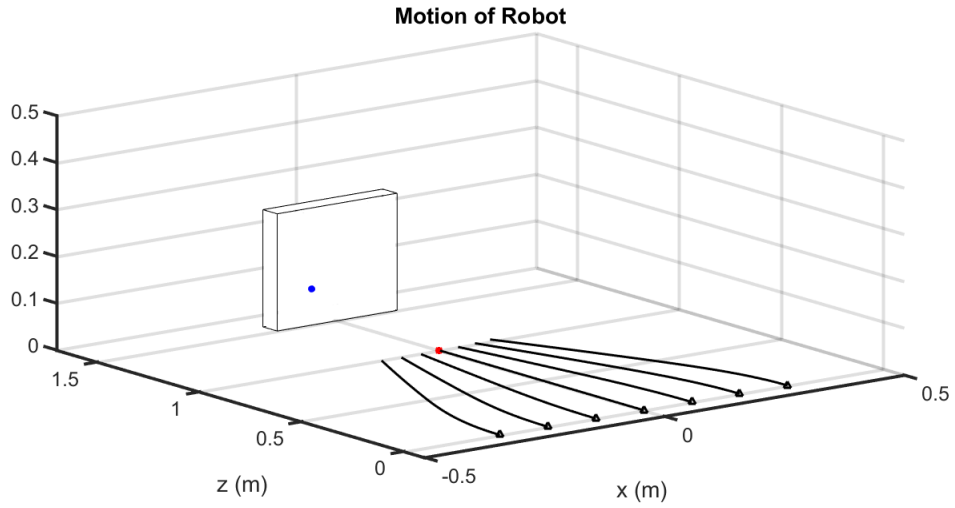


Figure 4.15: Motion of Robot Using One Marker

The initial position is varied from $x_0 = -0.3$ to $x_0 = 0.3$ in increments of 0.1 m, $z_0 = 0$ and $\theta_0 = 0$. The box with the blue marker represents the target the robot sees in order to get to the desired position. The robot reaches the desired position $(x_{ref}, z_{ref}, \theta_{ref}) = (0, 1, 0)$ only when $(x_0, z_0, \theta_0) = (0, 0, 0)$. This is due to local minima, i.e. the robot sees the marker from different positions the same way it sees it from the desired position.

Figures 4.16, 4.17, 4.18, 4.19, 4.20, 4.21, 4.22 show the trajectory followed by the marker on the image plane for each initial condition.

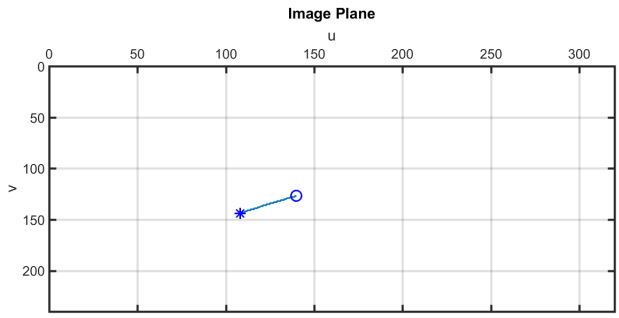


Figure 4.16: Trajectory of Marker on the Image Plane with $(x_0, z_0, \theta_0) = (0, 0, 0)$

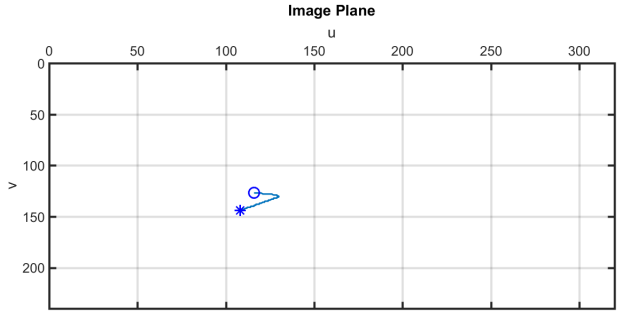


Figure 4.17: Trajectory of Marker on the Image Plane with $(x_0, z_0, \theta_0) = (0.1, 0, 0)$

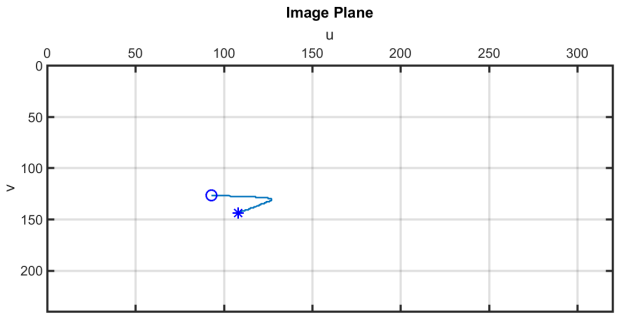


Figure 4.18: Trajectory of Marker on the Image Plane with $(x_0, z_0, \theta_0) = (0.2, 0, 0)$

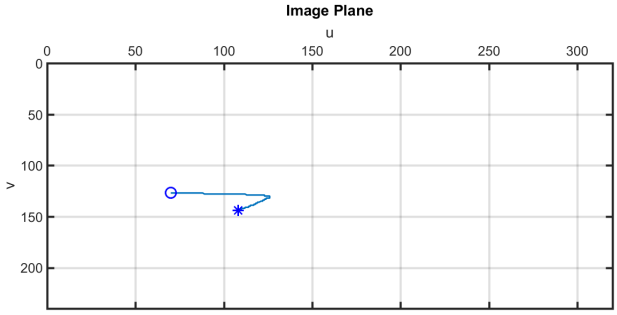


Figure 4.19: Trajectory of Marker on the Image Plane with $(x_0, z_0, \theta_0) = (0.3, 0, 0)$

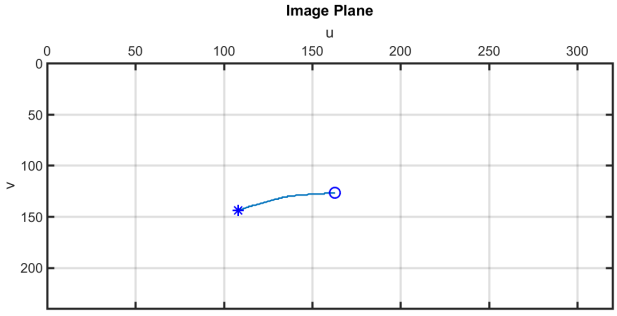


Figure 4.20: Trajectory of Marker on the Image Plane with $(x_0, z_0, \theta_0) = (-0.1, 0, 0)$

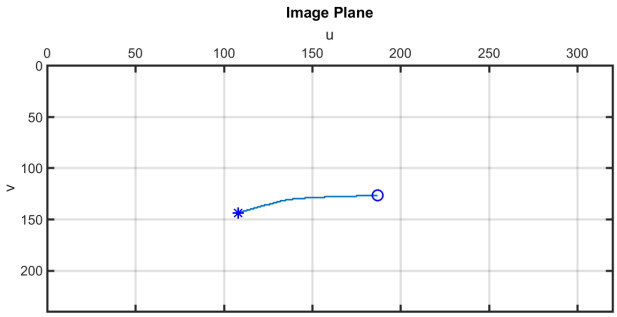


Figure 4.21: Trajectory of Marker on the Image Plane with $(x_0, z_0, \theta_0) = (-0.2, 0, 0)$

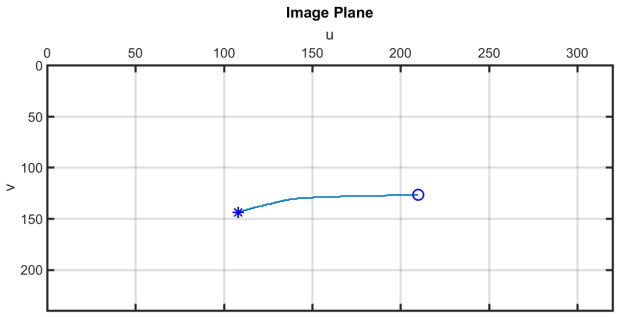


Figure 4.22: Trajectory of Marker on the Image Plane with $(x_0, z_0, \theta_0) = (-0.3, 0, 0)$

The '$o$' symbol represents the initial position of the marker in the image plane. The '$*$' symbol represents the position of the marker in the image plane when the robot is at the desired position $(x_{ref}, z_{ref}, \theta_{ref}) = (0, 1, 0)$. Note that in the image plane the marker gets to the desired pixel coordinate for any of the initial conditions.

Figure 4.23 shows the motion of the robot using two markers on the target.

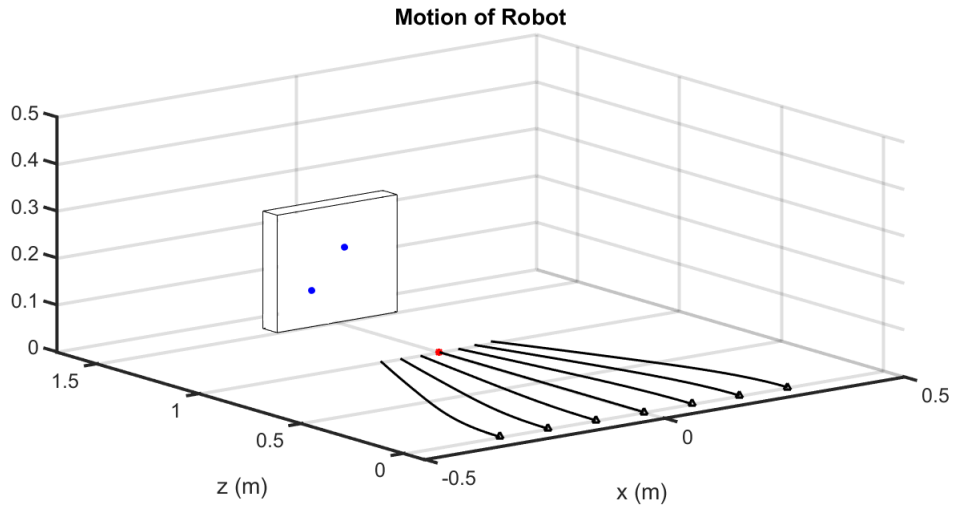


Figure 4.23: Motion of Robot Using Two Markers

Here, as with the case of one marker, the robot reaches desired position only when $(x_0, z_0, \theta_0) = (0, 0, 0)$.

Figures 4.24, 4.25, 4.26, 4.27, 4.28, 4.29, 4.30 show the trajectory followed by the markers on the image plane for each initial condition.

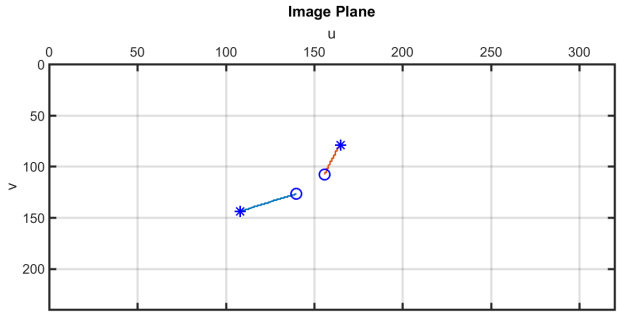


Figure 4.24: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0, 0, 0)$

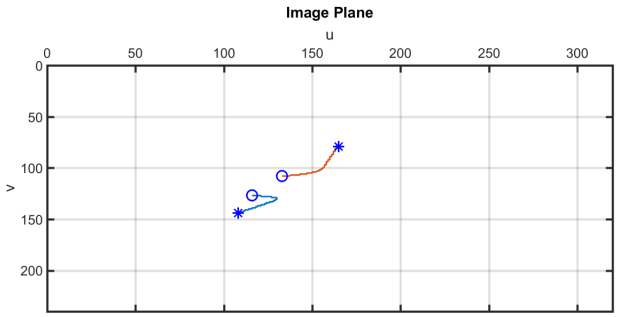


Figure 4.25: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.1, 0, 0)$

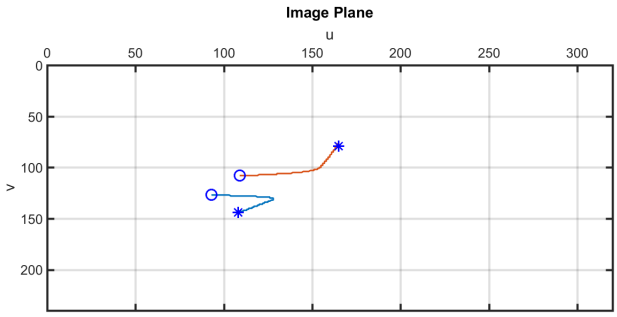


Figure 4.26: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.2, 0, 0)$

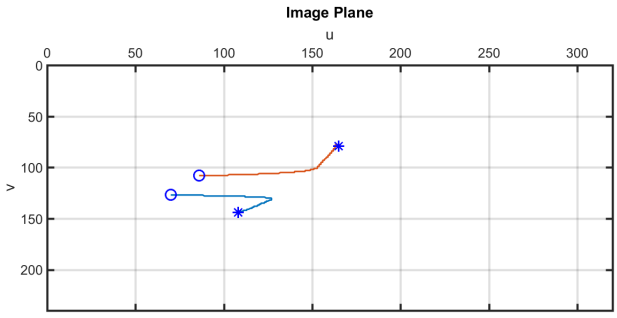


Figure 4.27: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.3, 0, 0)$

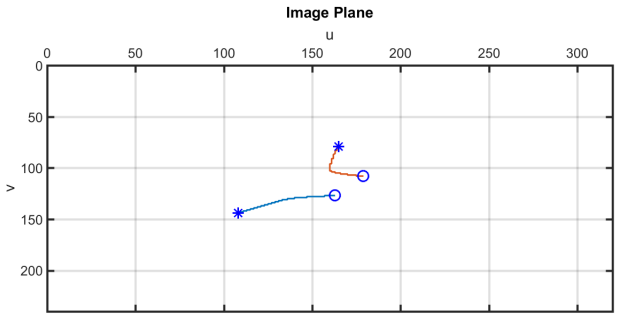


Figure 4.28: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.1, 0, 0)$

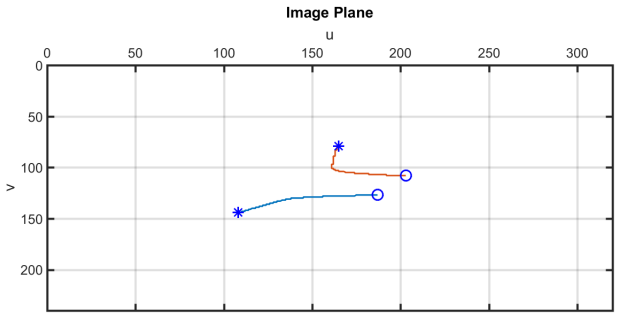


Figure 4.29: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.2, 0, 0)$

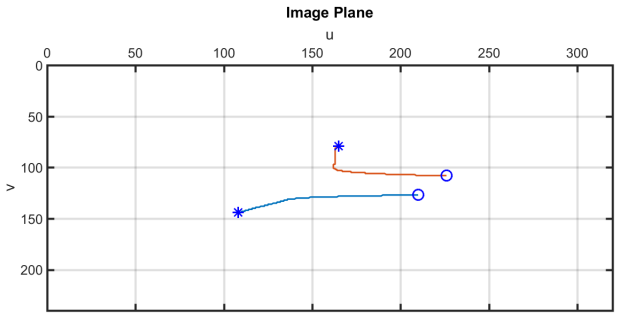


Figure 4.30: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.3, 0, 0)$

In the image plane both of the markers get close to the desired pixel coordinates for any of the initial conditions.

When using three markers on the target, the mobile robot still gets to the desired position only when $(x_0, z_0, \theta_0) = (0, 0, 0)$ as can be seen in Figure 4.31.

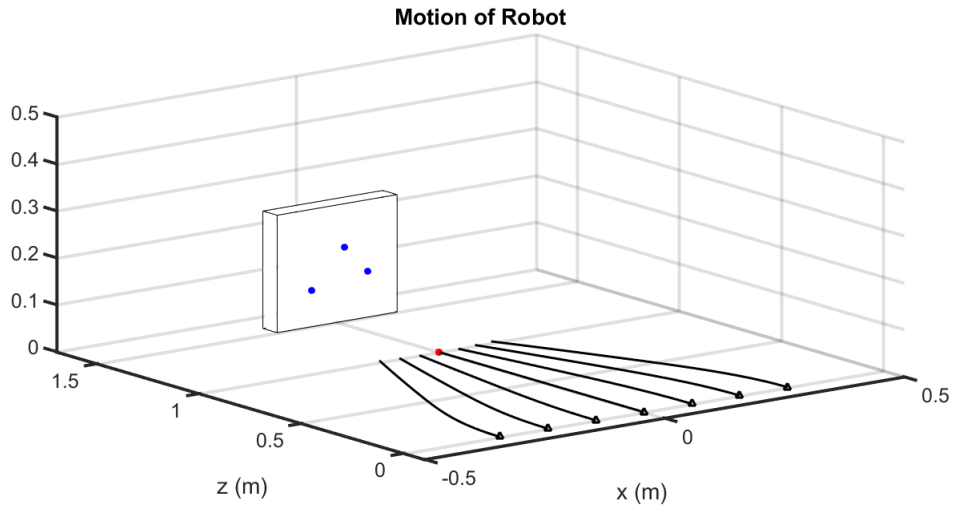


Figure 4.31: Motion of Robot Using Three Markers

Figures 4.32, 4.33, 4.34, 4.35, 4.36, 4.37, 4.38 show the trajectory followed by the markers on the image plane for each initial condition.

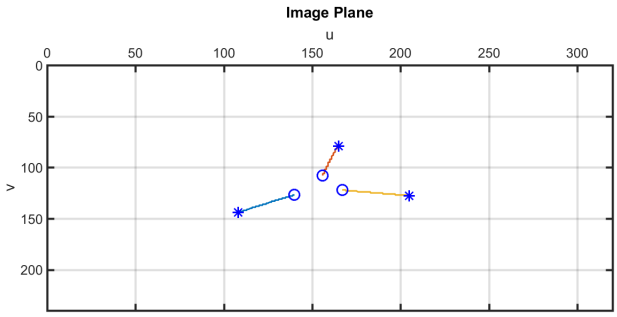


Figure 4.32: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0, 0, 0)$

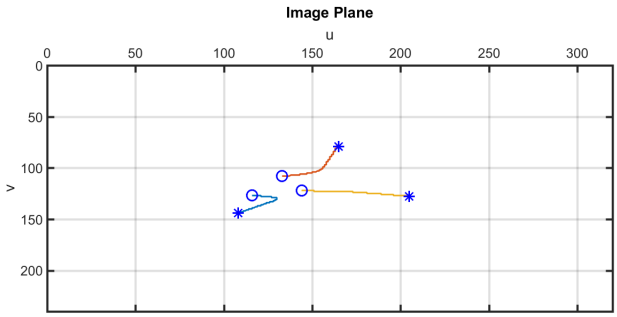


Figure 4.33: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.1, 0, 0)$

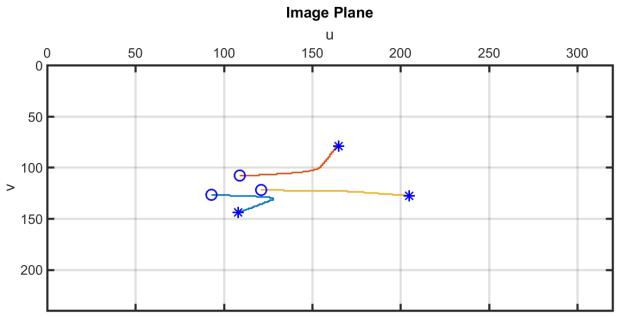


Figure 4.34: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.2, 0, 0)$

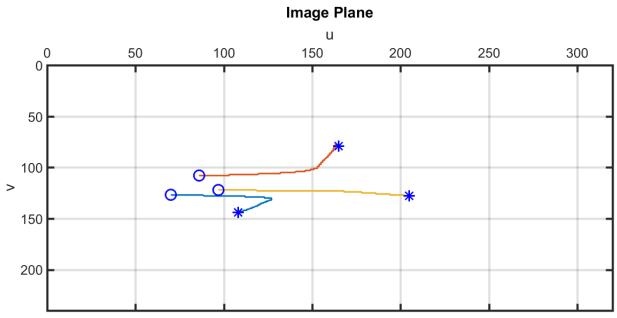


Figure 4.35: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.3, 0, 0)$

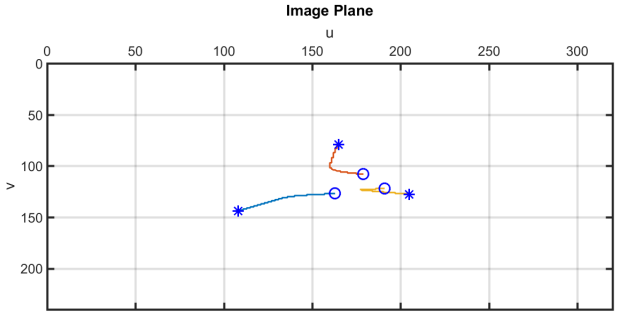


Figure 4.36: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.1, 0, 0)$

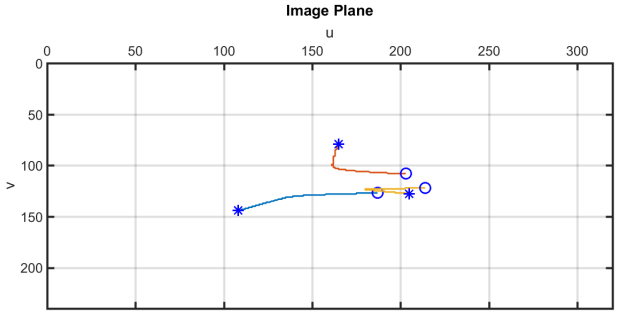


Figure 4.37: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.2, 0, 0)$

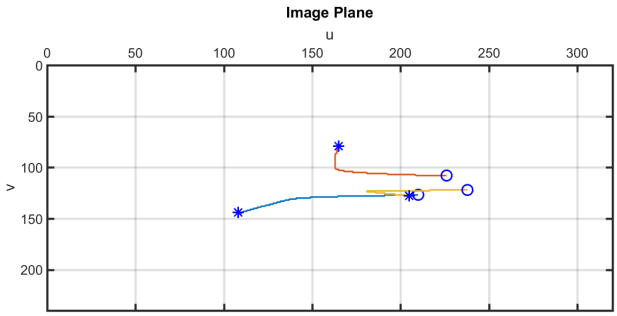


Figure 4.38: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.3, 0, 0)$

As seen above, there is no difference in using one, two or three markers on the target. In all three cases, starting the robot from $(x_0, z_0, \theta_0) \neq (0, 0, 0)$ causes the

robot to not finish at the desired position. This happens because there are different positions in the $xz$ plane from which the robot sees the markers the same way as it would see them from the desired or reference position $(x_{ref}, z_{ref}, \theta_{ref})$ (local minima).

In other words, the pixel errors are driven to zero even when the mobile robot does not reach the desired position. To show this idea, Figures 4.39, 4.40, 4.41, 4.42, 4.43, 4.44, 4.45, 4.46, 4.47 present the pixel errors when using one, two and three dots on the box.

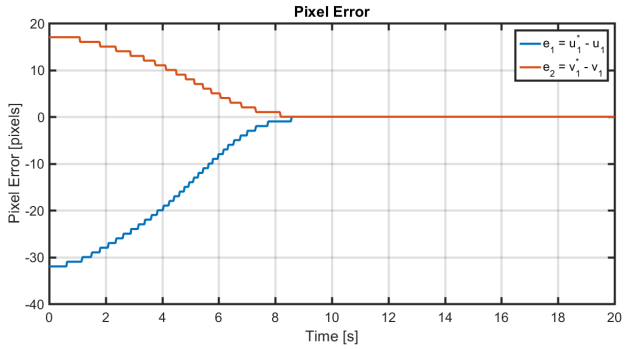


Figure 4.39: Pixel Errors for $(x_0, z_0, \theta_0) = (0, 0, 0)$ (One Marker)

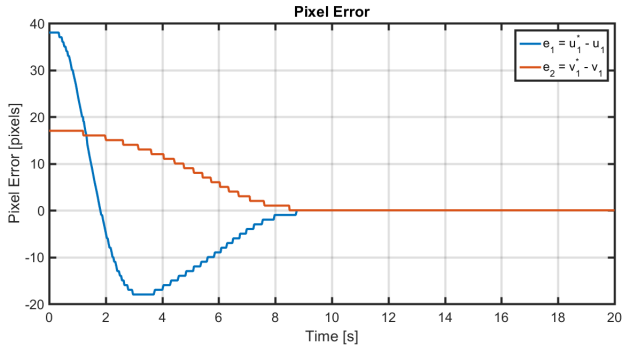


Figure 4.40: Pixel Errors for $(x_0, z_0, \theta_0) = (0.3, 0, 0)$ (One Marker)

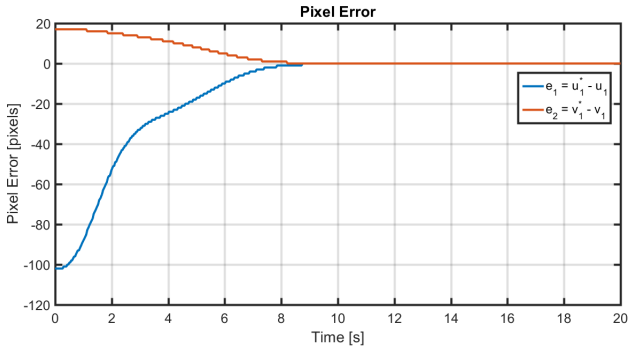


Figure 4.41: Pixel Errors for $(x_0, z_0, \theta_0) = (-0.3, 0, 0)$ (One Marker)

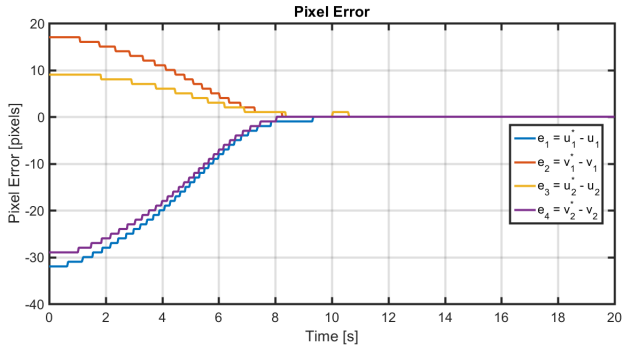


Figure 4.42: Pixel Errors for $(x_0, z_0, \theta_0) = (0, 0, 0)$ (Two Markers)

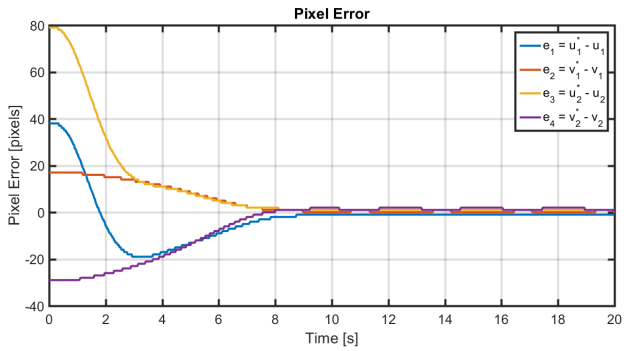


Figure 4.43: Pixel Errors for $(x_0, z_0, \theta_0) = (0.3, 0, 0)$ (Two Markers)

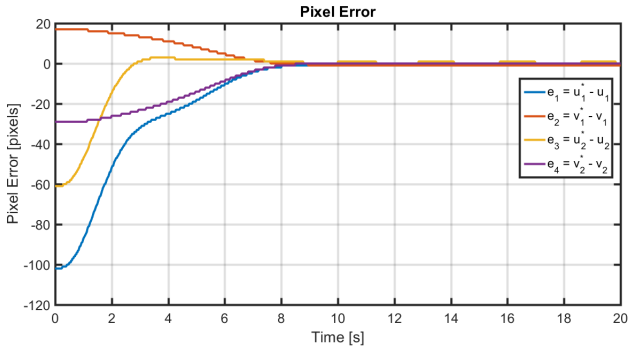


Figure 4.44: Pixel Errors for $(x_0, z_0, \theta_0) = (-0.3, 0, 0)$ (Two Markers)

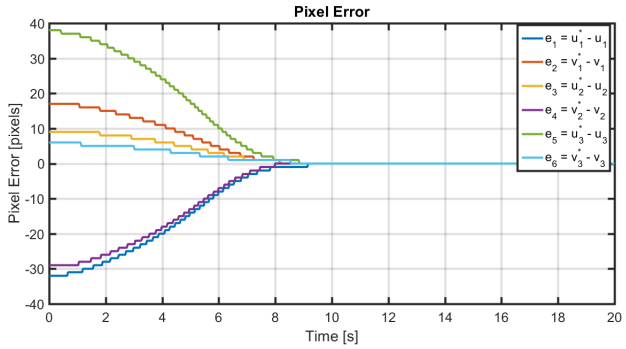


Figure 4.45: Pixel Errors for $(x_0, z_0, \theta_0) = (0, 0, 0)$ (Three Markers)

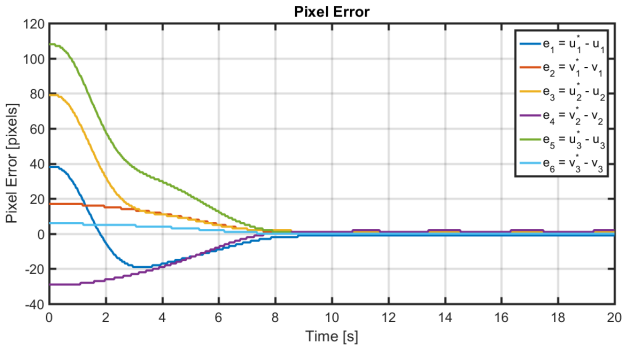


Figure 4.46: Pixel Errors for $(x_0, z_0, \theta_0) = (0.3, 0, 0)$ (Three Markers)

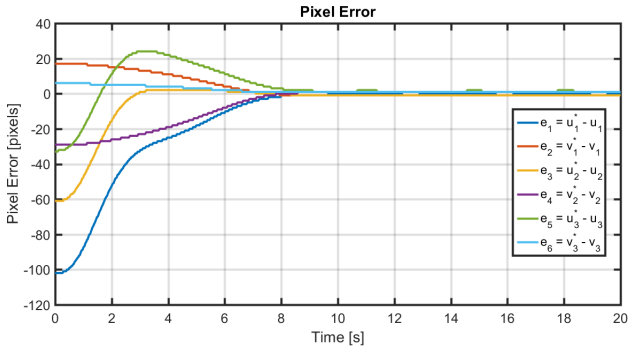


Figure 4.47: Pixel Errors for $(x_0, z_0, \theta_0) = (-0.3, 0, 0)$ (Three Markers)

As it can be seen from the figures above, the robot did not reach the desired position but the pixel errors are still driven to zero or close to zero.

### *4.5.3 Experimental Results*

In this section hardware or experimental results for the IBVS outer loop control are presented. Figure 4.48 shows how the mobile robot moves towards the desired position when the camera see one dot. It is in agreement with the simulation result presented above.

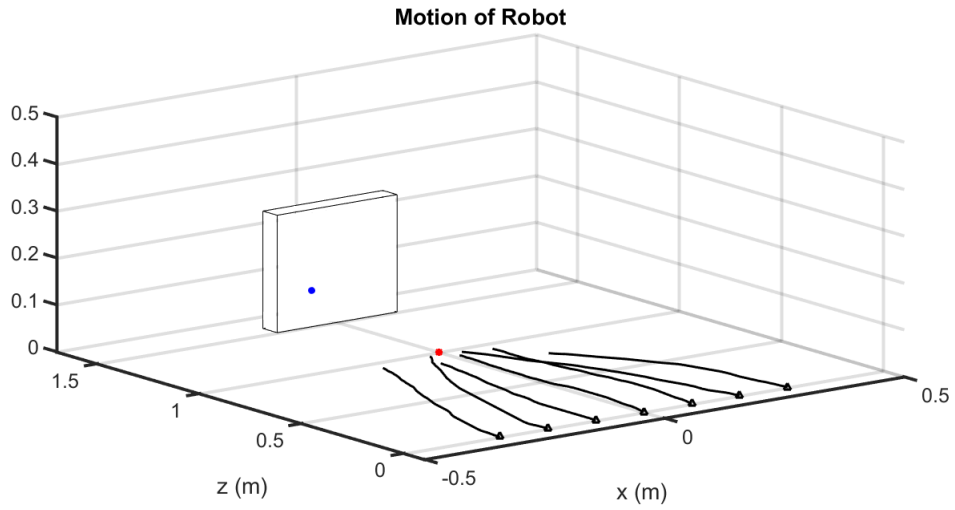


Figure 4.48: Motion of Robot Using One Marker - Experimental Result

Figures 4.49, 4.50, 4.51, 4.52, 4.53, 4.54, 4.55 show the trajectory followed by the marker on the image plane for each initial condition.

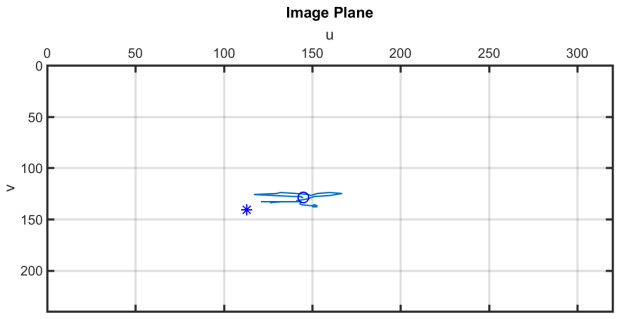


Figure 4.49: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0, 0, 0)$ - Experimental

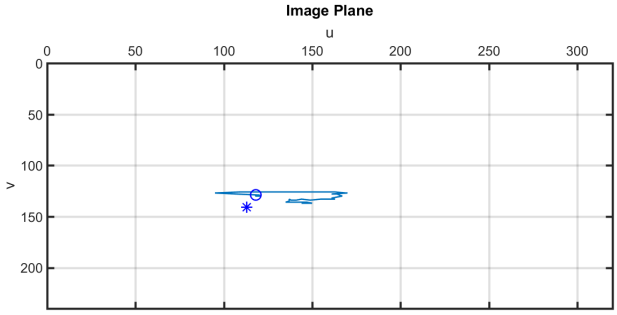


Figure 4.50: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.1, 0, 0)$ - Experimental

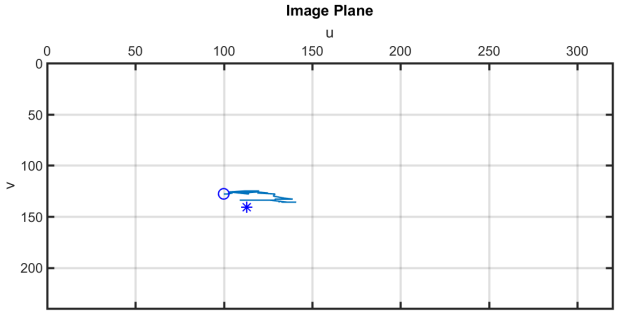


Figure 4.51: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.2, 0, 0)$ - Experimental

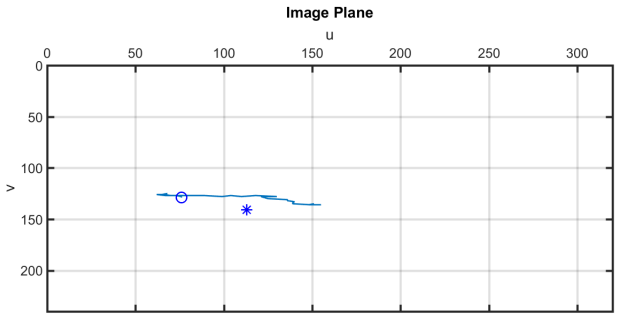


Figure 4.52: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.3, 0, 0)$ - Experimental

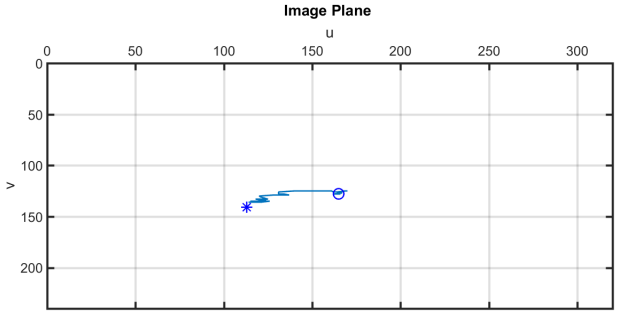


Figure 4.53: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.1, 0, 0)$ - Experimental

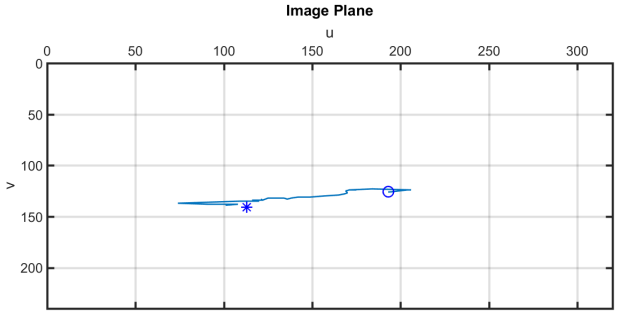


Figure 4.54: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.2, 0, 0)$ - Experimental

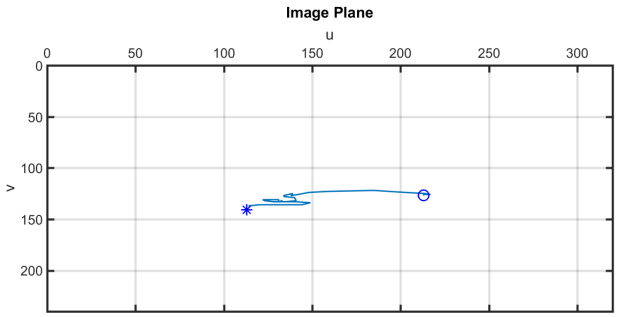


Figure 4.55: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.3, 0, 0)$ - Experimental

In Figure 4.56 the mobile robot moves trying to reach the desired position when 2 dots are in the FOV of the camera.

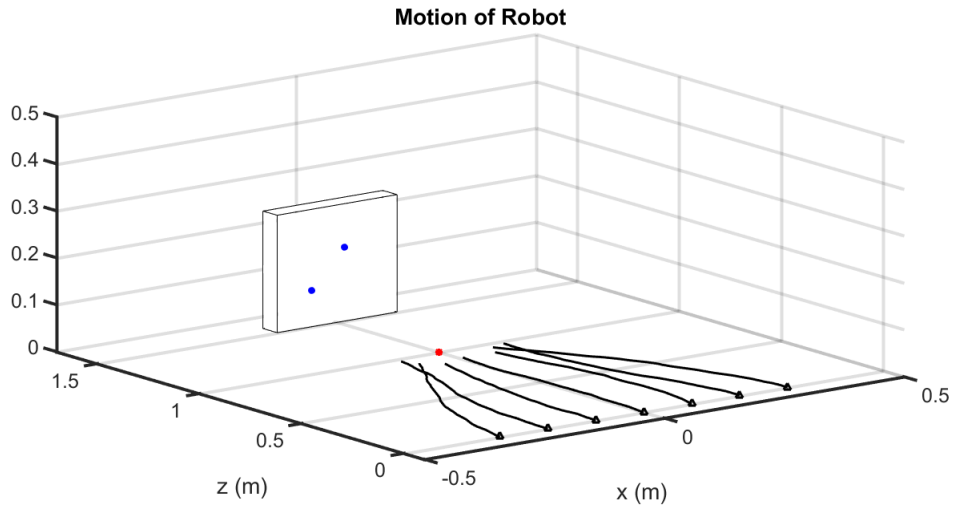


Figure 4.56: Motion of Robot Using Two Markers - Experimental Result

Figures 4.57, 4.58, 4.59, 4.60, 4.61, 4.62, 4.63 show the trajectory followed by the marker on the image plane for each initial condition.

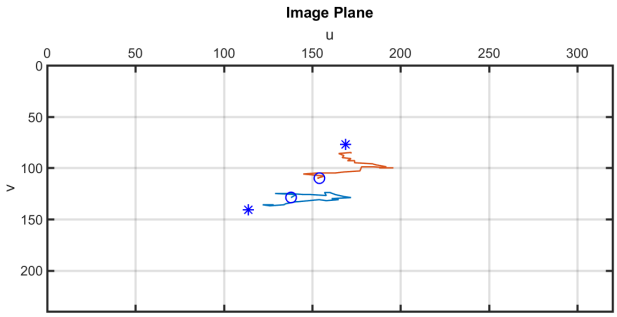


Figure 4.57: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0, 0, 0)$ - Experimental

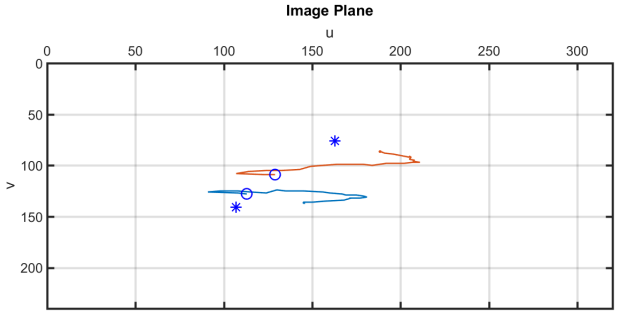


Figure 4.58: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.1, 0, 0)$ - Experimental

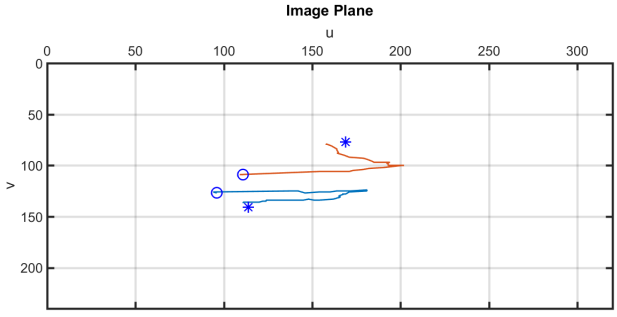


Figure 4.59: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.2, 0, 0)$ - Experimental

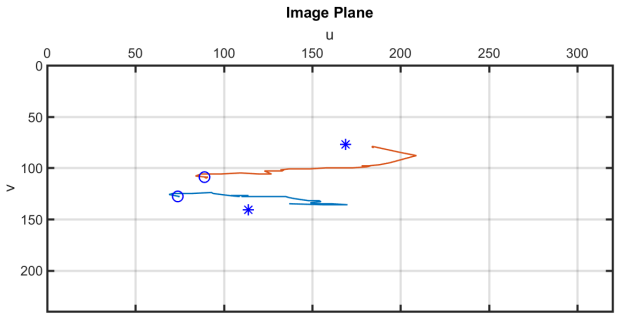


Figure 4.60: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.3, 0, 0)$ - Experimental

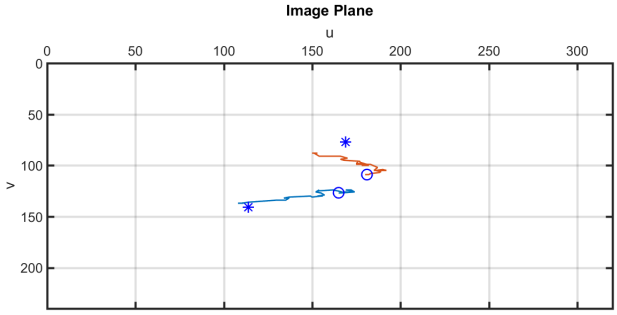


Figure 4.61: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.1, 0, 0)$ - Experimental

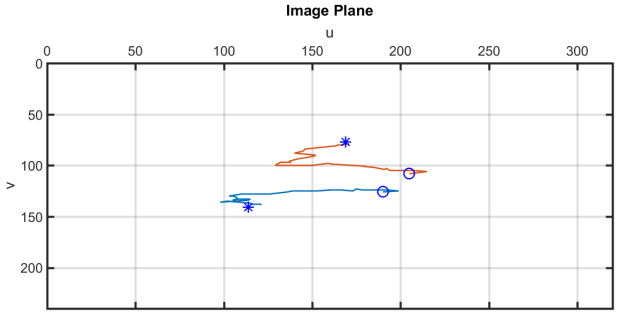


Figure 4.62: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.2, 0, 0)$ - Experimental

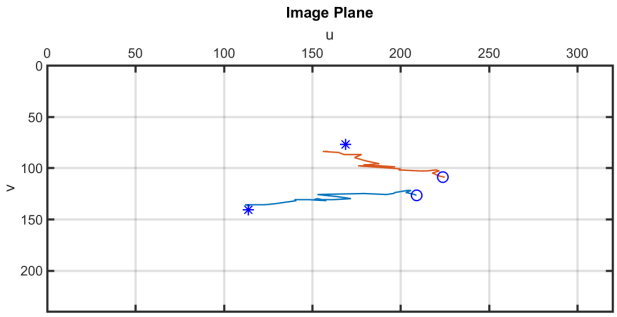


Figure 4.63: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.3, 0, 0)$ - Experimental

In Figure 4.64 the mobile robot moves on the $x - z$ plane trying to reach the desired position.

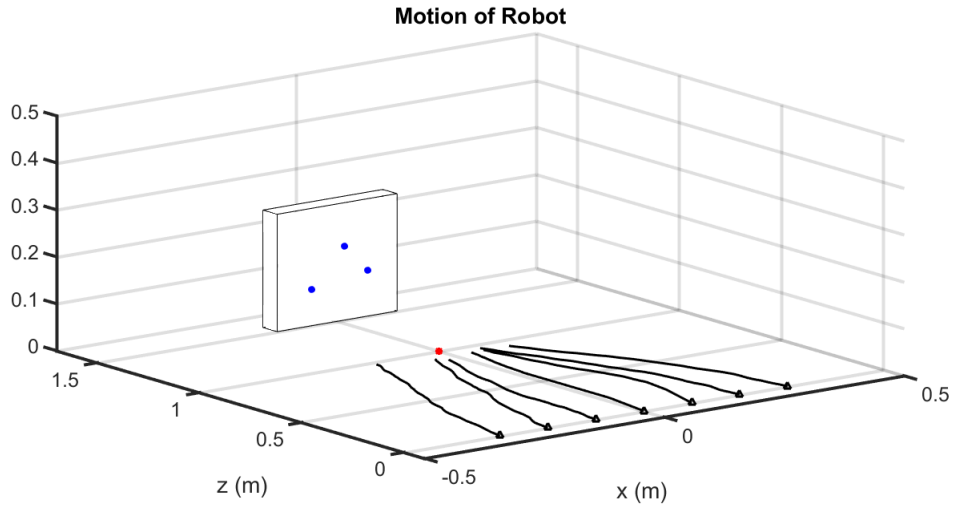


Figure 4.64: Motion of Robot Using Three Markers - Experimental Result

Figures 4.65, 4.66, 4.67, 4.68, 4.69, 4.70, 4.71 show the trajectory followed by the marker on the image plane for each initial condition.

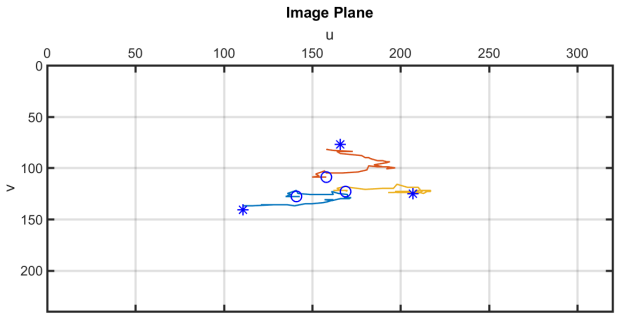


Figure 4.65: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0, 0, 0)$ - Experimental

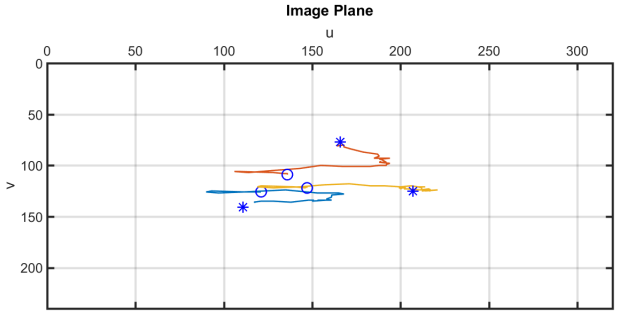


Figure 4.66: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.1, 0, 0)$ - Experimental

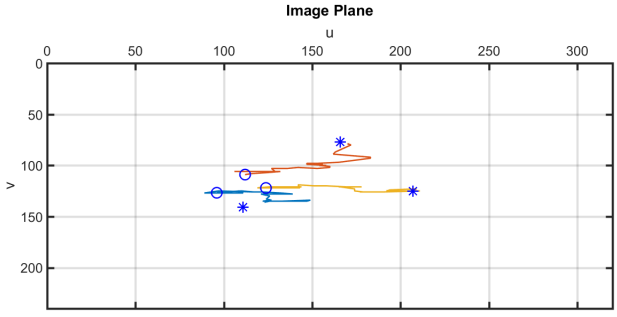


Figure 4.67: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.2, 0, 0)$ - Experimental

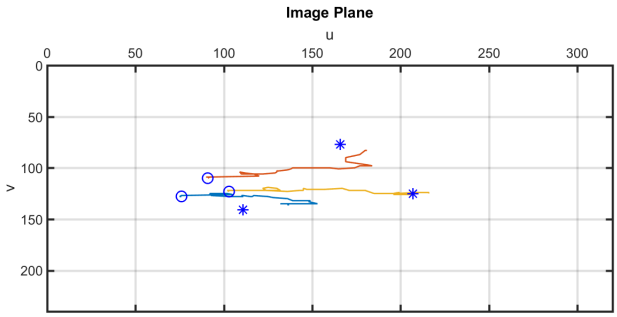


Figure 4.68: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (0.3, 0, 0)$ - Experimental

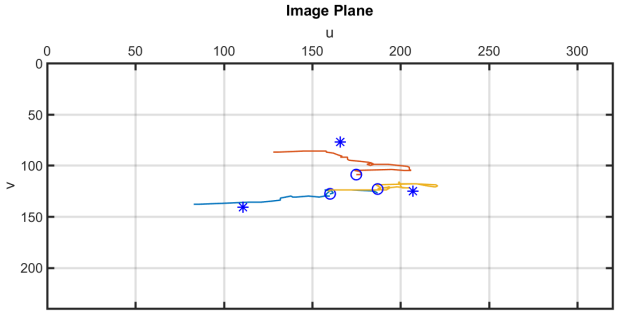


Figure 4.69: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.1, 0, 0)$ - Experimental

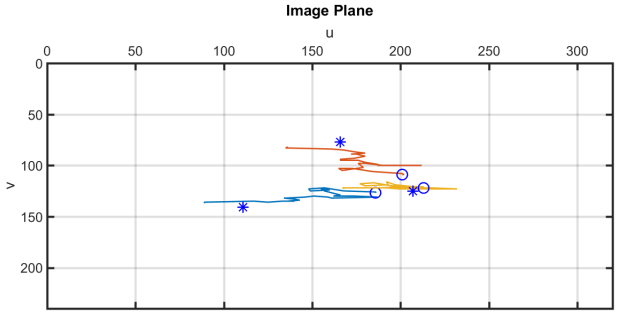


Figure 4.70: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.2, 0, 0)$ - Experimental

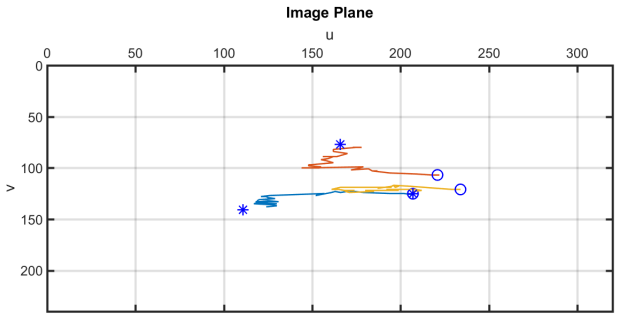


Figure 4.71: Trajectory of Markers on the Image Plane with $(x_0, z_0, \theta_0) = (-0.3, 0, 0)$ - Experimental

**CODE: PYTHON AND ARDUINO CODE.** The Python and Arduino code used for the Image-Based robot control can be found within Appendix C on page 202 and Appendix B on page 187.

## 4.6 Summary and Conclusion

This chapter has explored two methods of controlling a ground mobile robot, namely Position Based Visual Servoing (PBVS) and Image Based Visual Servoing (IBVS). The goal in PBVS is to estimate a target's position with respect to the robot and then to drive the robot to a desired position $(x_{ref}, z_{ref})$. As long as the chessboard remains in the FOV of the camera, the robot gets to the desired position. The goal in IBVS is to drive the pixel coordinates $(u, v)$ of the visual features (dots) to the desired pixel coordinates $(u_{ref}, v_{ref})$; by doing this the robot tries to get to the desired position $(x_{ref}, z_{ref}, \theta_{ref})$. Within IBVS the robot did not reach the desired pose due to the fact that the camera sees the dots the same way from different positions as it sees them from the desired or reference position.

# Chapter 5

# MINIMUM TIME OPTIMAL CONTROL FOR DIFFERENTIAL-DRIVE ROBOT

## 5.1 Introduction and Overview

Vehicular optimal control problems have been studied extensively since the early part of the 20th century. Progress in solving these problems has been driven primarily by applications in space and atmospheric flight [35].

Within this chapter, minimum time vehicle manoeuvring problem is addressed with a particular application to finding the minimum lap time for a Differential Drive Thunder Tumbler using two approaches, namely a camera-based and a noncamera-based. The minimum time vehicle manoeuvring problem is formulated as one of Optimal Control and is solved using mathematical programming methods [23].

The goal is to understand how one can use optimization concepts to obtain velocity profiles for the robot such that it travels a known path on the ground in minimum time. In short, the chapter presents results that will be useful for future optimization problems. The work of Casanova in [23] is mainly used within this chapter.

## 5.2 Optimal Control Theory

The main objective of optimal control is to determine control signals that will cause a process (plant) to satisfy some physical constraints and at the same time extremize (minimize or maximize) a chosen some performance criterion [31].

The formulation of optimal Control problems requires:

- Mathematical description (model) of the plant to be controlled

  A mathematical model for a generic system representing the rate of change of its states with respect to time, may be stated as follows:

$$\dot{\mathbf{x}}(t) = \mathbf{a}(\mathbf{x}(t), \mathbf{u}(t), t) \quad \mathbf{x}(t_0) = \mathbf{x}_0 \quad t \in [t_0, t_f] \tag{5.1}$$

  Here, $\mathbf{x}$ is the system states, and $\mathbf{u}$ is the control signal applied to the plant. The time history of state and control variables defined within the interval $[t_0, t_f]$ are referred to as state trajectory and control history respectively.

- Physical constraints

  The physical constraints in an optimal control problem are intended to limit the range of the state and control variables within values which are meaningful for the plant and for the problem which is being analyzed. A general physical constraint involving the state trajectory and the control history can be referred with the following:

$$c(t) = c(\mathbf{x}(t), \mathbf{u}(t), t) \leq 0 \quad t \in [t_0, t_f] \tag{5.2}$$

  A different kind of constraint is represented by constant control bounds. The definition reads:

$$\mathbf{u}_L \leq \mathbf{u}(t) \leq \mathbf{u}_U \quad t \in [t_0, t_f] \tag{5.3}$$

  A control history which satisfies the control constraints during the entire time interval is called an admissible control. In the same way, a state trajectory

which satisfies the state variable constraints during the entire time interval is referred to as a feasible trajectory.

- Performance criterion

  The definition of the problem involves the use of functionals. A functional $J$ is a rule of correspondance or a map, which assigns to each function $x(t)$ a unique real number. Intuitively, a functional may be seen as a "function of functions".

$$J(x(t)) = \int_{t_o}^{t_f} x(t)dt \tag{5.4}$$

  The objective of an optimal control problem is to minimize (or maximize) a quantitative measure of the performance of the plant. The most general definition for the performance measure involves a function of the final system states as well as a functional of the state trajectories and the control histories:

$$J = S(\mathbf{x}(t_f), t_f) + \int_{t_0}^{t_f} V(\mathbf{x}(t), \mathbf{u}(t), t)dt \tag{5.5}$$

  The performance measure characterizes the different types of optimal control problems, e.g. minimum time problems, minimum control effort problems, path tracking problems, etc.

Given the definitions above, a general optimal control problem is formally defined as follows:

$$
\begin{aligned}
\min_{u} \quad & J = S(\mathbf{x}(t_f), t_f) + \int_{t_0}^{t_f} V(\mathbf{x}(t), \mathbf{u}(t), t)dt \\
\text{subject to} \quad & \dot{\mathbf{x}}(t) = \mathbf{a}(\mathbf{x}(t), \mathbf{u}(t), t) \quad \mathbf{x}(t_0) = \mathbf{x}_0 \\
& \mathbf{c} \leq 0 \\
& \mathbf{u}_L \leq \mathbf{u}(t) \leq \mathbf{u}_U \\
\text{for all} \quad & t \in [t_0, t_f]
\end{aligned}
\tag{5.6}
$$

The task is to find an admissible control $\mathbf{u}^*$ which causes the system described by (5.1) to follow a feasible trajectory $\mathbf{x}^*$ which minimizes the performance measure $J$.

### *5.2.1 Necessary Conditions for Optimality*

Here the first order, necessary conditions for optimality are presented (as in [23]), i.e. the conditions that the state trajectory and the control history must satisfy when the performance measure $J$ is on a relative *extremum*.

The procedure is similar to the equivalent problem in calculus of finding an extremum of a function. Consider a continuous and differentiable function of a single variable $f(q)$. The theory of calculus states that the necessary condition for $q^*$ to be an extremum is that the first derivative of $f(q)$ vanishes when $q \to q^*$. If the increment of $f(q)$ is written for an arbitrary small $\Delta q$, such increment may be approximated with the differential of $f(q)$:

$$
f(q + \Delta q) - f(q) = df(q, \Delta q) + o(\Delta q) \equiv df(q, \Delta q) = \frac{df(q)}{dq} \cdot \Delta q \tag{5.7}
$$

where $o(\Delta q)$ represents the higher order terms in the series expansion of $f(q)$ when $\Delta q$ tends to 0. Hence, the necessary condition for $f(q)$ to have an extremum at $q = q^*$ is also that its differential vanishes:

$$df(q^*, \Delta q) = 0 \tag{5.8}$$

In the corresponding problem of Calculus of Variations the task is to define the first order approximation to the increment of a functional. Then the necessary condition for having an extremum will be that such approximation be equal to zero. For a differentiable functional $J(\mathbf{x})$ its increment may be written as follows:

$$J(\mathbf{x} + \delta\mathbf{x}) - J(\mathbf{x}) = \Delta J(\mathbf{x}^*, \delta\mathbf{x}) = \delta J(\mathbf{x}, \delta\mathbf{x}) + o(\delta\mathbf{x}) \tag{5.9}$$

Here, $\delta J(\mathbf{x}, \delta\mathbf{x})$ is called the *variation* of a functional and is the equivalent of the differential of a function in the theory of calculus. The function $\delta\mathbf{x}$ is an arbitrarily small perturbation distributed along the trajectory $\mathbf{x}$. Figure 5.1 visualizes qualitatively a generic perturbation $\delta x$ for the case of a scalar function.

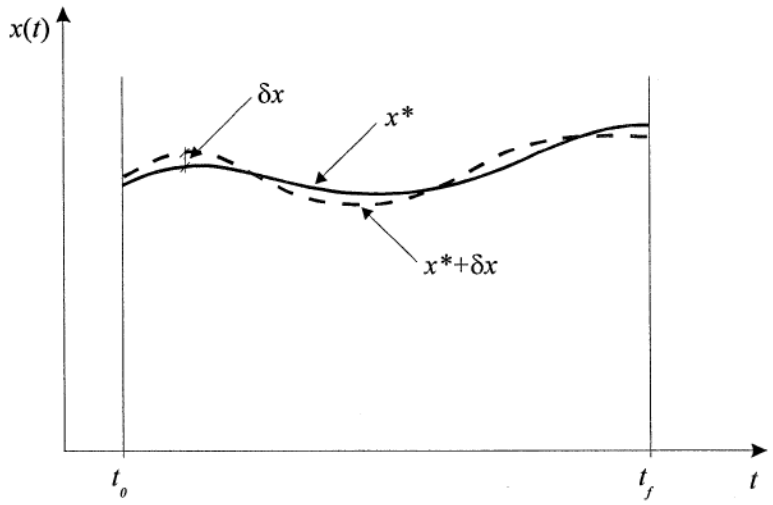


Figure 5.1: Generic Perturbation $\delta x$

When $\delta\mathbf{x}$ vanishes, the increment of a functional may be approximated with its variation. Then, the necessary condition for $J$ to have an extremum in $\mathbf{x}^*$ is that its variation must vanish on $\mathbf{x}^*$, that is:

$$\Delta J(\mathbf{x}^*, \delta\mathbf{x}) \cong \delta\, J(\mathbf{x}^*, \delta\mathbf{x}) = 0 \tag{5.10}$$

for all admissible $\delta\mathbf{x}$ (which means that if $\mathbf{\Omega}$ is the domain of $J$, $\mathbf{x}^* + \delta\mathbf{x}$ must still be a member of such domain).

Even if one is able to find a curve $\mathbf{x}^*$ which satisfies (5.10), there is no certainty that such a curve would be an extremum, as (5.10) only states a necessary condition. Furthermore, even if $\mathbf{x}^*$ were an extremum, nothing could be said on whether it is a local minimum or local maximum. Finally, it is not even guaranteed that the functional is differentiable at the extremum $\mathbf{x}^*$.

As well as in the case of theory of calculus, where the second derivative of a function establishes necessary and sufficient conditions for either local minimum or maximum to occur, the second order variation of a functional may be defined, and necessary and sufficient conditions of optimality may be derived. As a sufficient condition for minimum the second variation $\delta^2 J > 0$ and for maximum $\delta^2 J < 0$ [31]. However, this involves a rather complex manipulation of the problem equations which does not lead to something practical. Conversely, the necessary conditions for optimality provide a convenient starting point to use for searching a solution [23].

Next the necessary conditions for optimality for optimal control problems without state and control constraints and with fixed end time is reviewed. Then, control constraints are introduced and the Pontryagin's Minimum Principle as a general statement of the necessary conditions for optimality is shown.

**Unconstrained Optimal Control problems with fixed end time.** As in [23], consider a system with $p$ state variables and $q$ control variables described by the following set of first-order non-linear differential equations:

$$\dot{\mathbf{x}}(t) = \mathbf{a}(\mathbf{x}(t), \mathbf{u}(t), t) \qquad t \in [t_0, t_f] \tag{5.11}$$

Final time is fixed and that the initial conditions are given and are fixed as well is assumed:

$$\mathbf{x}(t_0) = \mathbf{x}_0 \tag{5.12}$$

The task is to find a control history $\mathbf{u}^*$ which causes the plant to follow a trajectory $\mathbf{x}^*$ which minimizes the performance measure:

$$J(\mathbf{u}) = S(\mathbf{x}(t_f), t_f) + \int_{t_0}^{t_f} V(\mathbf{x}(t), \mathbf{u}(t), t)\, dt \tag{5.13}$$

The functional $J$ is assumed to be dependent only on the control $\mathbf{u}$. This is because any control history $\mathbf{u}$ univocally determines a state trajectory $\mathbf{x}$ and also because the initial states $\mathbf{x}_0$ as well as the final time $t_f$ are fixed.

Adjoining the differential equations to the performance measure by introducing $p$ Lagrange multipliers $\boldsymbol{\lambda}(t)$, one has the following:

$$\bar{J}(\mathbf{u}) = S(\mathbf{x}(t_f), t_f) + \int_{t_0}^{t_f} \left[V(\mathbf{x}, \mathbf{u}, t) + \boldsymbol{\lambda}^T \cdot (\mathbf{a}(\mathbf{x}, \mathbf{u}, t) - \dot{\mathbf{x}})\right] dt \tag{5.14}$$

The last term in the integrand of Equation (5.14) may be solved by parts (using $\int u dv = uv - \int v du$) in order to eliminate the state derivatives and the result reads:

$$\int_{t_0}^{t_f} -\boldsymbol{\lambda}^T \cdot \dot{\mathbf{x}}\, dt = \boldsymbol{\lambda}(t_0)^T \cdot \mathbf{x}(t_0) - \boldsymbol{\lambda}(t_f)^T \cdot \mathbf{x}(t_f) + \int_{t_0}^{t_f} \dot{\boldsymbol{\lambda}}^T \cdot \mathbf{x}\, dt \tag{5.15}$$

Substituting Equation (5.15) in Equation (5.14) the following result is obtained:

$$\bar{J}(\mathbf{u}) = S(\mathbf{x}(t_f), t_f) + \boldsymbol{\lambda}(t_0)^T \cdot \mathbf{x}(t_0) - \boldsymbol{\lambda}(t_f)^T \cdot \mathbf{x}(t_f) + \int_{t_0}^{t_f} \left[ V(\mathbf{x}, \mathbf{u}, t) + \boldsymbol{\lambda}^T \cdot \mathbf{a}(\mathbf{x}, \mathbf{u}, t) + \dot{\boldsymbol{\lambda}}^T \cdot \mathbf{x} \right] dt \tag{5.16}$$

Now the *Hamiltonian* function is introduced, which is defined as:

$$\mathcal{H}(\mathbf{x}, \mathbf{u}, \boldsymbol{\lambda}, t) = V(\mathbf{x}, \mathbf{u}, t) + \boldsymbol{\lambda}^T \cdot \mathbf{a}(\mathbf{x}, \mathbf{u}, t) \tag{5.17}$$

By using the Hamiltonian function in Equation (5.16) the following is obtained:

$$\bar{J}(\mathbf{u}) = S(\mathbf{x}(t_f), t_f) + \boldsymbol{\lambda}(t_0)^T \cdot \mathbf{x}(t_0) - \boldsymbol{\lambda}(t_f)^T \cdot \mathbf{x}(t_f) + \int_{t_0}^{t_f} \left[ \mathcal{H}(\mathbf{x}, \mathbf{u}, \boldsymbol{\lambda}, t) + \dot{\boldsymbol{\lambda}}^T \cdot \mathbf{x} \right] dt \tag{5.18}$$

by differentiating Equation (5.18) with respect to $\mathbf{u}$ and $\mathbf{x}$, the variation of the functional $\bar{J}(\mathbf{u})$ is written:

$$\delta \bar{J}(\mathbf{u}, \delta\mathbf{u}) = \left[ \left( \frac{\partial S}{\partial \mathbf{x}} - \boldsymbol{\lambda} \right)^T \cdot \delta\mathbf{x} \right]_{t=t_f} + \left( \boldsymbol{\lambda}^T \cdot \delta\mathbf{x} \right)_{t=t_0} + \int_{t_0}^{t_f} \left[ \left( \frac{\partial \mathcal{H}}{\partial \mathbf{x}} + \dot{\boldsymbol{\lambda}} \right)^T \cdot \delta\mathbf{x} + \left( \frac{\partial \mathcal{H}}{\partial \mathbf{u}} \right)^T \cdot \delta\mathbf{u} \right] dt \tag{5.19}$$

Since the initial state values are fixed, the second term in the right hand member of Equation (5.19) is equal to zero. Then, as the control history univocally determines the state trajectory, it is assumed that the variation of the state trajectory $\delta\mathbf{x}$ depends on the variation of the control $\delta\mathbf{u}$. However, rather than trying to express $\delta\mathbf{x}$ as a function of $\delta\mathbf{u}$, the Lagrange multipliers are chosen in such a way that the terms in Equation (5.19) which multiply $\delta\mathbf{x}$ vanish [23]. In doing so, the following is obtained:

$$\dot{\boldsymbol{\lambda}} = -\frac{\partial \mathcal{H}}{\partial \mathbf{x}} = -\frac{\partial V(\mathbf{x}, \mathbf{u}, t)}{\partial \mathbf{x}} - \boldsymbol{\lambda}^T \cdot \frac{\partial \mathbf{a}(\mathbf{x}, \mathbf{u}, t)}{\partial \mathbf{x}} \tag{5.20}$$

$$\boldsymbol{\lambda}(t_f) = \left(\frac{\partial S}{\partial \mathbf{x}}\right)_{t=t_f} \tag{5.21}$$

For an extremum to occur, the variation of the functional must be zero for any arbitrary $\delta\mathbf{u}$. Therefore, after deleting all the terms equal to 0 in Equation (5.19), this condition reads:

$$\delta\bar{J}(\mathbf{u}, \delta\mathbf{u}) = \int_{t_0}^{t_f} \left[\left(\frac{\partial \mathcal{H}}{\partial \mathbf{u}}\right)^T \cdot \delta\mathbf{u}\right] dt = 0 \tag{5.22}$$

However, Equation (5.22) is satisfied only if:

$$\frac{\partial \mathcal{H}}{\partial \mathbf{u}} = \frac{\partial V(\mathbf{x}, \mathbf{u}, t)}{\partial \mathbf{u}} + \boldsymbol{\lambda}^T \cdot \frac{\partial \mathbf{a}(\mathbf{x}, \mathbf{u}, t)}{\partial \mathbf{u}} = \mathbf{0} \tag{5.23}$$

Note that the plant Equation (5.11) can be written in terms of the Hamiltonian as:

$$\dot{\mathbf{x}}(t) = \frac{\partial \mathcal{H}}{\partial \boldsymbol{\lambda}} \tag{5.24}$$

Equation (5.20), (5.23) and (5.24) are also known as the *co-state, control* and *state equations*, respectively. In summary, to find a control history $\mathbf{u}(t)$ which produces a stationary point of the performance measure $J$, the following $2p$ differential equations must be solved:

$$\dot{\mathbf{x}}(t) = \mathbf{a}(\mathbf{x}, \mathbf{u}, t) \tag{5.25}$$

$$\dot{\boldsymbol{\lambda}} = -\frac{\partial \mathcal{H}}{\partial \mathbf{x}} \tag{5.26}$$

for any $t \in [t_0, t_f]$, where $\mathbf{u}(t)$ is determined by $q$ algebraic equations:

$$\frac{\partial \mathcal{H}}{\partial \mathbf{u}} = \mathbf{0} \tag{5.27}$$

The boundary conditions for Equations (5.25) and (5.26) are split. That is, some are specified for $t = t_0$ and some are specified for $t = t_f$.

$$\mathbf{x}(t_0) = \mathbf{x}_0 \tag{5.28}$$

$$\boldsymbol{\lambda}(t_f) = \left(\frac{\partial S}{\partial \mathbf{x}}\right)_{t=t_f} \tag{5.29}$$

Thus, a solution for a non-linear two-point boundary-value problem (TPBVP) is needed. In [31], [32] it is shown in more detail how to find the necessary conditions for optimality for a variety of systems.

**Optimal Control problems with control boundaries.** In the previous formulation it was assumed that the control $\mathbf{u}(t)$ and the states $\mathbf{x}(t)$ are *unconstrained*, i.e. there are no limitations on the magnitudes of the control and state variables. In reality the physical systems to be controlled in an optimum manner have some *constraints* on their inputs, internal variables and/or outputs.

The above framework is extended, as in [23], for dealing with Optimal Control problems with control constraints. The generalization of the necessary conditions for optimality leads to the Pontryagin's Minimum Principle.

Consider the analogous case in calculus first. Given a function $f(q)$ as in Figure 5.2, and if there is no restriction for the values that $q$ may take, this function has a local minimum in $q = q^*$.

Here, the necessary condition that the derivative of $f(q)$ vanishes at the extremum applies. If the value of $q$ is restricted within the interval $[q_1, q_2]$, the function $f(q)$ has a minimum point in this interval when $q = q_2$ but here the above necessary condition does not apply. Instead, the necessary conditions for $f(q)$ to have relative minima at the end points of the interval are as follows. If the linear part of the increment of $f(q)$

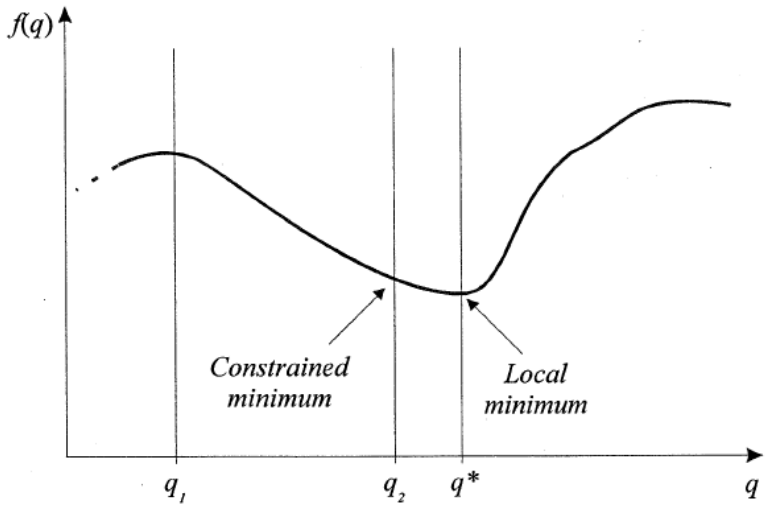


Figure 5.2: Constrained and Local Minimum

is considered, i.e. the differential of $f(q)$, such increment must always be positive for any *admissible* variation $\Delta q$:

$$\Delta f(q_1, \Delta q) \cong \left(\frac{\partial f(q_1)}{\partial q}\right) \cdot \Delta q \geq 0 \qquad \forall \Delta q \geq 0 \tag{5.30}$$

$$\Delta f(q_2, \Delta q) \cong \left(\frac{\partial f(q_2)}{\partial q}\right) \cdot \Delta q \geq 0 \qquad \forall \Delta q \leq 0 \tag{5.31}$$

In other words, when $q = q_1$ it is only allowed to increase $q$, and the condition for this point to be a local minimum is that the differential of $f(q)$ must be zero or positive. Instead, when $q = q_2$, it is only allowed to decrease $q$, and the condition for this point to be a local minimum is again that the differential of $f(q)$ must be zero or positive.

If the same idea is applied to the corresponding problem of Calculus of Variations, the condition stated in Equation (5.22) changes as follows:

$$\delta \bar{J}(\mathbf{u}, \delta\mathbf{u}) = \int_{t_0}^{t_f} \left[\left(\frac{\partial \mathcal{H}}{\partial \mathbf{u}}\right)^T \cdot \delta\mathbf{u}\right] dt \geq 0 \tag{5.32}$$

Here, $\delta\mathbf{u}$ must be admissible. That is, the control $\mathbf{u}+\delta\mathbf{u}$ must not violate the control constraints. If the integrand in Equation (5.32) is expanded by writing explicitly the variation of the Hamiltonian, the following is obtained:

$$\int_{t_0}^{t_f} \Big[\mathcal{H}(\mathbf{x}, \mathbf{u}+\delta\mathbf{u}, \boldsymbol{\lambda}, t) - \mathcal{H}(\mathbf{x}, \mathbf{u}, \boldsymbol{\lambda}, t)\Big] dt \geq 0 \tag{5.33}$$

Equation (5.33) is satisfied only if:

$$\mathcal{H}(\mathbf{x}, \mathbf{u}+\delta\mathbf{u}, \boldsymbol{\lambda}, t) \geq \mathcal{H}(\mathbf{x}, \mathbf{u}, \boldsymbol{\lambda}, t) \qquad \forall\, admissible\ \delta\mathbf{u} \tag{5.34}$$

The previous relation, which means that the *necessary condition for the constrained optimal control system is that the optimal control should minimize the Hamiltonian*, is the main contribution of the Pontryagin Minimum Principle [31]. Thus, Equation (5.34), together with (5.25),(5.26) and the boundary conditions in (5.28) and (5.29) constitute the necessary conditions for optimality for the general case of an Optimal Control problem with control constraints.

### *5.2.2 Indirect Methods*

*Indirect methods* aim to solve an Optimal Control problem by applying the optimality conditions explicitly. This involves the setting up the adjoint Equations (5.26) and (5.29), and the optimality condition (5.27), and requires an iterative procedure to solve the resulting non-linear two-point boundary value problem. The general approach consists of using an initial guess to obtain a solution to a problem where one or more of the optimality conditions is not satisfied. The solution is then used to adjust the initial guess in order to force the next solution to be closer to satisfying all the necessary conditions, until the iterative procedure eventually converges [23].

### 5.2.3 Direct Methods

A class of methods for solving Optimal Control problems, known as *direct transcription methods* does not use the necessary conditions for optimality and the Pontryagin's Minimum Principle. Instead, the original Optimal Control problem is converted into a *Non-Linear Programming* problem and is solved directly using mathematical programming techniques [23].

The basic concept of direct methods is that the continuous control history is replaced with a discrete approximation. It is assumed that the control input can only be adjusted at a number of fixed positions along the trajectory, while the intermediate values are estimated by means of interpolation techniques. Let $\mathbf{u_n}$ be the vector of discrete control parameters and $\mathbf{t_n}$ be the vector of the corresponding instances within the time interval $[t_0, t_f]$. The control parameters univocally determine the control history which, in turn, determine the system state trajectory. Therefore the performance measure and the constraints may be expressed directly as functions of these control parameters. Hence, the original optimal control problem may be stated as a Non-Linear Programming problem, i.e. to find the set of control parameters $\mathbf{u_n}$ which minimizes a generic non-linear multi-variable function subjected to general equality and inequality constraints:

$$\begin{aligned} \min_{\mathbf{u_n}} \quad & J(\mathbf{u_n}) \\ \text{subject to} \quad & c_i(\mathbf{u_n}) = 0 \quad i \in E \\ & c_i(\mathbf{u_n}) \geq 0 \quad i \in I \end{aligned} \tag{5.35}$$

Here, $E$ and $I$ represent the set of equality and inequality constraints respectively. The performance measure $J(\mathbf{u_n})$ is also called the objective function.

The first order, necessary conditions for the set of independent variables $\mathbf{u_n}$ to be a

constrained minimizer for the function $J(\mathbf{u_n})$ are known as the *Karush-Kuhn-Tucker* conditions, and a solution for the problem defined by Equation (5.35) is often referred to as *KKT point* [23] [33].

**Karush-Kuhn-Tucker conditions.** Consider the constrained minimization problem in (5.35). Let us adjoin the constraints to the objective function by using the Lagrange multipliers $\boldsymbol{\lambda}$ to form the Lagrangian function:

$$\mathcal{L}(\mathbf{u_n}, \boldsymbol{\lambda}) = J(\mathbf{u_n}) + \sum_{i \in E \cup I} \lambda_i \cdot c_i(\mathbf{u_n}) = J(\mathbf{u_n}) + \boldsymbol{\lambda}^T \cdot \mathbf{c}(\mathbf{u_n}) \tag{5.36}$$

The first order necessary conditions for a local minimizer $\mathbf{u_n}^*$, $\boldsymbol{\lambda}^*$ for the problem defined by Equation (5.35) are defined as followed [33]:

$$\begin{aligned}
&\nabla_{\mathbf{u_n}} \mathcal{L}(\mathbf{u_n}^*, \boldsymbol{\lambda}^*) = \frac{\partial \mathcal{L}(\mathbf{u_n}^*, \boldsymbol{\lambda}^*)}{\partial \mathbf{u_n}} = 0 \\
&c_i(\mathbf{u_n}^*) = 0, \qquad \forall\, i \in E \\
&c_i(\mathbf{u_n}^*) \geq 0, \qquad \forall\, i \in I \\
&\lambda_i^* \geq 0, \qquad \forall\, i \in I \\
&\lambda_i^*\, c_i(\mathbf{u_n}^*) = 0, \qquad \forall\, i \in E \cup I
\end{aligned} \tag{5.37}$$

The necessary conditions defined above are the *Karush-Kuhn-Tucker conditions*. The final condition in (5.37) are *complementarity conditions*, they imply that either constraint $i$ is active or $\lambda_i = 0$, or posibly both. In particular, the Lagrange multipliers corresponding to inactive inequality constraints are zero.

**Direct transcription methods.** For transcribing the optimal problem into an Non-Linear Programming Problem several transcription methods exist [25], [23]. Two main directions are the direct shooting and full collocation. In direct shooting, the control history $u(t)$ is discretized into a finite number of variables $(u_1, u_2, ..., u_N)$. The performance index and constraints are calculated by propagating through the

differential equations. Since only the control inputs are considered as optimization variables, this approach results in relatively small-scale problems. The disadvantage however is the chance of numerical difficulties for the applied solver, as a result of the large difference in sensitivity to early and late controls. This effect is even stronger for nonlinear and unstable systems. Multiple shooting methods address this problem by dividing the problem in multiple shooting segments. Each segment is treated as a direct shooting segment, and the segments are connected by defect constraints. As such, the problem is partially decoupled, leading to better conditioning of Jacobian.

In the full or direct collocation approach, the shooting segment has exactly the length of one discretization interval. Since the states at each segment are connected, this means that not only the control inputs, but also the discretized state trajectory $(x_1, x_2, ..., x_N)$ is included in the set of decision variables. The dynamics of the system may be replaced with a finite difference approximation by introducing the vector of *defects* $\boldsymbol{\xi}$. Different discretization schemes may be employed for this purpose. The trapezoidal method is used here:

$$\boldsymbol{\xi}_i = +\mathbf{x}_{i-1} - \mathbf{x}_i + \frac{\Delta_i}{2}\left[\mathbf{a}_i + \mathbf{a}_{i-1}\right] \qquad i = 1, ..., N \tag{5.38}$$

where $\Delta_i$ is the constant integration step size and $\mathbf{a}$ comes from the systems dynamics, i.e. $\dot{\mathbf{x}} = \mathbf{a}(\mathbf{x}, \mathbf{u}, t)$.

The full collocation approach leads to maximal decoupling [25]. Problems with a moderate number of states in the dynamics but a high number of discretization intervals and very nonlinear dynamics are often transcribed using full collocation. Typically, shooting methods are used for problems with a high amount of states, since applying full collocation simply would lead to a too large problem. In this thesis, full collocation method is used.

### 5.2.4 Approximation of Direct to Indirect Methods

Does a solution obtained from any direct method satisfy also the necessary conditions for optimality?. The answer to this question is, in general, yes. It can be shown that the discrete Lagrange multipliers associated with the solution to an optimal control problem obtained by using a direct collocation method are, in fact, a discrete approximation to the solution of the adjoint co-state equations [23], [29], [30].

It is important to point out that direct solution methods only return *approximate solutions* as a consequence of the problem discretization.

As in [23], it is now described how the solution obtained from a direct method satisfies the necessary conditions for optimality.

Consider a general unconstrained optimal control problem.

$$\begin{aligned} \min_{u} \quad & J = S(\mathbf{x}(t_f), t_f) \\ \text{subject to} \quad & \dot{\mathbf{x}}(t) = \mathbf{a}\left(\mathbf{x}, \mathbf{u}, t\right) \quad \mathbf{x}(t_0) = \mathbf{x}_0 \\ \text{for all} \quad & t \in [t_0, t_f] \end{aligned} \tag{5.39}$$

By applying the necessary conditions of optimality the adjoint equations that the optimal solution must satisfy are derived. These include the co-state differential equations:

$$\dot{\boldsymbol{\lambda}} = -\boldsymbol{\lambda}^T \cdot \frac{\partial\, \mathbf{a}(\mathbf{x}, \mathbf{u}, t)}{\partial\, \mathbf{x}} \tag{5.40}$$

with end conditions given by:

$$\boldsymbol{\lambda}(t_f) = \left(\frac{\partial\, S}{\partial\, \mathbf{x}}\right)_{t=t_f} \tag{5.41}$$

and finally the optimality condition:

$$\boldsymbol{\lambda}^T \cdot \frac{\partial\, \mathbf{a}(\mathbf{x}, \mathbf{u}, t)}{\partial\, \mathbf{u}} = \mathbf{0} \tag{5.42}$$

Consider now to solve (5.39) using a direct collocation method. A discretisation grid with $N$ time segments for both control history $\mathbf{u}$ and state trajectory $\mathbf{x}$ is used:

$$\boldsymbol{\Delta} = \{t_0 < t_1 < t_2 < \cdots < t_{N-1} < t_N = t_f\} \tag{5.43}$$

The nodes of the grid are evenly spaced, so that the constant length of the time segments reads:

$$\Delta = t_i - t_{i-1} \qquad i = 1, ...N \tag{5.44}$$

The notation $\mathbf{x}_i$ and $\mathbf{u}_i$ refers to the values of the state and control variables at the $i^{th}$ node respectively, and $\mathbf{a}_i$ to the evaluation of the state equations at the same node:

$$\mathbf{a}_i = \mathbf{a}(\mathbf{x}_i, \mathbf{u}_i, t_i) \tag{5.45}$$

Finally, $\mathbf{x_N}$ and $\mathbf{u_N}$ represent the set of state and control parameters respectively, and the vector of all the independent optimization variables is:

$$\mathbf{y} = \mathbf{x_N} \cup \mathbf{u_N} \tag{5.46}$$

Using the trapezoidal method for discretization, the vector of defects $\boldsymbol{\xi}_i$ at each node reads:

$$\boldsymbol{\xi}_i = \mathbf{x}_{i-1} - \mathbf{x}_i + \frac{\Delta}{2}\left[\mathbf{a}_i + \mathbf{a}_{i-1}\right] \qquad i = 1, ..., N \tag{5.47}$$

Problem (5.39) may then be converted into the following Non-Linear Programming problem:

$$\begin{aligned} \min_{\mathbf{u_n}} \quad & J(\mathbf{y}) \\ \text{subject to} \quad & \boldsymbol{\xi}_i(\mathbf{y}) = 0 \quad i = 1, ..., N \end{aligned} \tag{5.48}$$

Deriving the necessary optimality condition for problem (5.48), i.e. Karush-Kuhn-Tucker conditions, the Lagrangian function is formed:

$$\mathcal{L} = J(\mathbf{y}) + \sum_{i=1}^{N} \boldsymbol{\lambda}_i^T \boldsymbol{\xi}_i \tag{5.49}$$

The necessary condition for $\mathbf{y}$ to be a local constrained minimizer is

$$\nabla_{\mathbf{y}} \mathcal{L} = 0 \tag{5.50}$$

Since the objective $J$ depends exclusively on the final state values, this part of the necessary condition includes only the defects:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_k} = \sum_{i=1}^{N} \boldsymbol{\lambda}_i^T \frac{\partial \boldsymbol{\xi}_i}{\partial \mathbf{x}_k} = \mathbf{0} \qquad k = 1, ..., N-1 \tag{5.51}$$

But the state parameters $\mathbf{x}_k$ affect only the adjacent defects, i.e. $\boldsymbol{\xi}_i$ and $\boldsymbol{\xi}_{i+1}$ hence (5.51) reduces to:

$$\boldsymbol{\lambda}_k^T \frac{\partial \boldsymbol{\xi}_k}{\partial \mathbf{x}_k} + \boldsymbol{\lambda}_{k+1}^T \frac{\partial \boldsymbol{\xi}_{k+1}}{\partial \mathbf{x}_k} = \mathbf{0} \tag{5.52}$$

Using the definition for the defects of (5.47) in (5.52) yields:

$$\boldsymbol{\lambda}_k^T \left( -\mathbf{I} + \frac{\Delta}{2} \frac{\partial \mathbf{a}_k}{\partial \mathbf{x}_k} \right) + \boldsymbol{\lambda}_{k+1}^T \left( \mathbf{I} + \frac{\Delta}{2} \frac{\partial \mathbf{a}_k}{\partial \mathbf{x}_k} \right) = \mathbf{0} \tag{5.53}$$

where $\mathbf{I}$ is the identity matrix. Rearranging the terms in (5.53) the following is finally obtained:

$$\boldsymbol{\lambda}_k^T - \boldsymbol{\lambda}_{k+1}^T - \frac{\Delta}{2} \left( \boldsymbol{\lambda}_k^T + \boldsymbol{\lambda}_{k+1}^T \right) \frac{\partial \mathbf{a}_k}{\partial \mathbf{x}_k} = \mathbf{0} \tag{5.54}$$

Equation (5.54) is clearly a discrete version of the adjoint Equations (5.40). In fact, Equation (5.40) may be approximated over a time segment as follows:

$$\int_{t_k}^{t_{k+1}} \dot{\boldsymbol{\lambda}}\, dt = -\int_{t_k}^{t_{k+1}} \left(\frac{\partial\, \mathbf{a}(\mathbf{x},\mathbf{u},t)}{\partial\, \mathbf{x}}\right)^T \boldsymbol{\lambda}\, dt \tag{5.55}$$

Assuming that the jacobian in the right hand term of Equation (5.55) is constant over the time interval $h$, one may write:

$$\boldsymbol{\lambda}_{k+1} - \boldsymbol{\lambda}_k \cong -\left(\frac{\partial\, \mathbf{a}_k}{\partial\, \mathbf{x}_k}\right)^T \int_{t_k}^{t_{k+1}} \boldsymbol{\lambda}\, dt \cong -\left(\frac{\partial\, \mathbf{a}_k}{\partial\, \mathbf{x}_k}\right)^T \frac{\Delta}{2}\left(\boldsymbol{\lambda}_{k+1} + \boldsymbol{\lambda}_k\right) \tag{5.56}$$

Consider now the end point of the time interval. The necessary condition now reads:

$$\frac{\partial \mathcal{L}}{\partial\, \mathbf{x}_N} = \frac{\partial\, S(\mathbf{x}_N)}{\partial\, \mathbf{x}_N} + \boldsymbol{\lambda}_N^T \frac{\partial\, \boldsymbol{\xi}_N}{\partial\, \mathbf{x}_N} = \mathbf{0} \tag{5.57}$$

Proceeding as above, differentiating the defects with respect to $\mathbf{x}_N$ provides the terminal boundary conditions for Equation (5.54):

$$\boldsymbol{\lambda}_N^T - \boldsymbol{\lambda}_N^T \frac{\Delta}{2} \frac{\partial\, \mathbf{a}_N}{\partial\, \mathbf{x}_N} = \frac{\partial\, S(\mathbf{x}_N)}{\partial\, \mathbf{x}_N} \tag{5.58}$$

which corresponds to the condition in (5.41).

Finally consider the partial derivative of the Lagrangian with respect to the control variables at the interior nodes:

$$\frac{\partial \mathcal{L}}{\partial\, \mathbf{u}_k} = \sum_{i=1}^{N} \boldsymbol{\lambda}_i^T \frac{\partial\, \boldsymbol{\xi}_i}{\partial\, \mathbf{u}_k} = \mathbf{0} \qquad k = 1, ..., N-1 \tag{5.59}$$

Since the control parameters $\mathbf{u}_k$ only affect adjacent defects, Equation (5.59) simplifies to:

$$\boldsymbol{\lambda}_k^T \frac{\partial \boldsymbol{\xi}_k}{\partial \mathbf{u}_k} + \boldsymbol{\lambda}_{k+1}^T \frac{\partial \boldsymbol{\xi}_{k+1}}{\partial \mathbf{u}_k} = \mathbf{0} \tag{5.60}$$

Differentiating the defects and substituting the result in (5.60) yields:

$$\boldsymbol{\lambda}_k^T \left( \frac{\Delta}{2} \frac{\partial \mathbf{a}_k}{\partial \mathbf{u}_k} \right) + \boldsymbol{\lambda}_{k+1}^T \left( \frac{\Delta}{2} \frac{\partial \mathbf{a}_k}{\partial \mathbf{u}_k} \right) = \mathbf{0} \tag{5.61}$$

Rearranging the terms:

$$\left( \boldsymbol{\lambda}_k^T + \boldsymbol{\lambda}_{k+1}^T \right) \frac{\Delta}{2} \frac{\partial \mathbf{a}_k}{\partial \mathbf{u}_k} = \mathbf{0} \tag{5.62}$$

Equation (5.62) is a discretized version of the optimality condition (5.42). Therefore the solution to the discretized problem also satisfies the optimality principle.

## 5.3 Minimum Lap Time Problem

In this section the minimum time optimal control problem for the differential drive mobile robot is defined. The mathematical model of the system, performance criterion and the constraints are specified.

### *5.3.1 Vehicle Model*

The model used in this thesis to describe the differential drive robot is composed of the cruise control system along with the kinematics of the vehicle, as it is shown in Figure 5.3.

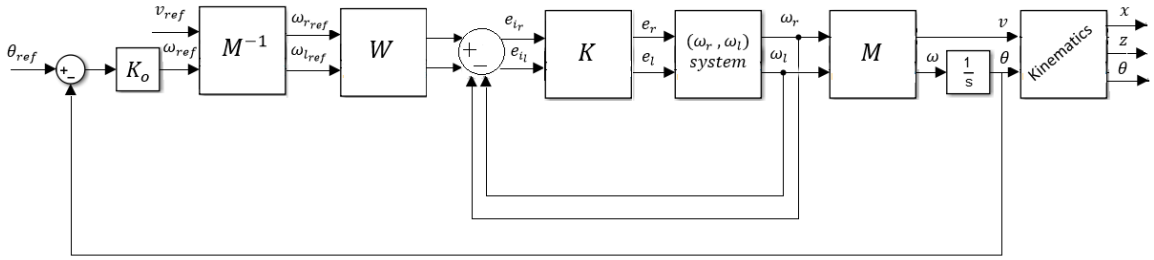


Figure 5.3: Cruise Control and Kinematics

The outer-loop controller $K_o$ is a PD controller with the following structure:

$$K_o = g\,(s+z)\left[\frac{100}{s+100}\right]^2 \tag{5.63}$$

A low frequency approximation of the cruise control system together with the kinematics of the mobile robot are used in this work in order to solve the optimal control problem. The state space representation for this low frequency approximation system is defined as follows:

$$
\begin{aligned}
\text{cruise control} \quad & \dot{x}_1 = -2.616x_1 - 2.489x_2 + 2v_{ref} \\
& \dot{x}_2 = 2x_1 \\
& \dot{x}_3 = -2.616x_3 - 2.489x_4 + 2\,e_\theta \\
& \dot{x}_4 = 2x_3 \\
& \dot{x}_5 = x_4 \\
& v = 1.245x_2 \\
& \theta = 0.6223x_4 + 1.494x_5 \\
\text{kinematics} \quad & \dot{x}_6 = \dot{x} = v\sin\theta = (1.245x_2)\sin(0.6223x_4 + 1.494x_5) \\
& \dot{x}_7 = \dot{z} = v\cos\theta = (1.245x_2)\cos(0.6223x_4 + 1.494x_5)
\end{aligned}
\tag{5.64}
$$

Or in a more compact form

$$
\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \quad \mathbf{x}(0) = \mathbf{x_0} \quad t \in [0, T]
\tag{5.65}
$$

### *5.3.2 Track*

A race track is designed such that the differential-drive mobile robot can traverse on it and it is described by the parameters $(x_t, z_t, \theta_t)$. The angle of the track tangent is described by $\theta_t$, $(x_t, z_t)$ are the coordinates of the line. All these parameters are functions of $s$. This is useful since it makes it possible to write the entire track as a function independent of time [24].

Figure 5.4 shows the track used in this thesis.

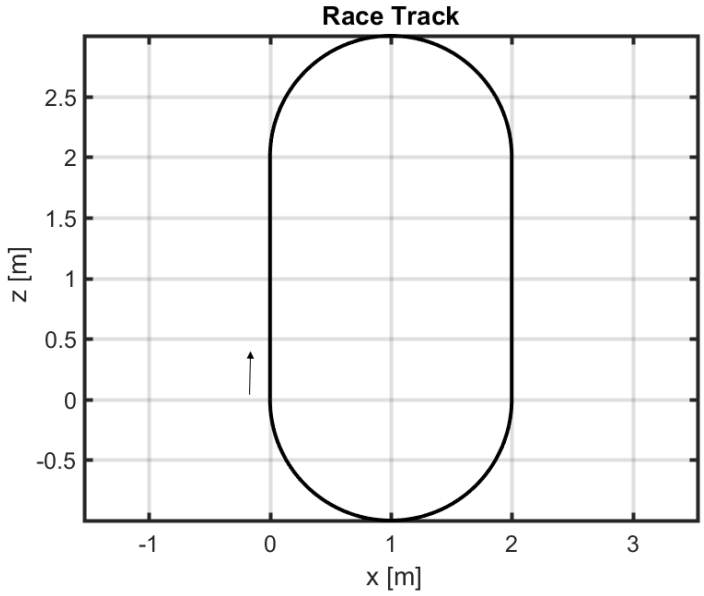


Figure 5.4: Oval Race Track

For this specific track the parameters $(x_t, z_t, \theta_t)$ with respect to the travelled distance $s$ are plotted in Figure 5.5.

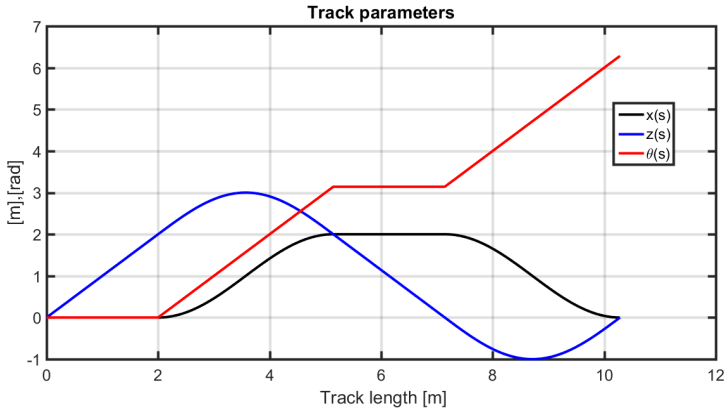


Figure 5.5: Oval Track Parameters

The MATLAB code used to generate the racetrack data can be found Appendix A on page 178.

### 5.3.3 *Physical Constraints*

As it was mentioned earlier, physical constraints in an optimal control problem are intended to limit the range of the state and control variables within values which are meaningful for the plant and for the problem which is being analyzed.

Given the position of the car in the absolute reference axis system fixed in space $(x, z)$ and the coordinate $(x_t, z_t)$ and orientation $\theta_t$ of the corresponding point on the track, the distance $d$ between the car and the track, shown in Figure 5.6 is calculated as follows:

$$d = (x - x_t)\cos\theta_t - (z - z_t)\sin\theta_t \tag{5.66}$$

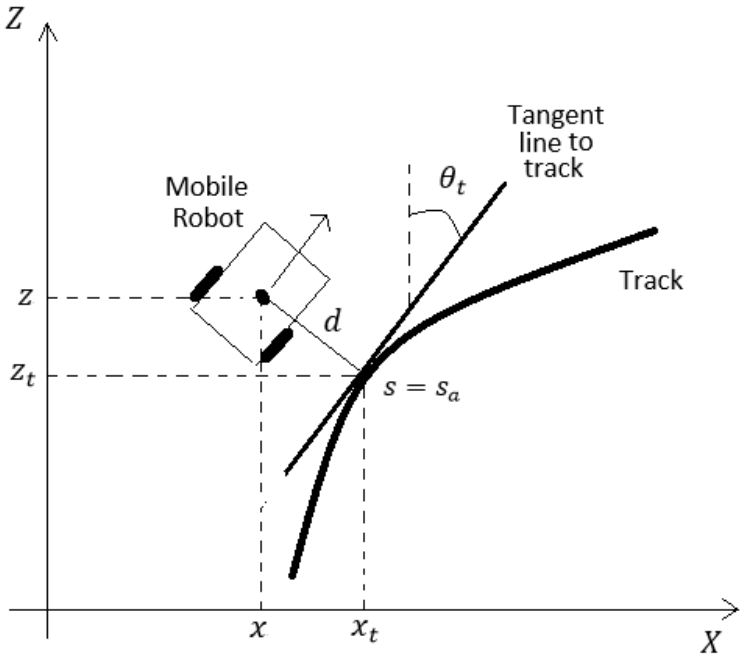


Figure 5.6: Distance Between Mobile Robot and Track

Note that $d$ is positive when the mobile robot is to the right of the track and negative when it is to the left of the track.

For implementing a camera-based outer loop control system shown in Figure 5.3 the Raspberry Pi Camera was used. This camera has an horizontal Field of View of 53.5°. The camera is looking ahead of the mobile robot $l = 20$cm and the camera is placed approximately 10 cm ahead of mobile robot center of gravity (see Figure 5.7). Looking ahead 20 cm implies that the robot's camera will be able to see approximately $W = 20.161$ cm horizontally.

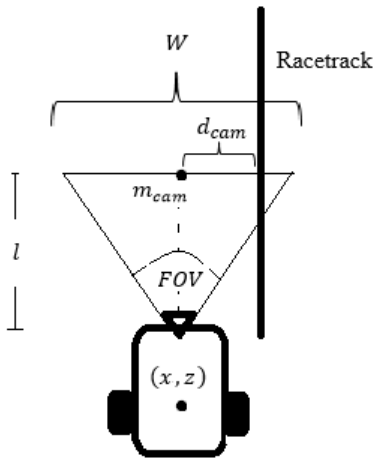


Figure 5.7: Field of View Constraint

Hence a constraint that the race track is within the FOV of the camera is made as follows:

$$d_{cam} = \frac{(x_{ts} - x_{cam})\cos\theta_{ts} + (z_{cam} - z_{ts})\sin\theta_{ts}}{\cos(\theta_{ts} - \theta)} \tag{5.67}$$

$$-\frac{20.161}{2} <= d_{cam} <= \frac{20.161}{2} \tag{5.68}$$

where $m_{cam} = (x_{cam}, z_{cam})$ is the position of the middle point of $W$, and $x_{ts}(s) = x_t(s{+}0.3)$ , $z_{ts}(s) = z_t(s{+}0.3)$, $\theta_{ts}(s) = \theta_t(s{+}0.3)$ are shifted parameters of the track[1].

In the camera-based solution, $e_\theta$ information is obtained from the camera as it is shown in Figure 5.8. For the noncamera-based solution $e_\theta$ is obtained by directly computing $\theta_{ref} - \theta$.

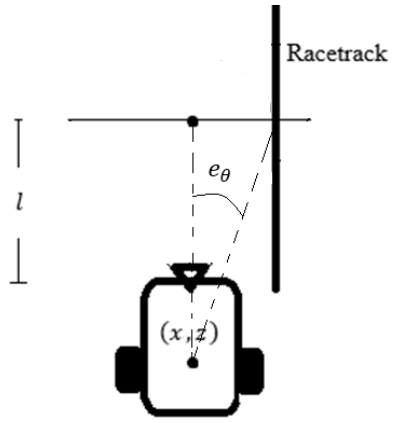


Figure 5.8: Computation of $e_\theta$

The maximum observed speed achieved by the Enhanced Thunder Tumbler is $2.3m/s$, and the maximum observed acceleration is about $1m/s^2$. In this work the constraints on the commanded and actual velocities and accelerations of the car are of the form

$$\begin{aligned} 0 <= v_{ref} <= 0.5 \\ 0 <= v <= 2.3 \\ -1 <= \dot{v}_{ref} <= 1 \\ -1 <= \dot{v} <= 1 \end{aligned} \tag{5.69}$$

[1] $x_t, z_t, \theta_t$ are all distance-dependent variables and $s$ is the independent variable in meters.

Also imposing constraint on the jerk, i.e $\ddot{v}_{ref}$ and $\ddot{v}$ makes the resulting motion of the robot smooth. This constraint is given as

$$\begin{aligned} -1 <= \ddot{v}_{ref} <= 1 \\ -1 <= \ddot{v} <= 1 \end{aligned} \tag{5.70}$$

Two constraints are placed on the maximum angular velocity of the wheels as follows:

$$\begin{aligned} 0 <= \omega_R <= 46 \\ 0 <= \omega_L <= 46 \end{aligned} \tag{5.71}$$

For the noncamera-based solution, extra constraints are placed on first and second derivatives of orientation of the vehicle:

$$\begin{aligned} -10 <= \dot{\theta}_{ref} <= 10 \\ -1 <= \ddot{\theta}_{ref} <= 1 \\ -1 <= \ddot{\theta} <= 1 \end{aligned} \tag{5.72}$$

The initial conditions used in this thesis (for both camera and noncamera-based methods) are defined as follows:

$$
\begin{aligned}
x_1(0) &= 0 \\
x_2(0) &= 0.008 \\
x_3(0) &= 0 \\
x_4(0) &= 0 \\
x_5(0) &= 0 \\
x_6(0) &= 0 \\
x_7(0) &= 0
\end{aligned}
\tag{5.73}
$$

or in a compact form as

$$
\mathbf{x}(0) = \mathbf{x_0} \quad s \in [0, S] \tag{5.74}
$$

### *5.3.4 The Performance Measure*

The goal in this thesis is to minimize the time it takes for the robot to traverse a given track, however with a general formulation as given in Equation 5.6, that is not possible since the lap time, or in other words, the final time $t_f$ is unknown [24].

Distance $s$ will be used as independent variable instead of the time $t$. This distance is a natural choice because it makes it easier to parameterize the track and the final distance $s_f$ is then a known constant, simply the track length.

For this purpose the time to distance scaling factor $\alpha$ shall be used. The task is to express the increment $ds$ of the distance travelled along a reference line corresponding to the increment $dt$. If the vehicle trajectory coincided exactly with the ideal path (race track), the scaling factor would simply read:

$$
\alpha = \frac{dt}{ds} = \frac{1}{v} \tag{5.75}
$$

where $v$ is the linear velocity of the mobile robot. However the trajectory of the mobile robot might not coincide with this path, hence a different scale factor, taken from [23], is used [2]

$$\alpha = \frac{dt}{ds} = \frac{1 - dk_t}{v\cos(\theta - \theta_t)} \tag{5.76}$$

where $\theta$ is the orientation of the vehicle, $\theta_t$ is the angle of the race track tangent, $d$ is the distance between the car and the race track, $k$ is the curvature of the race track.

Given this, the mathematical model given in (5.65) now becomes

$$\frac{d\mathbf{x}}{ds} = \alpha\frac{d\mathbf{x}}{dt} = \alpha\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) = \bar{\mathbf{f}}(\mathbf{x}(s), \mathbf{u}(s)) \quad \mathbf{x}(0) = \mathbf{x_0} \quad s \in [0, S] \tag{5.77}$$

where $S$ is the length of the track to be traversed, which is therefore fixed.

The performance measure is the time that the differential drive mobile robot takes to traverse the given track. The time to distance scaling factor offers a straightforward way to evaluate the maneuver time. One more state variable $x_{p+1}$ can be added which satisfies:

$$\dot{x}_{p+1}(s) = \alpha(s) \quad x_{p+1}(0) = 0 \quad s \in [0, S] \tag{5.78}$$

According to the definition of the scaling factor given in (5.76) the added state variable represents the time elapsed from the beginning of the maneuver. Thus, the performance measure simply reads

$$J = x_{p+1}(S) \tag{5.79}$$

[2] assuming lateral velocity of mobile robot is zero.

Consider Figure 5.9 to better understand the scaling factor $\alpha$.

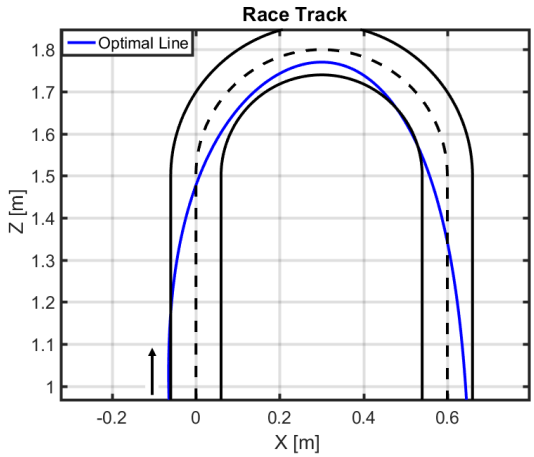


Figure 5.9: Scaling Factor Impact on Optimal Line

Here a clockwise direction is assumed. For the right hairpin turn the curvature $k_t$ is positive. When the vehicle takes an inner line, the distance $d$, expressed in Equation 5.66, is positive and the numerator in Equation 5.76 is smaller than one. If the vehicle takes instead an outer line, $d$ becomes negative and the numerator in Equation 5.76 is greater that one. Therefore the scaling factor is greater than in the previous case. When the vehicle is travelling along an outer line the greater scaling factor indicates that the vehicle takes more time to traverse the same road distance compared with the case of the vehicle taking the inner line [23].

### 5.3.5 Problem Statement

Given the vehicle model, physical constraints and the performance measure above, the minimum time optimal control problem is now defined.

$$\begin{aligned} \min_{\mathbf{u}(s)} \quad & J = x_{p+1}(S) \\ \text{subject to:} \quad & \frac{d\mathbf{x}}{ds} = \bar{\mathbf{f}}(\mathbf{x}(s), \mathbf{u}(s)) \quad \mathbf{x}(0) = \mathbf{x_0} \\ & c_i \leq 0 \quad \forall i \\ & \mathbf{u}_L \leq \mathbf{u}(s) \leq \mathbf{u}_U \\ \text{for all} \quad & s \in [0, S] \end{aligned} \tag{5.80}$$

As it was stated in above, there are two distinct classes of methods for solving an optimal control problem, namely direct and indirect methods. Indirect methods rely on the application of the theory of calculus of variations, i.e. the Pontryagin's Minimum Principle. Direct methods, instead, aim to solve the problem by converting the original continuous problem into a discretised problem and applying mathematical programming techniques.

One aspect in favour of direct methods is the greater freedom in defining and including state and control constraints in the optimization problem compared with indirect methods. The solution of constrained optimization problems by indirect methods either requires the use of specially designed algorithms or the use of penalty functions to include the constraints in the objective function. Also with direct methods the discretization of the control history allows to model any type of control law straightforwardly [23].

The basic concept of direct methods is that the continuous control history is replaced with a discrete approximation. It is assumed that the control input can only be adjusted at a number of fixed positions along the trajectory.

The disadvantage with direct methods is that because of their discrete nature they only return approximate solutions.

In this thesis, a direct method is used to solve the minimum time optimal control problem (direct collocation). Since both the track and the vehicle model are specified as functions of the variable $s$, the problem is discretized by dividing the track in $m$ points. The length of the track is 10.282 meters and the step size used in this thesis is 0.001; this means that $m = 10283$.

Discretizing the control history is done first. In general the model that is used here has the following input $\mathbf{u} = \begin{bmatrix} v_{ref} \\ \\ \theta_{ref} \end{bmatrix} = \begin{bmatrix} u_1 \\ \\ u_2 \end{bmatrix}$ but since the controls are being discretized in $m$ points, the following is obtained:

$$\mathbf{u_m} = \{\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_{m-2}, \mathbf{u}_{m-1}\} \tag{5.81}$$

The above means that

$$\begin{aligned} \mathbf{u}_1 &= \{u_{10}, u_{11}, ..., u_{1m-2}, u_{1m-1}\} \\ \mathbf{u}_2 &= \{u_{20}, u_{21}, ..., u_{2m-2}, u_{2m-1}\} \end{aligned} \tag{5.82}$$

The dimension of the control array is given by $\mathbf{u}_1 \cup \mathbf{u}_2$, which is $2m$.

Next, the state trajectory is divided in $m$ segments as well.

$$\mathbf{x_m} = \{\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{m-2}, \mathbf{x}_{m-1}\} \tag{5.83}$$

If there are $p$ state variables, the vector $\mathbf{x_m}$ will have $p \times m$ elements. Finally, the control parameters are combined with the state parameters to define the vector $y$ of all the independent optimization variables, whose dimension will be $2m + p \times m$:

$$\mathbf{y} = \mathbf{u_m} \cup \mathbf{x_m} \tag{5.84}$$

Since the objective and constraints functions can be expressed directly as functions of the independent optimization variables $\mathbf{y}$, the original minimum time optimal control problem discussed in 5.3.4 may be converted into the following Non-Linear Programming problem:

$$\begin{aligned} \min_{y} \quad & J(\mathbf{y}) \\ \text{subject to:} \quad & c_i(\mathbf{y}) = 0 \quad i \in E \\ & c_i(\mathbf{y}) \leq 0 \quad i \in I \\ & \mathbf{y}_L \leq \mathbf{y} \leq \mathbf{y}_U \end{aligned} \tag{5.85}$$

where $E$ and $I$ represent the set of equality and inequality constraints respectively. The performance measure $J(\mathbf{y})$ is often referred to in the context of numerical optimization simply as the *objective function*.

Since the minimum time optimal control problem has been discretized, the finite dimensional nonlinear programming problem can be solved by a NLP solver, in this case Knitro solver.

Before presenting simulation results, a description of the tools used to solve the minimum time optimal control problem, namely Knitro and Advance Mathematical Programming Language (AMPL) are presented.

**KNITRO Solver.** KNITRO, short for Nonlinear Interior point Trust Region Optimization is a solver for nonlinear optimization problems. Knitro is a package for solving nonlinear optimization problems. It is designed for large-scale applications, but it is also effective on small and medium scale problems [34]. The solver implements state-of-the-art interior-point and active-set methods for solving problems [43].

**AMPL.** Developed in Bell Laboratories, AMPL is a comprehensive and powerful algebraic modeling language for linear and nonlinear, continuous or discrete system

optimization problems [36] [41]. It is user friendly, making the user focus on the modeling of the problem, not the technical details for programming. All the variables, parameters, cost functions, constraint functions are defined intuitively and straightforward. The main difference between AMPL with other programming languages such as C or Fortran are the expressions of the variables. In AMPL, "set" and "index" are used to invoke the specific variable. On the other hand, the mathematical expression is generally adapted from an advanced programming language, e.g. "sum" or ">" and so on as arithmetic or logical operators are used .

In this thesis AMPL is used to write (model) the performance measure, vehicle dynamics and physical constraints as it was discussed in subsection 5.3.4.

The AMPL system supports the entire optimization modeling lifecycle  formulation, testing, deployment, and maintenance  in an integrated way promotes rapid development and reliable results. AMPL integrates a modeling language for describing optimization data, variables, objectives, and constraints; a command language for debugging models and analyzing results; and a scripting language for manipulating data and implementing optimization strategies [42].

Once the AMPL model is finished, solution to this problem is obtained by interfacing this AMPL model with KNITRO solver.

The solver can be found under the NEOS server which is a free internet-based service for solving numerical optimization problems [37], [38], [39], [40].

## 5.4 Camera Based Solution

In this section simulation and experimental results for a camera based solution to the minimum time problem are presented.

### *5.4.1 Simulation Results*

For the camera based solution an outer loop controller with roll-off and the following gains was used: $k_p = 1.2$, and $k_d = 0.5$. The resulting minimum time was found to be 25.0161 seconds. In Figure 5.10 it is shown the race track used in this thesis, along with the resulted path obtained from KNITRO solver in the NEOS server.

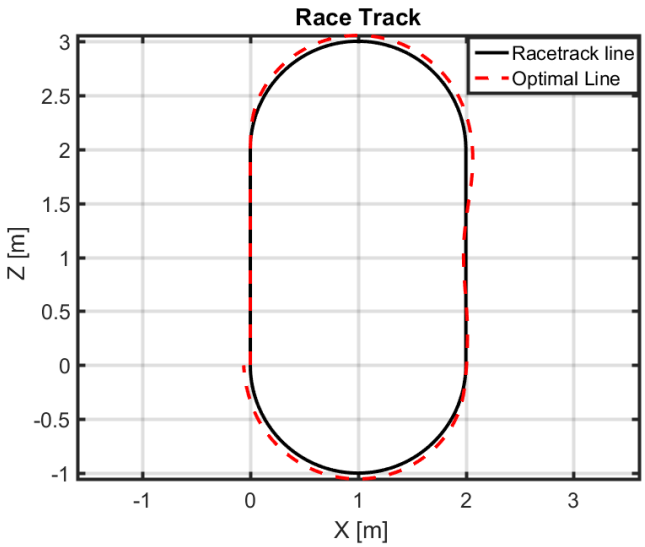


Figure 5.10: Camera Based Optimal Line - Simulation

Figure 5.11 shows the optimal $v_{ref}$ command. Also the actual or the achieved velocity of the robot is plotted. The commanded velocity decreases for taking the turn and increases again once the robot is on the straight segment.

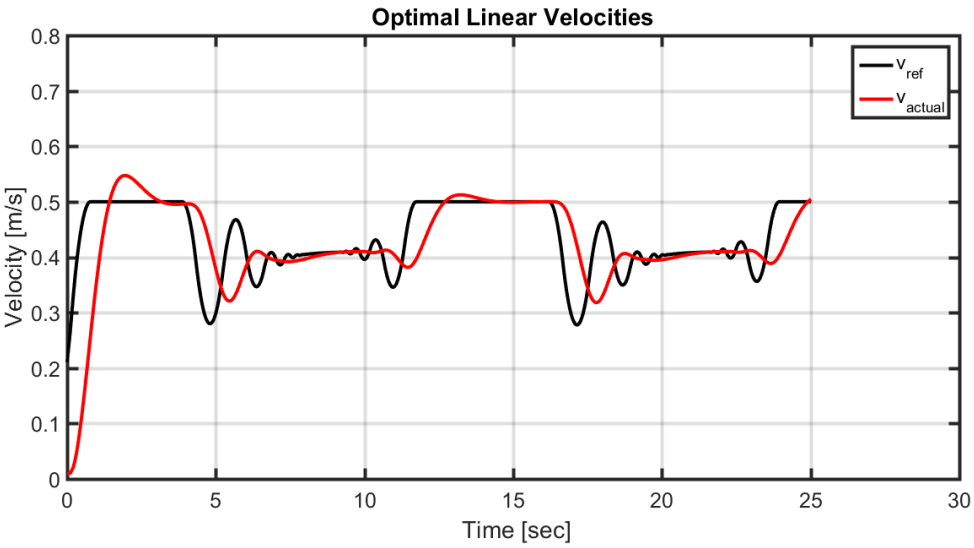


Figure 5.11: Camera Based Optimal Linear Velocities - Simulation

The commanded and actual orientation of the car are shown in Figure 5.12.

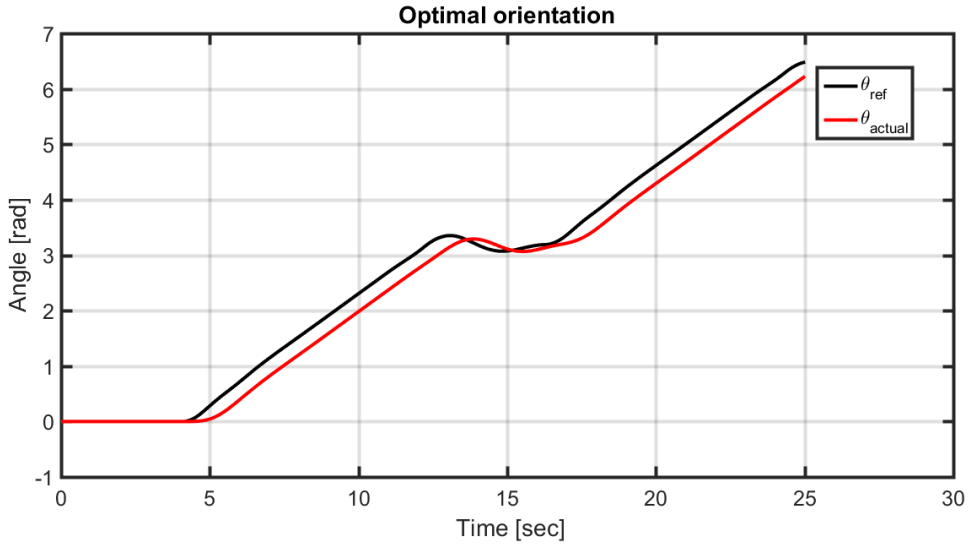


Figure 5.12: Camera Based Orientation - Simulation

The control effort for both the left and right wheels is shown in Figure 5.13.

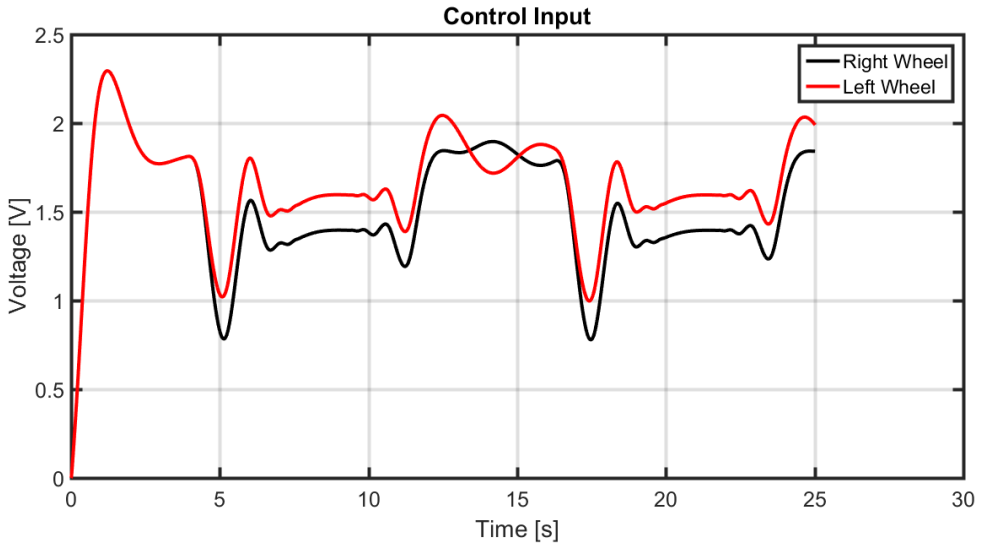


Figure 5.13: Camera Based Control Input - Simulation

The Field-of-View (FOV) of a camera is an important parameter to consider in the solution of the minimum time problem. Figure 5.14 shows the impact that the FOV of the camera has in the resulting minimum time for the given racetrack.

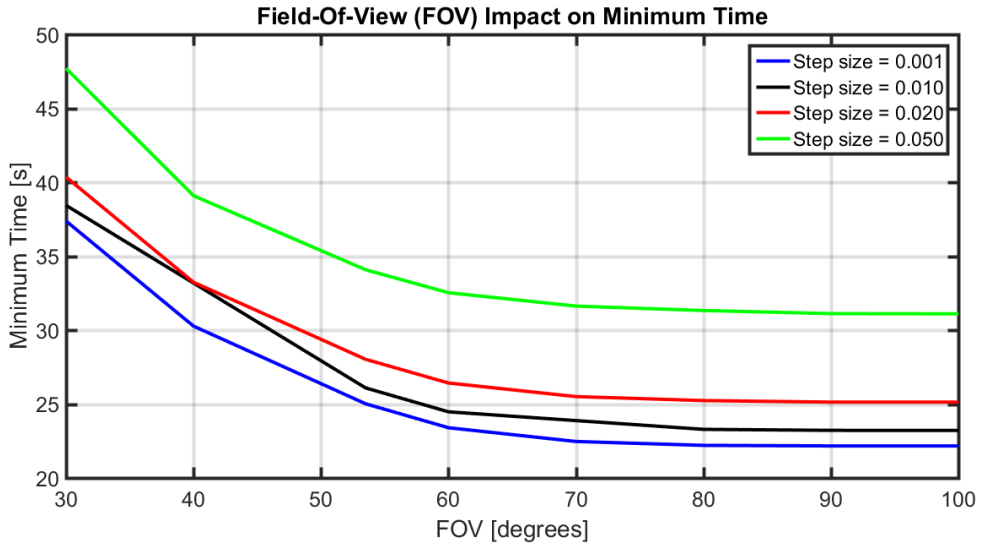


Figure 5.14: Field Of View Impact on Minimum Time

As the FOV of the camera increases, the optimal time to traverse the racetrack decreases. Note also that if a bigger (integration) step size is used, minimum time increases.

**CODE: AMPL.** The AMPL code used to obtain the results shown above can be found within Appendix D on page 225.

### *5.4.2 Experimental Results*

In Figure 5.15 it is shown the resulted path when using the Raspberry Pi camera. Note that the path starts to drift away from the racetrack, this is because of dead reckoning errors in the computation of the $(x, z)$ position from the velocity and orientation measurements. The time it takes the robot to complete the racetrack is around 41 seconds.

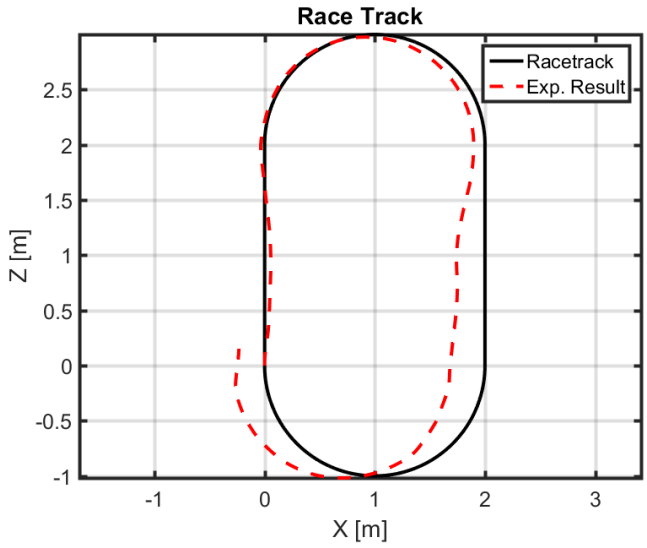


Figure 5.15: Camera Based Optimal Line - Experimental

Figure 5.16 shows the commanded and actual linear velocities of the mobile robot. The encoder resolution here is approximately 0.098 m/s. The commanded velocity

profile is different from the one obtained in simulation because as one increases the commanded speed, the robot starts to miss the track.

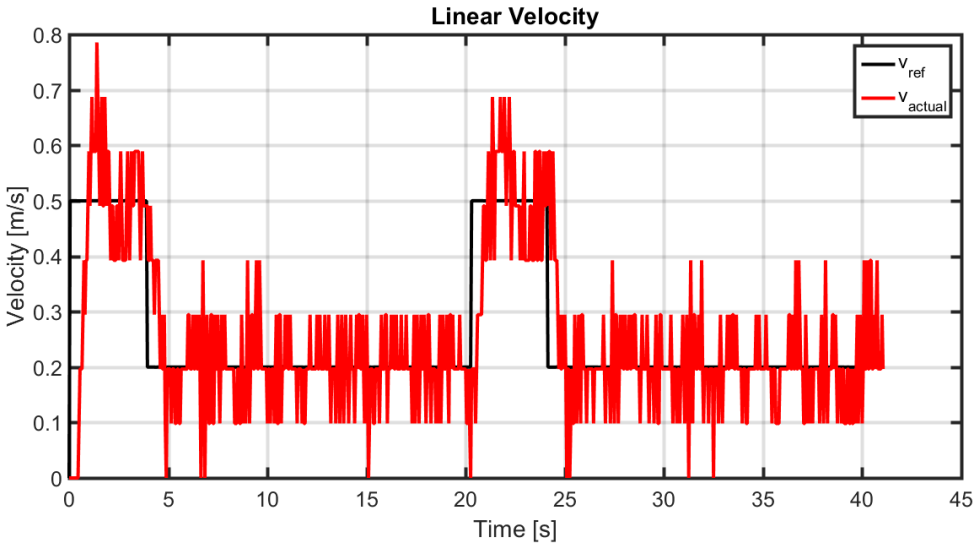


Figure 5.16: Camera Based Optimal Linear Velocities - Experimental

Figure 5.17 shows the reference and actual orientation for the mobile robot.

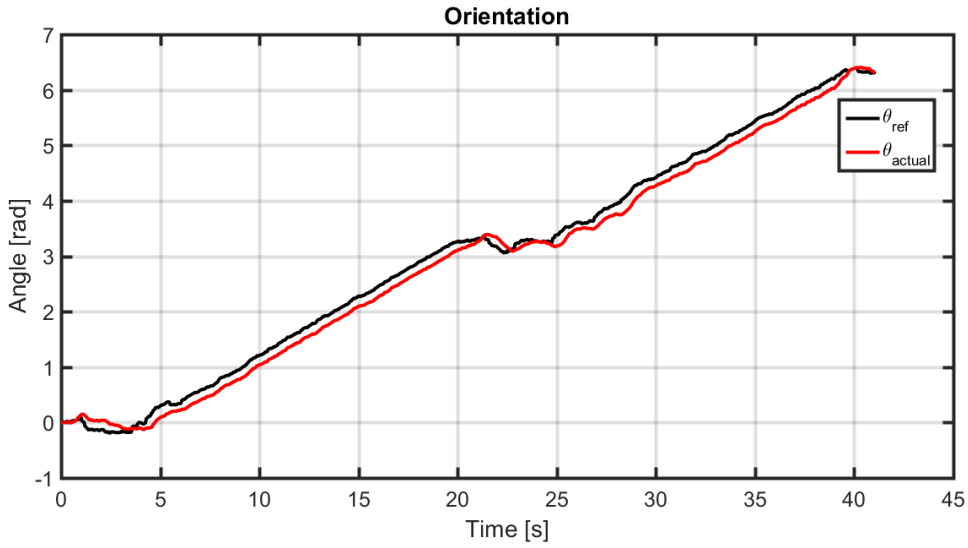


Figure 5.17: Camera Based Orientation - Experimental

The control effort for both the left and right wheels is shown in Figure 5.18.

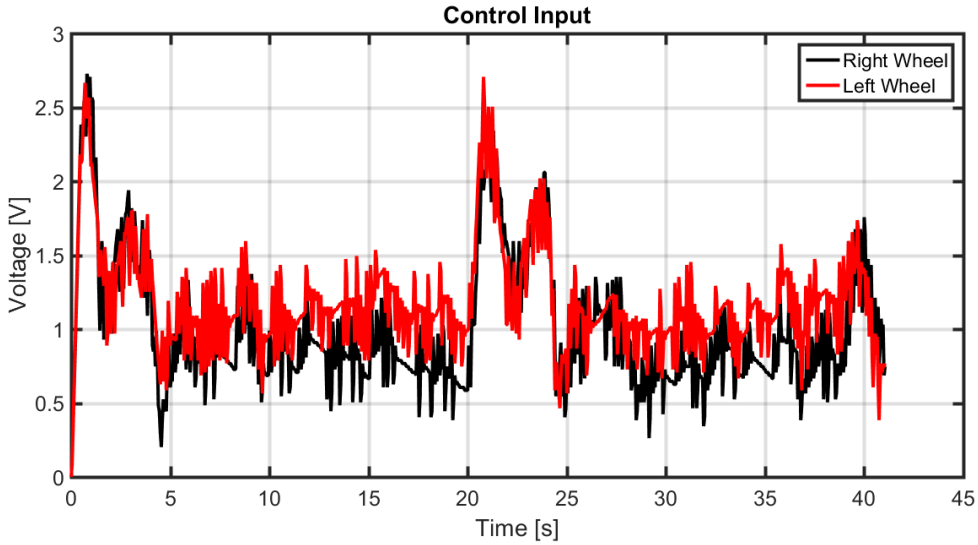


Figure 5.18: Camera Based Control Input - Experimental

Figure 5.19 shows the effect of increasing the commanded speed in the tracking of the racetrack by the mobile robot.

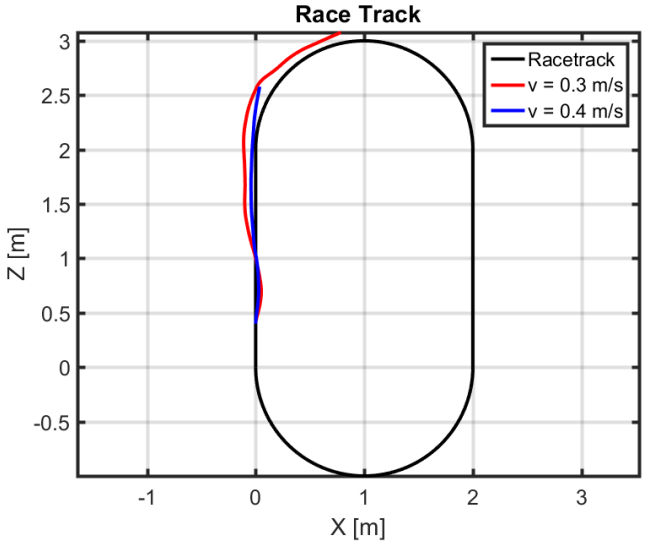


Figure 5.19: Resulting Path as Speed Increases

As the speed increases the mobile robot starts to lose the racetrack.

The difference between simulation and experimental results for the camera based solution is now explained. The sampling time $T_s$ in the experimental results is around 0.125 sec, and so the time difference between samples is 0.125 sec and the distance difference between samples $d_s$ can be calculated with $d_s = v\,T_s$, where $v$ is the linear velocity of the robot. Since $v$ is not constant(from 0.2 to 0.5 m/s), $d_s$ varies from around 0.025 m to 0.0625 m. Within this thesis the $d_s$ used in simulation is the same as the step size ($0.001 = 1mm$). Further investigation shall be done in the future to increase the simulated $d_s$ while maintaining a small step size (to preserve accuracy in the solution of the differential equations).

**CODE: PYTHON AND ARDUINO.** The Python and Arduino code used to obtain the experimental results presented above can be found within Appendix C on page 219 and Appendix B on page 191. The Python code shown in Appendix C on page 222 was used on the Raspberry Pi 2 to receive data from Arduino (e.g. wheel speeds, control action, among others).

## 5.5 Noncamera Based Solution

Simulation and experimental results are presented next for a non-camera based solution to the minimum time problem. Here an IMU was used to estimate the orientation of the car and form the $(v, \theta)$ cruise control system.

### *5.5.1 Simulation Results*

For the noncamera based solution an outer loop controller with roll-off and the following gains was used: $k_p = 0.869$, and $k_d = 0.396$. Since a $(v, \theta)$ cruise control system is used, one can obtain the optimal inputs $(v_{ref}, \theta_{ref})$ that will make the car traverse along the racetrack in an optimal way.

The state space representation (low frequency) for the outer loop cruise control system shown in Figure 5.3 for a noncamera-based method is presented next:

$$
\begin{aligned}
\text{cruise control} \quad & \dot{x}_1 = -2.616x_1 - 2.489x_2 + 2v_{ref} \\
& \dot{x}_2 = 2x_1 \\
& \dot{x}_3 = -2.556x_3 - 1.703x_4 - 1.08x_5 + \theta_{ref} \\
& \dot{x}_4 = 4x_3 \\
& \dot{x}_5 = x_4 \\
& v = 1.245x_2 \\
& \theta = 0.4924x_4 + 1.08x_5 \\
\text{kinematics} \quad & \dot{x}_6 = \dot{x} = v\sin\theta = (1.245x_2)\sin(0.4924x_4 + 1.08x_5) \\
& \dot{x}_7 = \dot{z} = v\cos\theta = (1.245x_2)\cos(0.4924x_4 + 1.08x_5)
\end{aligned}
\tag{5.86}
$$

Figure 5.20 shows the resulted path obtained from KNITRO solver in the NEOS server when no camera is used, i.e. the camera constraint explained above is removed

(a constraint for the robot to lie within +/- 1 cm from the racetrack is imposed instead). The resulting minimum time was found to be 21.14 seconds.

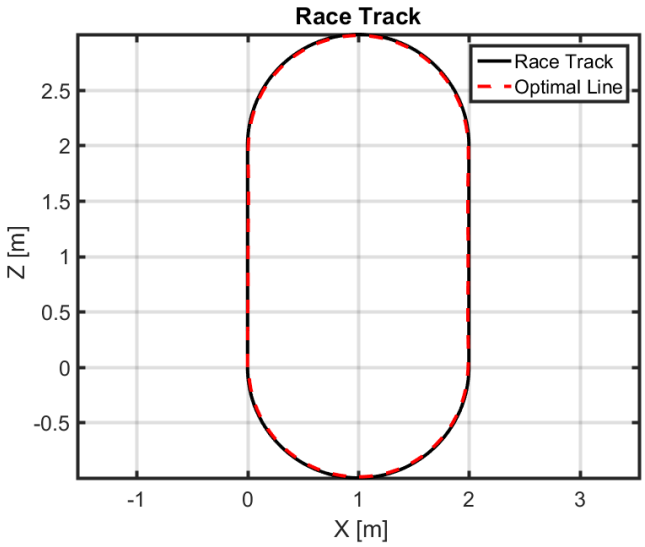


Figure 5.20: Noncamera Based Optimal Line -Simulation

The optimal linear velocities, commanded and actual, are shown in Figure 5.21. In contrast to the camera-based solution shown above, the commanded linear velocity is constant for almost the entire time with a value of $0.5\frac{m}{s}$.

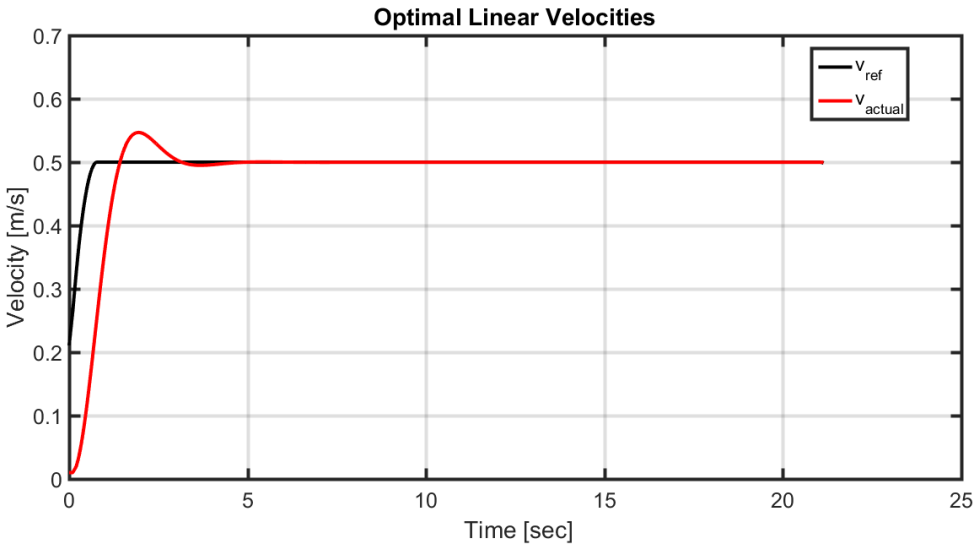


Figure 5.21: Noncamera Based Optimal Linear Velocities - Simulation

Figure 5.22 shows the commanded orientation $\theta_{ref}$ and the actual orientation of the mobile robot $\theta$.

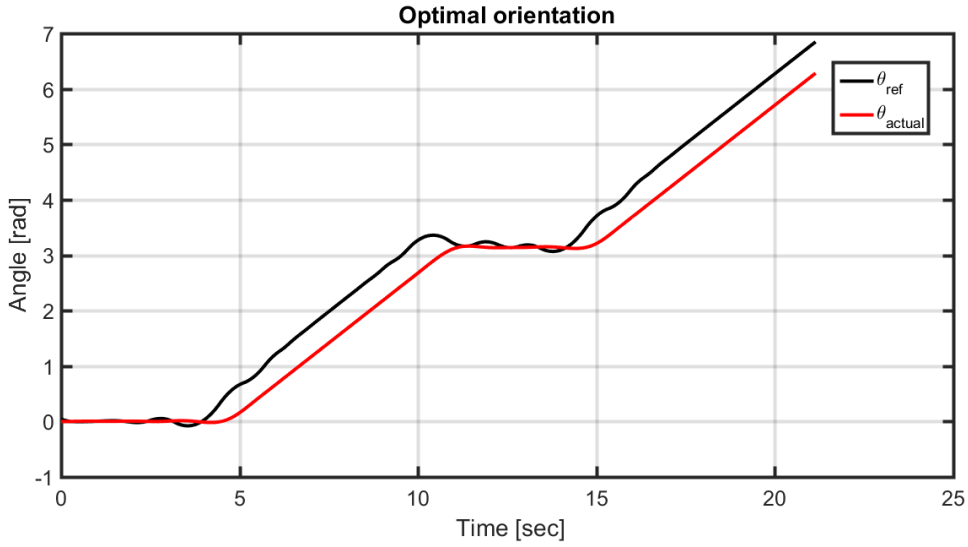


Figure 5.22: Noncamera Based Optimal Orientation - Simulation

The control input signals obtained in simulation for both the left and right wheels are shown in Figure 5.23.

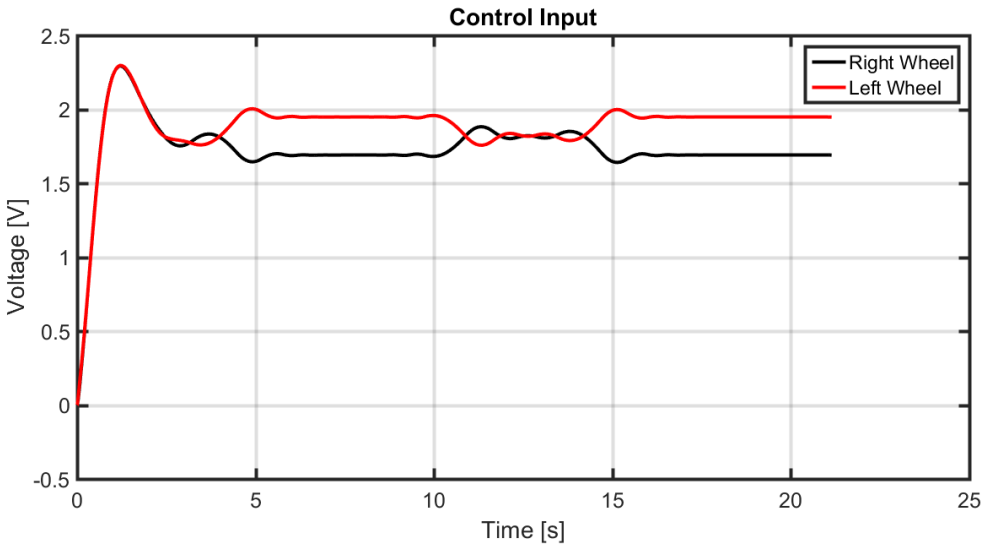


Figure 5.23: Noncamera Based Control Input - Simulation

**CODE: AMPL.** The AMPL code used to obtain the results shown above can be found within Appendix D on page 227.

### *5.5.2 Experimental Results*

In Figure 5.24 it is shown the resulted path when using an IMU (BNO055) to estimate the robot's orientation $\theta$. Also there is drift because of dead reckoning errors in the computation of position $(x, z)$. The time it takes the robot to traverse the racetrack is around 21.3 seconds.

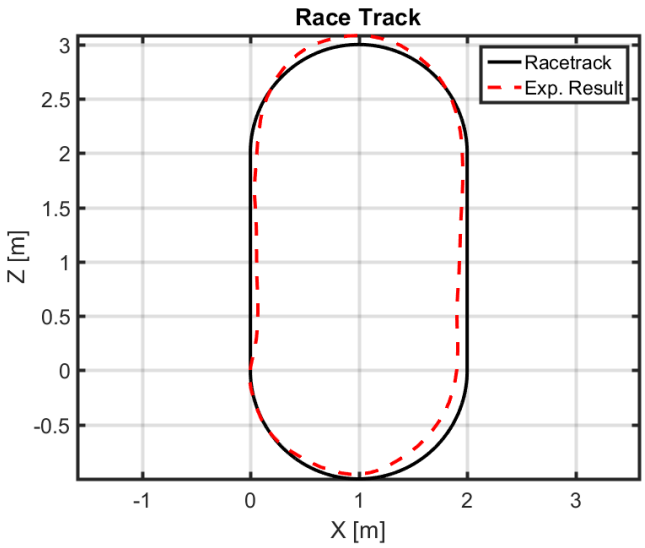


Figure 5.24: Noncamera Based Optimal Line - Experimental

Figure 5.25 shows the commanded and actual linear velocities of the mobile robot. Here the encoder resolution is around 0.049 m/s.

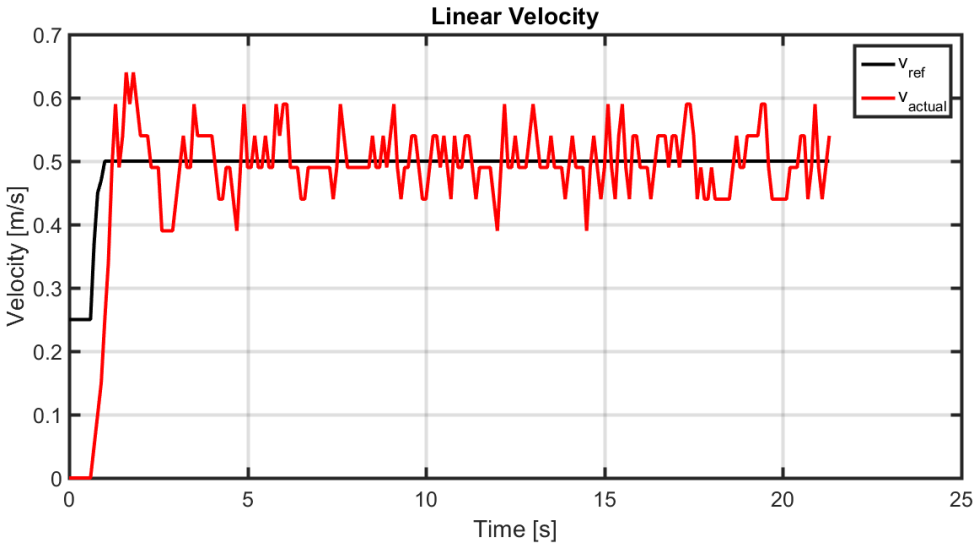


Figure 5.25: Noncamera Based Optimal Linear Velocities - Experimental

Figure 5.26 shows the commanded orientation $\theta_{ref}$ and the actual orientation $\theta$. The Gyroscope within the IMU has a resolution of 0.001 rad/s (with a range of +/-

34.9 rad/s, 16 bits resolution and 32 Hz bandwidth).

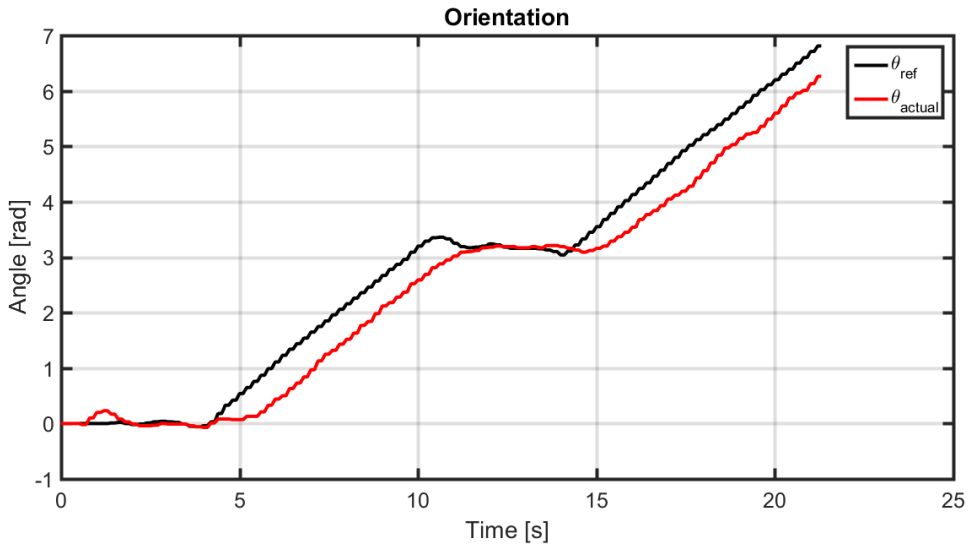


Figure 5.26: Noncamera Based Optimal Orientation - Experimental

Control input signals for both the left and right wheels are shown in Figure 5.27.

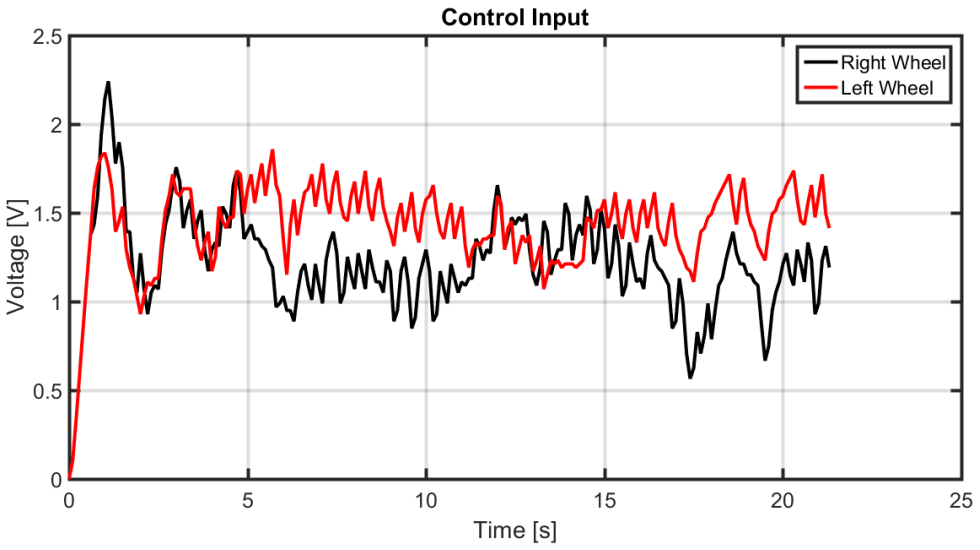


Figure 5.27: Noncamera Based Control Input - Experimental

Polynomial fitting was used to generate the commanded linear velocity $v_{ref}$ and orientation $\theta_{ref}$ in Arduino. Once the simulated $v_{ref}$ and $\theta_{ref}$ were obtained, poly-

nomial fitting was used in MATLAB to obtain the required coefficients to generate such signals. Later those coefficients were used in the Arduino program.

For the noncamera-based method, due to dead reckoning errors, the mobile robot does not follow the racetrack precisely as compared with the simulation results shown above.

**CODE: ARDUINO.** The Arduino code used to obtain the experimental results presented above can be found within Appendix B on page 195.

## 5.6 Summary and Conclusions

This chapter has discussed the optimal control minimum time problem for the differential-drive Thunder Tumbler going around a race track. Relevant theory was first presented to have a basic understanding of the problem at hand. The different components that make up the optimal control problem were stated and defined for the ground mobile robot. With the camera-based solution, the mobile robot is slow but stays on the track. Non-camera based solution makes the robot go faster but it is not as precise as with the camera-based method due to dead reckoning errors. Field-of-View of the camera, as shown above, reduces the minimum time for the camera-based method. Simulation and experimental data were presented.

# Chapter 6

# SUMMARY AND FUTURE DIRECTIONS

## 6.1 Summary of Work

This thesis addressed control issues that are important to achieve the longer-term *FAME* objective. The following summarizes key themes within the thesis.

1. **Literature Survey.** A fairly comprehensive literature survey of relevant work was presented.

2. **FAME Architecture.** A general *FAME* architecture has been described.

3. **Modeling.** Kinematic and dynamic models for differential-drive mobile ground robot were presented.

4. **Control.** Position Based and Image Based robot control methodologies were presented for the differential-drive ground mobile robot. A camera-based and a noncamera-based $(v, \theta)$ cruise control systems were used for the solution of the minimum time optimal control problem. to obtain a solution to the minimum time optimal control problem.

5. **Hardware Demonstrations.** Hardware demonstrations were conducted - with simulation data corroborating the experimental results.

## 6.2 Directions for Future Research

Future work will involve each of the following:

- **Onboard Sensing.** Addition of multiple onboard sensors; e.g. additional ultrasonics, cameras, lidar, GPS, etc.

- **Multi-Vehicle Cooperation.** Cooperation between ground, air, and sea vehicles - including quadrotors, micro-air vehicles and nano-air vehicles.

- **Parallel Onboard Computing.** Use of multiple processors on a robot for computationally intense work; e.g. onboard optimization and decision making.

- **Modeling and Control.** More accurate dynamic models and control laws. This can include the development of multi-rate control laws that can significantly lower sampling requirements.

- **Control-Centric Vehicle Design.** Understanding when simple control laws are possible and when complex control laws are essential. This includes knowing how control-relevant specifications impact (or can drive) the design of a vehicle.

- **Reconstruction of a racetrack.** Mapping of any racetrack online, i.e. obtaining the racetrack parameters $(x_t, z_t, \theta_t)$ as robot travels through it.

- **Use of different NLP solvers** Comparison of the optimal results for different available NLP solvers, such as IPOPT, LOQO, SNOPT.

- **Different discretization schemes** Investigate the effect on the solution to the optimal control minimum time problem by using various discretization techniques.

- **Step size study** Investigate how the optimal solution changes as the step size is varied along the racetrack.

REFERENCES


[1] R. W. Brockett, "Asymptotic Stability and Feedback Stabilization," in R. W. Brockett, R. S. Millman, and H. J. Sussmann, editors, *Differential Geometric Control Theory*, Birkhauser, Boston, MA, 1983

[2] A. Astolfi, "On the Stabilization of Nonholonomic Systems," *Proceedings of the 33rd IEEE Conference on Decision and Control*, vol.4, no., pp. 3481-3486, vol.4, 14-16 Dec 1994.

[3] J.K. Hedrick and A. Girard, *Control of Nonlinear Dynamic Systems: Theory and Applications*, 2005.

[4] F.C. Vieira, A.A.D. Medeiros, P.J. Alsina, A.P. Araujo, "Position and Orientation Control of a Two-Wheeled Differentially Driven Nonholonomic Mobile Robot," *ICINCO Proceedings*, 7 pages, 2004.

[5] R. Dhaouadi and A.A. Hatab, "Dynamic Modeling of Differential-Drive Mobile Robots using Lagrange and Newton-Euler Methodologies: A Unified Framework," *Adv Robot Autom* 2.107, 2013.

[6] P.I. Corke, "Robotics, Vision and Control," *Springer*, 2011.

[7] J. Gao and Y. Zhang, "An Improved Iterative Solution to the PnP Problem," *International Conference on Virtual Reality and Visualization*, 2013.

[8] M. Ito, T. Ebata and M. Shibata, "Vision-based Switching Control Strategy for a Nonholonomic Wheeled Mobile Robot with a Pan Camera," *Proceedings of the 9th International Workshop on Robot Motion and Control*, 2013.

[9] T. Ebata, M. Ito, M. Shibata, "Visual Feedback Control Method of a Wheeled Mobile Robot Using a Pan Camera Preferentially," *2014 IEEE 13th International Workshop on Advanced Motion Control (AMC)*, pp. 149-154, 2014.

[10] K. Hashimoto, "A review on vision-based control of robot manipulators," *Advanced Robotics*, volume 17, pp. 969-991, 2003.

[11] S.A. Hutchinson, G.D. Hager, P.I. Corke, "A Tutorial on Visual Servo Control," *IEEE Transactions on Robotics and Automation*, volume 12, pp. 651-670, 1996.

[12] D. Kulas, J. Wang, M. Obeng, X. Wu, "Image-Based Path Following and Target Tracking Control for a Mobile Robot," *2013 Proceedings of IEEE Southeastcon*, pp. 1-6, 2013.

[13] Z. Zhang, R. Sa, Y. Wang, "A Real Time Object Tracking Approach for Mobile Robot Visual Servo Control," *2011 First Asian Conference on Pattern Recognition (ACPR)*, pp. 500-504, 2011.

[14] P.I. Corke, S.A. Hutchinson, "A New Partitioned Approach to Image-Based Visual Servo Control," *IEEE Transactions on Robotics and Automation*, volume 17,pp. 507-515, 2001.

[15] F. Chaumette and S. Hutchinson, "Visual Servoing and Visual Tracking", *Springer Handbook of Robotics* B. Siciliano and O. Khatib, eds., Cambridge University Press, pages 563-583, 2008.

[16] F. Chaumette and E. Malis, "2 1/2 D visual servoing: a possible solution to improve image-based and position-based visual servoings," *IEEE International Conference on Robotics and Automation* , 2000.

[17] G. Bradski and A. Kaehler, "Learning OpenCV: Computer vision with the OpenCV Library," *O'Reilly Media, Inc.*, 2008.

[18] B. Jähne, *Practical Handbook on Image Processing for Scientific and Technical Applications*, CRC Press, Second Edition, 2004.

[19] R. Laganire, *OpenCV 2 Computer Vision Application Programming Cookbook*, Packt Publishing, 2011.

[20] L. Shapiro, G. Stockman, *Computer Vision*, 2000.

[21] J.E. Solem, *Programming Computer Vision with Python*, 2012.

[22] R. Szeliski, *Computer Vision Algorithms and Applications*, Springer, 2011.

[23] D. Casanova, "On Minimum Time Vehicle Manoeuvring: The Theoretical Optimal Lap," Cranfield University, PhD Thesis, 2000.

[24] T. Gustafsson, "Computing the Ideal Racing Line Using Optimal Control," Linkping University, MS Thesis, 2008.

[25] S. van Koutrik, "Optimal Control for Race Car Minimum Time Maneuvering," Delft University of Technology, MS Thesis, 2015.

[26] R. Verschueren, S. De Bruyne, et. al., "Towards Time-Optimal Race Car Driving using Nonlinear MPC in Real-Time." *IEEE Conference on Decision and Control*, 2014.

[27] D.J.N. Limebeer and G. Perantoni "Optimal Control of a Formula One Car on a Three-Dimensional Track - Part 2: Optimal Control." *Journal of Dynamic Systems, Measurement, and Control*, vol.137, no.2, pp. 36-56, 2015.

[28] E. Velenis, P. Tsiotras and J. Lu "Modeling Aggresive Maneuvers on Loose Surfaces: The Cases of Trail-Braking and Pendulum Turn." *Proceedings of the European Control Conference*, 2007.

[29] P.J. Enright and B.A. Conway, "Discrete Approximations to Optimal Trajectories Using Direct Transcription and Nonlinear Programming," *Journal of Guidance, Control, and Dynamics*, vol.15, No. 4, pp. 994-1002, 1992.

[30] H. Seywald, R.R. Kumar and T.A. Wetzel, "Does the Principle of Optimality Hold Along Collocation Solutions?," *AIAA Guidance Navigation and Control Conference*, 1996.

[31] D. Subbaram, "Optimal Control Systems," *CRC Press*, 2003.

[32] D.E. Kirk "Optimal Control Theory: An Introduction," *Prentice Hall*, 1970.

[33] J. Nocedal, S. Wright "Numerical Optimization," *Springer*, 2006.

[34] R.H. Byrd, J. Nocedal and R.A. Waltz "KNITRO: An Integrated Package for Nonlinear Optimization." In G. di Pilo and M. Roma, editors, *Large-Scale Nonlinear Optimization*, pp. 35-59. Springer-Verlag, 2006.

[35] D.J.N. Limebeer and A.V. Rao "Faster, Higher, and Greener: Vehicular Optimal Control." *IEEE Control Systems*, vol.35, no.2, pp. 36-56, 2015.

[36] W. Zhang, T. Inanc, "A Tutorial for Applying DMOC to Solve Optimization Control Problems," *2010 IEEE International Conference on Systems Man and Cybernetics (SMC)*, pp. 1857-1862, 2010.

[37] NEOS Server: State-of-the-Art Solvers for Numerical Optimization, http://www.neos-server.org/neos/

[38] J. Czyzyk, et al., "The NEOS Server," *IEEE Journal on Computational Science and Engineering*, 5(3), pp. 68-75, 1998.

[39] E. Dolan, "The NEOS Server 4.0 Administrative Guide", *Technical Memorandum ANL/MCS-TM-250*, Mathematics and Computer Science Division, Argonne National Laboratory, 2001.

[40] W. Gropp, J.J. Mor, "Optimization Environments and the NEOS Server", *Approximation Theory and Optimization* M. D. Buhmann and A. Iserles, eds., Cambridge University Press, pages 167-182, 1997.

[41] R. Fourer, D.M. Gay, B.W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, Cengage Learning, 2003.

[42] AMPL, http://ampl.com/

[43] R. Nylin, "Evaluation of Optimization Solvers in Mathematica with focus on Optimal Control Problems", Chalmers University of Technology, MS Thesis, 2013.

[44] M.I. Ribeiro and P. Lima, "Kinematic Models of Mobile Robots," Instituto de Sistemas e Robotica, 2002: 1000-1049.

[45] R. Fierro and F.L. Lewis, "Control of a Nonholonomic Mobile Robot: Backstepping Kinematics into Dynamics," *Proceedings of the 34th IEEE Conference In Decision and Control*, volume 4, 1995.

[46] Y. Kanayama, Y. Kimura, F. Miyazaki, et al., "A Stable Tracking Control Method for an Autonomous Mobile Robot," *Proceedings of IEEE International Conference on Robotics and Automation,* pp. 384-389, 1990.

[47] A.M. Bloch, M. Reyhanoglu and H. McClamorch, "Control and Stabilization of Nonholonomic Dynamic Systems," *IEEE Transactions On Automatic Control*, 37, 11, pp. 1746-1757, 1992.

[48] I. Kolmanovsky, N.H. McClamroch, "Developments in Nonholonomic Control Problems," *IEEE Control Systems,* 1995, 15(6): 20-36.

[49] A. Gholipour, M.J. Yazdanpanah, "Dynamic Tracking Control of Nonholonomic Mobile Robot with Model Reference Adaptation for Uncertain Parameters," *European Control Conference (ECC)*, University of Cambridge, UK, 2003.

[50] R. Pepy, A. Lambert, and H. Mounier, "Path Planning using a Dynamic Vehicle Model," *Information and Communication Technologies, ICTTA'06. 2nd. IEEE, 1: 781-786*, 2006.

[51] B.A. Francis, W.M. Wonham, "The Internal Model Principle of Control Theory," *Automatica*, 1976, 12(5): 457-465.

[52] G. Stein, "Respect the Unstable," *IEEE Control Systems Magazine*, 2003.

[53] K. J. Astrom and T. Hagglund, *PID Controllers: Theory, Design, and Tuning*, Instrument Society of America, Research Triangle Park, North Carolina, 1995.

[54] B. Singh, R.P. Payasi, K.S. Verma, et al., "Design of Controllers PD, PI & PID for Speed Control of DC Motor Using IGBT Based Chopper," *German Journal of Renewable and Sustainable Energy Research (GJRSER)*, 2013, 1(1).

[55] A.A. Rodriguez, *Analysis and Design of Feedback Control Systems*, Control3D,L.L.C., Tempe, AZ, 2002.

[56] A.A. Rodriguez, *Linear Systems: Analysis and Design*, Control3D,L.L.C., Tempe, AZ, 2002.

[57] I. Anvari, "Non-holonomic Differential-Drive Mobile Robot Control & Design: Critical Dynamics and Coupling Constraints", Arizona State University, MS Thesis, 2013.

[58] Z. Lin, "Modeling, Design and Control of Multiple Low-Cost Robotic Ground Vehicles," Arizona State University, MS Thesis, 2015.

[59] Z. Li, "Modeling and Control of a Longitudinal Platoon of Ground Robotic Vehicles," Arizona State University, MS Thesis, 2016.

[60] M. Brambilla, E. Ferrante, M. Birattari, et al., "Swarm robotics: A review from the swarm engineering perspective," Swarm Intelligence, 2013, 7(1): 1-41.

[61] B. Siciliano, L. Sciavicco, L. Villani, et al., Robotics: Modeling, Planning and Control, Springer Science & Business Media, 2009.

[62] C.W. De Silva, *Sensors and actuators: control system instrumentation,* CRC Press, 2007.

[63] W.P. Aung, "Analysis on Modeling and Simulink of DC Motor and its Driving System Used for Wheeled Mobile Robot," World Academy of Science, Engineering and Technology, 2007, 32: 299-306.

[64] S.E. Lyshevski, "Nonlinear control of mechatronic systems with permanent-magnet DC motors," *Mechatronics*, 1999.

[65] T. Kara, I. Eker, "Nonlinear modeling and identification of a DC motor for bidirectional operation with real time experiments," *Energy Conversion and Management*, 2004, 45(7): 1087-1106.

[66] C. Batten, "Control for Mobile Robots," *Maslab IAP Robotics Course*, 2005.

[67] L. Feng, Y. Koren, J. Borenstein, "Cross-coupling Motion Controller for Mobile Robots," IEEE Control Systems, 1993, 13(6): 35-43.

[68] Arduino Uno description, https://www.arduino.cc/en/Main/arduinoBoardUno.

[69] Introducing the Raspberry Pi 2 - Model B - Adafruit Learning. https://learn.adafruit.com/introducing-the-raspberry-pi-2-model-b/overview

[70] "Basics of rotary encoders: Overview and new technologies," http://machinedesign.com/sensors/basics-rotary-encoders-overview-and-new-technologies-0

[71] Raspberry Pi 5MP camera datasheet. https://www.raspberrypi.org/documentation/hardware/camera.md

[72] Adafruit BNO055 Absolute Orientation Sensor, https://learn.adafruit.com/adafruit-bno055-absolute-orientation-sensor/overview

[73] SG90 9g Micro Servo, http://www.micropik.com/PDF/SG90Servo.pdf

# APPENDIX A

# MATLAB CODE

```
% INNER LOOP design
% Plots for L,T,S,Try, Tru, Tdiy are presented

clear
close
clc


s = tf('s');
% SSR for P = 5.495 * (1.73/(s+1.73))  (Decoupled)
Ap = [-1.73    0
       0  -1.73];
Bp = [9.507     0
         0     9.507];
Cp = [1   0
       0   1];
Dp = 0*ones(2,2);

r = 0.05;
dw = 0.14;
M = [r/2   r/2
     -r/dw  r/dw];
Minv = inv(M);

P = ss(Ap,Bp,Cp,Dp); %(wr,wl) system

%%
figure(1)
step(P, 5)
grid on
%set( findobj(gca,'type','line'), 'LineWidth', 2);
h = findobj(gcf, 'type', 'line');
set(h, 'LineWidth', 3);
a = findobj(gcf, 'type', 'axes');
set(a, 'linewidth', 4);
set(a, 'FontSize', 15);
title('Plant Step Response', 'FontSize', 20)
ylabel('Angular velocity [rad/s]', 'FontSize', 13)

%%
figure(2)
opts = bodeoptions;
opts.InputLabels.FontSize = 10;
opts.OutputLabels.FontSize = 10;
%opts.YLim = {[-400,100]}; %{maglimits;phaselimits}
%opts.YLimMode = {'manual'};
%opts.XLim = {[1e-01,1e02]}; %{maglimits;phaselimits}
%opts.XLimMode = {'manual'};
bode(P,opts)
title('Plant Frequency Response', 'FontSize', 20);
grid on;
%set( findobj(gca,'type','line'), 'LineWidth', 3);
h = findobj(gcf, 'type', 'line');
set(h, 'LineWidth', 3);
a = findobj(gcf, 'type', 'axes');
set(a, 'linewidth', 4);
set(a, 'FontSize', 15);
%xlabel('freq', 'FontSize', 24);
%ylabel('', 'FontSize', 24);


%% wg = 1, PM = 80

kp = 0.073;
ki = 0.194;

```

```
Kinner_0 = [ kp*(s+ki/kp)*100/(s*(s+100))           0
      0             kp*(s+ki/kp)*100/(s*(s+100))   ];

Kinner_0 = ss(Kinner_0);

%open loop
Linner_0 = P*Kinner_0;

% sensitivity
asen = Linner_0.a - Linner_0.b*Linner_0.c;
bsen = Linner_0.b;
csen = -Linner_0.c;
[row,col] = size(csen);
[row1,col1] = size(bsen);
dsen = eye(row,col1);
Sinner_0 = ss(asen,bsen,csen,dsen);

% comp sensitivity unfiltered
acl = Linner_0.a - Linner_0.b*Linner_0.c;
bcl = Linner_0.b;
ccl = Linner_0.c;
dcl = Linner_0.d;
T_0 = ss(acl,bcl,ccl,dcl);

z = ki/kp;
W_0 = [z/(s+z) 0
          0      z/(s+z)];
W_0 = ss(W_0);

% try = comp sensitivity filtered
Try_0 = T_0*W_0;

% Tdiy
Adiy_0 = [Ap-Bp*Kinner_0.d*Cp     Bp*Kinner_0.c
         -Kinner_0.b*Cp             Kinner_0.a   ];
[row,col]=size(Kinner_0.b);
Bdiy_0 = [        Bp
              0*ones(row,2)     ];
[row1,col1]=size(Kinner_0.a);
Cdiy_0 = [Cp   0*ones(2,col1)];
Ddiy_0 = 0*ones(2,2);
Tdiy_0 = ss(Adiy_0,Bdiy_0,Cdiy_0,Ddiy_0);


Tru_0 = Kinner_0*Sinner_0;      % Tru unfiltered
Truf_0 = Kinner_0*Sinner_0*W_0;    % Tru filtered

Tru_0_vw = Tru_0*Minv;           % Tru vw unfiltered
Tdiy_0_vw = M*Tdiy_0;            % Tdiy vw


%% wg = 2, PM = 60

kp = 0.096;
ki = 0.519;

Kinner_1 = [ kp*(s+ki/kp)*100/(s*(s+100))           0
0                         kp*(s+ki/kp)*100/(s*(s+100))   ];

Kinner_1 = ss(Kinner_1);

Linner_1 = P*Kinner_1;  %open loop


asen = Linner_1.a - Linner_1.b*Linner_1.c;% sensitivity
bsen = Linner_1.b;
csen = -Linner_1.c;
```

```
[row,col] = size(csen);
[row1,col1] = size(bsen);
dsen = eye(row,col1);
Sinner_1 = ss(asen,bsen,csen,dsen);

acl = Linner_1.a - Linner_1.b*Linner_1.c; %comp sens unfiltered
bcl = Linner_1.b;
ccl = Linner_1.c;
dcl = Linner_1.d;
T_1 = ss(acl,bcl,ccl,dcl);

z = ki/kp;
W_1 = [z/(s+z) 0
          0      z/(s+z)];
W_1 = ss(W_1);

Try_1 = T_1*W_1;   % try = comp sensitivity filtered

% Tdiy
Adiy_1 = [Ap-Bp*Kinner_1.d*Cp     Bp*Kinner_1.c
         -Kinner_1.b*Cp             Kinner_1.a   ];
[row,col]=size(Kinner_1.b);
Bdiy_1 = [        Bp
              0*ones(row,2)    ];
[row1,col1]=size(Kinner_1.a);
Cdiy_1 = [Cp  0*ones(2,col1)];
Ddiy_1 = 0*ones(2,2);
Tdiy_1 = ss(Adiy_1,Bdiy_1,Cdiy_1,Ddiy_1);


Tru_1 = Kinner_1*Sinner_1;      % Tru unfiltered
Truf_1 = Kinner_1*Sinner_1*W_1;   % Tru filtered

Tru_1_vw = Tru_1*Minv;           % Tru vw unfiltered
Tdiy_1_vw = M*Tdiy_1;            % Tdiy vw


%% wg = 4, PM = 60

kp = 0.288;
ki = 1.426;

Kinner_2 = [ kp*(s+ki/kp)*100/(s*(s+100))           0
0                        kp*(s+ki/kp)*100/(s*(s+100))   ];

Kinner_2 = ss(Kinner_2);

Linner_2 = P*Kinner_2;  %open loop


asen = Linner_2.a - Linner_2.b*Linner_2.c; % sensitivity
bsen = Linner_2.b;
csen = -Linner_2.c;
[row,col] = size(csen);
[row1,col1] = size(bsen);
dsen = eye(row,col1);
Sinner_2 = ss(asen,bsen,csen,dsen);

acl = Linner_2.a - Linner_2.b*Linner_2.c;% comp sens unfiltered
bcl = Linner_2.b;
ccl = Linner_2.c;
dcl = Linner_2.d;
T_2 = ss(acl,bcl,ccl,dcl);

z = ki/kp;
W_2 = [z/(s+z) 0
          0      z/(s+z)];
```

```
W_2 = ss(W_2);

Try_2 = T_2*W_2;        % try = comp sensitivity filtered

% Tdiy
Adiy_2 = [Ap-Bp*Kinner_2.d*Cp     Bp*Kinner_2.c
         -Kinner_2.b*Cp            Kinner_2.a   ];
[row,col]=size(Kinner_2.b);
Bdiy_2 = [        Bp
              0*ones(row,2)    ];
[row1,col1]=size(Kinner_2.a);
Cdiy_2 = [Cp   0*ones(2,col1)];
Ddiy_2 = 0*ones(2,2);
Tdiy_2 = ss(Adiy_2,Bdiy_2,Cdiy_2,Ddiy_2);


Tru_2 = Kinner_2*Sinner_2;      % Tru unfiltered
Truf_2 = Kinner_2*Sinner_2*W_2;    % Tru filtered

Tru_2_vw = Tru_2*Minv;           % Tru vw unfiltered
Tdiy_2_vw = M*Tdiy_2;            % Tdiy vw
%%



figure(3)   % open loop
w = logspace(-2,3,100);
sv_L0=sigma(Linner_0,w); sv_L1=sigma(Linner_1,w);...
    sv_L2=sigma(Linner_2,w);
sv_L0=20*log10(sv_L0); sv_L1=20*log10(sv_L1);...
    sv_L2=20*log10(sv_L2);
semilogx(w, sv_L0(1,:),'b' , w, sv_L1(1,:),'k',...
    w, sv_L2(1,:),'r',...
         w, sv_L0(2,:),'b' , w, sv_L1(2,:),'k',...
         w, sv_L2(2,:),'r','LineWidth', 3)
title('Open Loop Singular Values','FontWeight','normal')
grid on
xlabel('Frequency (rad/sec)')
ylabel('Singular Values (dB)')
legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951')
set(gca,'fontsize',19)
set(gca,'linewidth',3)

figure(4)    %sensitivity
w = logspace(-2,3,100);
sv_S0 = sigma(Sinner_0,w); sv_S1=sigma(Sinner_1,w);...
    sv_S2=sigma(Sinner_2,w);
sv_S0 = 20*log10(sv_S0); sv_S1 = 20*log10(sv_S1);...
    sv_S2 = 20*log10(sv_S2);
semilogx(w, sv_S0(1,:),'b' ,w, sv_S1(1,:),'k', w,...
sv_S2(1,:),'r',...
         w, sv_S0(2,:),'b' ,w, sv_S1(2,:),'k', w,...
         sv_S2(2,:),'r', 'LineWidth', 3)
title(' S ','FontWeight','normal')
grid on
xlabel('Frequency (rad/sec)')
ylabel('Singular Values (dB)')
legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951',...
    'Location', 'southeast')
set(gca,'fontsize',19)
set(gca,'linewidth',3)

figure(5)   %comp sensitivity
w = logspace(-2,3,100);
sv_T0 = sigma(T_0,w); sv_T1 = sigma(T_1,w);...
```

```
    sv_T2 = sigma(T_2,w);
sv_T0 = 20*log10(sv_T0); sv_T1 = 20*log10(sv_T1);...
    sv_T2 = 20*log10(sv_T2);
semilogx(w, sv_T0(1,:),'b', w, sv_T1(1,:),'k',...
    w, sv_T2(1,:),'r',...
         w, sv_T0(2,:),'b', w, sv_T1(2,:),'k',...
         w, sv_T2(2,:),'r','LineWidth', 3)
title(' T ','FontWeight','normal')
grid on
xlabel('Frequency (rad/sec)')
ylabel('Singular Values (dB)')
legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951')
set(gca,'fontsize',19)
set(gca,'linewidth',3)

figure(6)    %tdiy
w = logspace(-2,4,100);
sv_Tdiy0 = sigma(Tdiy_0,w);sv_Tdiy1 = sigma(Tdiy_1,w);...
    sv_Tdiy2 = sigma(Tdiy_2,w);
sv_Tdiy0 = 20*log10(sv_Tdiy0);sv_Tdiy1 = 20*log10(sv_Tdiy1);...
    sv_Tdiy2 = 20*log10(sv_Tdiy2);
semilogx(w, sv_Tdiy0(1,:),'b',w, sv_Tdiy1(1,:),...
    'k',w, sv_Tdiy2(1,:),'r',...
         w, sv_Tdiy0(2,:),'b',w, sv_Tdiy1(2,:),...
         'k',w, sv_Tdiy2(2,:),'r','LineWidth', 3)
title(' T_{diy} ','FontWeight','normal')
grid on
xlabel('Frequency (rad/sec)')
ylabel('Singular Values (dB)')
legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951')
set(gca,'fontsize',19)
set(gca,'linewidth',3)

figure(7)    %tru
w = logspace(-2,4,100);
sv_Tru0 = sigma(Tru_0,w);sv_Tru1 = sigma(Tru_1,w);...
    sv_Tru2 = sigma(Tru_2,w);
sv_Tru0 = 20*log10(sv_Tru0);sv_Tru1 = ...
    20*log10(sv_Tru1);sv_Tru2 = 20*log10(sv_Tru2);
semilogx(w, sv_Tru0(1,:),'b',w, sv_Tru1(1,:),...
    'k',w, sv_Tru2(1,:),'r',...
         w, sv_Tru0(2,:),'b',w, sv_Tru1(2,:),...
         'k',w, sv_Tru2(2,:),'r','LineWidth', 3)
title(' T_{ru} Unfiltered ','FontWeight','normal')
grid on
xlabel('Frequency (rad/sec)')
ylabel('Singular Values (dB)')
legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951')
set(gca,'fontsize',19)
set(gca,'linewidth',3)

figure(8)    %tru_filtered
w = logspace(-2,4,100);
sv_Truf0 = sigma(Truf_0,w);sv_Truf1 = ...
    sigma(Truf_1,w);sv_Truf2 = sigma(Truf_2,w);
sv_Truf0 = 20*log10(sv_Truf0);sv_Truf1 = ...
    20*log10(sv_Truf1);sv_Truf2 = 20*log10(sv_Truf2);
semilogx(w, sv_Truf0(1,:),'b',w, sv_Truf1(1,:),...
    'k',w, sv_Truf2(1,:),'r',...
         w, sv_Truf0(2,:),'b',w, sv_Truf1(2,:),...
         'k',w, sv_Truf2(2,:),'r','LineWidth', 3)
title(' T_{ru} Filtered ','FontWeight','normal')
grid on
xlabel('Frequency (rad/sec)')
ylabel('Singular Values (dB)')
```

```
332 legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951')
333 set(gca,'fontsize',19)
334 set(gca,'linewidth',3)
335
336
337 % t = 0:0.02:5;
338 % [y, t, x] = lsim(10*Try_0,[ones(size(t))' zeros(size(t))'],t);
339 % plot(t,y(:,:,1))
340 %%
341 opt = stepDataOptions('StepAmplitude',10);
342 %%
343 figure(9)   % Try W
344 step(Try_0, 'b',Try_1, 'k', Try_2, 'r', opt);
345 h = findobj(gcf, 'type', 'line');
346 set(h, 'LineWidth', 3);
347 a = findobj(gcf, 'type', 'axes');
348 set(a, 'linewidth', 4);
349 set(a, 'FontSize', 15);
350 title('Step response (filtered)', 'FontSize', 20)
351 ylabel('Angular velocity [rad/s]', 'FontSize', 13)
352 legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951')
353 grid on
354
355 figure(10)   % Try
356 step(T_0, 'b', T_1,'k', T_2,'r', opt)
357 h = findobj(gcf, 'type', 'line');
358 set(h, 'LineWidth', 3);
359 a = findobj(gcf, 'type', 'axes');
360 set(a, 'linewidth', 4);
361 set(a, 'FontSize', 15);
362 title('Step response (unfiltered)', 'FontSize', 20)
363 ylabel('Angular velocity [rad/s]', 'FontSize', 13)
364 legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951')
365 grid on
366
367
368
369 figure(11)    % Tru
370 step(Tru_0, 'b', Tru_1,'k', Tru_2,'r', opt)
371 h = findobj(gcf, 'type', 'line');
372 set(h, 'LineWidth', 3);
373 a = findobj(gcf, 'type', 'axes');
374 set(a, 'linewidth', 4);
375 set(a, 'FontSize', 15);
376 title('Unfiltered control response', 'FontSize', 20)
377 ylabel('Voltage [V]', 'FontSize', 13)
378 legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951')
379 grid on
380
381 figure(12)   % Tru W
382 step(Truf_0,'b', Truf_1,'k', Truf_2,'r', opt)
383 h = findobj(gcf, 'type', 'line');
384 set(h, 'LineWidth', 3);
385 a = findobj(gcf, 'type', 'axes');
386 set(a, 'linewidth', 4);
387 set(a, 'FontSize', 15);
388 title('Filtered control response', 'FontSize', 20)
389 ylabel('Voltage [V]', 'FontSize', 13)
390 legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951')
391 grid on
392 %%
393
394 figure(13)    %MSP
395 w = logspace(-2,4,100);
396 sv_MSP0 = sigma(Tdiy_0_vw,w);sv_MSP1 = ...
```

```
397      sigma(Tdiy_1_vw,w);sv_MSP2 = sigma(Tdiy_2_vw,w);
398 sv_MSP0 = 20*log10(sv_MSP0);sv_MSP1 = ...
399      20*log10(sv_MSP1);sv_MSP2 = 20*log10(sv_MSP2);
400 semilogx(w, sv_MSP0(1,:),'b',w,...
401      sv_MSP1(1,:),'k',w, sv_MSP2(1,:),'r',...
402           w, sv_MSP0(2,:),'b',w, ...
403           sv_MSP1(2,:),'k',w, sv_MSP2(2,:),'r','LineWidth', 3)
404 title(' MSP ','FontWeight','normal')
405 grid on
406 xlabel('Frequency (rad/sec)')
407 ylabel('Singular Values (dB)')
408 legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951')
409 set(gca,'fontsize',19)
410 set(gca,'linewidth',3)
411
412
413 figure(14)    %KSM^{-1}
414 w = logspace(-2,4,100);
415 sv_KSM0 = sigma(Tru_0_vw,w);sv_KSM1 =...
416      sigma(Tru_1_vw,w);sv_KSM2 = sigma(Tru_2_vw,w);
417 sv_KSM0 = 20*log10(sv_KSM0);sv_KSM1 = ...
418      20*log10(sv_KSM1);sv_KSM2 = 20*log10(sv_KSM2);
419 semilogx(w, sv_KSM0(1,:),'b',w,...
420      sv_KSM1(1,:),'k',w, sv_KSM2(1,:),'r',...
421           w, sv_KSM0(2,:),'b',w,...
422       sv_KSM1(2,:),'k',w, sv_KSM2(2,:),'r','LineWidth', 3)
423 title(' KSM^{-1} ','FontWeight','normal')
424 grid on
425 xlabel('Frequency (rad/sec)')
426 ylabel('Singular Values (dB)')
427 legend('g=0.073,z=2.657','g=0.096, z=5.406','g=0.288, z=4.951')
428 set(gca,'fontsize',19)
429 set(gca,'linewidth',3)
```

```
1  % INNER LOOP simulation and experimental
2  % results
3
4  clear
5  close
6  clc
7
8  %%
9  %Experimental data
10 % wL  wR   PWML  PWMR
11 out2_2 = [0.00  0.00  10  10
12   0.00  0.00  35  35
13   0.00  0.00  61  61
14   0.00  0.00  87  87
15   1.96  1.96  103  103
16   3.93  3.93  111  111
17   5.89  3.93  116  126
18   7.85  5.89  115  132
19   7.85  9.82  119  120
20   9.82  9.82  115  117
21   11.78  11.78  103  110
22   11.78  13.74  98  92
23   13.74  13.74  84  82
24   15.71  13.74  62  73
25   13.74  13.74  56  63
26   11.78  15.71  59  44
27   11.78  13.74  55  36
28   11.78  11.78  50  41
29   11.78  11.78  46  36
30   9.82  11.78  51  31
31   9.82  9.82  54  37
32   9.82  7.85  53  49
```

```
    7.85   7.85   64   56
    7.85   7.85   72   61
    9.82   9.82   66   57
    9.82   9.82   65   55
    7.85   7.85   76   67
    9.82   9.82   74   64
    9.82   9.82   71   61
    7.85   7.85   83   73
    7.85   7.85   90   80
    9.82   9.82   85   75
    11.78   11.78   73   63
    9.82   9.82   78   68
    9.82   9.82   81   71
    11.78   9.82   70   70
    9.82   9.82   74   71
    9.82   11.78   78   61
    11.78   11.78   66   55
    11.78   9.82   60   61
    9.82   9.82   66   63
    9.82   11.78   69   53
    9.82   9.82   68   56
    9.82   9.82   69   60
    11.78   9.82   59   58
    9.82   9.82   63   61
    7.85   9.82   77   60
    9.82   7.85   72   71
    9.82   9.82   70   69
    9.82   9.82   73   65
    9.82   9.82   72   68
    9.82   9.82   73   67
    11.78   9.82   63   69
    9.82   11.78   66   59
    9.82   9.82   71   62
    9.82   7.85   69   76
    9.82   9.82   71   72
    9.82   11.78   70   60
    9.82   9.82   71   64
    11.78   9.82   61   68
    11.78   11.78   55   56
    11.78   9.82   52   60
    9.82   9.82   56   64
    9.82   11.78   59   53
    11.78   11.78   48   47
    9.82   11.78   52   43
    9.82   9.82   56   48
    11.78   9.82   44   51
    9.82   11.78   49   40
    9.82   9.82   52   44];

wL2_2 = out2_2(:,1);
wR2_2 = out2_2(:,2);
PWML2_2 = out2_2(:,3);
PWMR2_2 = out2_2(:,4);
[m22,~]=size(wL2_2);
off_d = zeros(m22,1);
time22=0:0.1:m22*0.1-0.1;
%%
s = tf('s');
% SSR for P = 5.495 * (1.73/(s+1.73))  (Decoupled)
Ap = [-1.73    0
       0  -1.73];
Bp = [9.507     0
         0     9.507];
Cp = [1   0
       0   1];
Dp = 0*ones(2,2);

r = 0.05;
dw = 0.14;
M = [r/2   r/2
```

```
      -r/dw  r/dw];
Minv = inv(M);

P = ss(Ap,Bp,Cp,Dp);

%% wg = 2, PM = 60

kp = 0.096;
ki = 0.519;

Kinner_1 = [ kp*(s+ki/kp)*100/(s*(s+100))            0
0                          kp*(s+ki/kp)*100/(s*(s+100))   ];

Kinner_1 = ss(Kinner_1);

%open loop
Linner_1 = P*Kinner_1;

% sensitivity
asen = Linner_1.a - Linner_1.b*Linner_1.c;
bsen = Linner_1.b;
csen = -Linner_1.c;
[row,col] = size(csen);
[row1,col1] = size(bsen);
dsen = eye(row,col1);
Sinner_1 = ss(asen,bsen,csen,dsen);

% comp sensitivity unfiltered
acl = Linner_1.a - Linner_1.b*Linner_1.c;
bcl = Linner_1.b;
ccl = Linner_1.c;
dcl = Linner_1.d;
T_1 = ss(acl,bcl,ccl,dcl);

z = ki/kp;
W_1 = [z/(s+z)  0
          0     z/(s+z)];
W_1 = ss(W_1);

 % try = comp sensitivity filtered
Try_1 = T_1*W_1;

% Tdiy
Adiy_1 = [Ap-Bp*Kinner_1.d*Cp     Bp*Kinner_1.c
         -Kinner_1.b*Cp            Kinner_1.a   ];
[row,col]=size(Kinner_1.b);
Bdiy_1 = [        Bp
             0*ones(row,2)     ];
[row1,col1]=size(Kinner_1.a);
Cdiy_1 = [Cp   0*ones(2,col1)];
Ddiy_1 = 0*ones(2,2);
Tdiy_1 = ss(Adiy_1,Bdiy_1,Cdiy_1,Ddiy_1);


Tru_1 = Kinner_1*Sinner_1;      % Tru unfiltered
Truf_1 = Kinner_1*Sinner_1*W_1;   % Tru filtered

Tru_1_vw = Tru_1*Minv;          % Tru vw unfiltered
Tdiy_1_vw = M*Tdiy_1;           % Tdiy vw


%%
opt = stepDataOptions('StepAmplitude',10);
set(0,'defaultAxesFontSize', 20);

%%
% Try W
```

```
[y_filt ,time1] = step(Try_1, 'k', opt, 6.9);

figure(9)
subplot(2,2,1)
plot(time1, y_filt(:,1,1), 'b', time22,...
    wR2_2, 'r', 'LineWidth', 3)
grid on
ylabel('Angular Velocity [rad/s]')
xlabel('Time [s]')
legend('Simulation', 'Experimental')
set(gca,'fontsize',13)
set(gca,'linewidth',3)

subplot(2,2,2)
plot(time1, y_filt(:,1,2), 'b', time22,...
    off_d, 'r', 'LineWidth', 3)
grid on
ylabel('Angular Velocity [rad/s]')
xlabel('Time [s]')
legend('Simulation', 'Experimental')
set(gca,'fontsize',13)
set(gca,'linewidth',3)

subplot(2,2,3)
plot(time1, y_filt(:,2,1), 'b', time22,...
    off_d, 'r','LineWidth', 3)
grid on
ylabel('Angular Velocity [rad/s]')
xlabel('Time [s]')
legend('Simulation', 'Experimental')
set(gca,'fontsize',13)
set(gca,'linewidth',3)

subplot(2,2,4)
plot(time1, y_filt(:,2,2), 'b',time22,...
    wL2_2, 'r', 'LineWidth', 3)
grid on
ylabel('Angular Velocity [rad/s]')
xlabel('Time [s]')
legend('Simulation', 'Experimental')
set(gca,'fontsize',13)
set(gca,'linewidth',3)

mtit('Step Response (Filtered)', 'fontsize', 20);

%% Tru W


[u_filt, time2] = step( Truf_1,'k', opt, 6.9);
figure(12)
subplot(2,2,1)
plot(time2, u_filt(:,1,1), 'b', time22,...
    PWMR2_2*5.15/255, 'r','LineWidth', 3)
ylabel('Voltage [V]')
grid on
xlabel('Times [s]')
legend('Simulation', 'Experimental')
set(gca,'fontsize',15)
set(gca,'linewidth',3)

subplot(2,2,2)
plot(time2, u_filt(:,1,2), 'b', ...
    time22, off_d, 'r','LineWidth', 3)
ylabel('Voltage [V]')
grid on
xlabel('Times [s]')
```

```
legend('Simulation', 'Experimental')
set(gca,'fontsize',15)
set(gca,'linewidth',3)


subplot(2,2,3)
plot(time2, u_filt(:,2,1), 'b', ...
    time22, off_d, 'r','LineWidth', 3)
ylabel('Voltage [V]')
grid on
xlabel('Times [s]')
legend('Simulation', 'Experimental')
set(gca,'fontsize',15)
set(gca,'linewidth',3)

subplot(2,2,4)
plot(time2, u_filt(:,2,2), 'b', ...
    time22, PWML2_2*5.15/255, 'r', 'LineWidth', 3)
ylabel('Voltage [V]')
grid on
xlabel('Times [s]')
legend('Simulation', 'Experimental')
set(gca,'fontsize',15)
set(gca,'linewidth',3)
mtit('Control Response (Filtered)', 'fontsize', 20);

%%
wL2_2 = out2_2(:,1);
wR2_2 = out2_2(:,2);

linearVexp = (r/2)*(wR2_2 + wL2_2);
angularVexp = (-r/dw)*wR2_2 + (r/dw)*wL2_2;

linearVsim = (r/2)*(y_filt(:,1,1) + y_filt(:,2,2));
angularVsim = (-r/dw)*y_filt(:,1,1) + (r/dw)*y_filt(:,2,2);

figure(40)
subplot(1,2,1)
plot(time1, linearVsim, 'b', time22,...
    linearVexp, 'r', 'LineWidth',3)
ylabel('Linear Velocity [m/s]')
grid on
xlabel('Times [s]')
legend('Simulation', 'Experimental')
set(gca,'fontsize',19)
set(gca,'linewidth',3)

subplot(1,2,2)
plot(time1, angularVsim, 'b', time22,...
    angularVexp, 'r', 'LineWidth',3)
ylabel('Angular Velocity [rad/s]')
grid on
xlabel('Times [s]')
ylim([-2 2])
legend('Simulation', 'Experimental')
set(gca,'fontsize',19)
set(gca,'linewidth',3)
mtit('(v, \omega) Inner Loop Response (Filtered)',...
    'fontsize', 20);
```

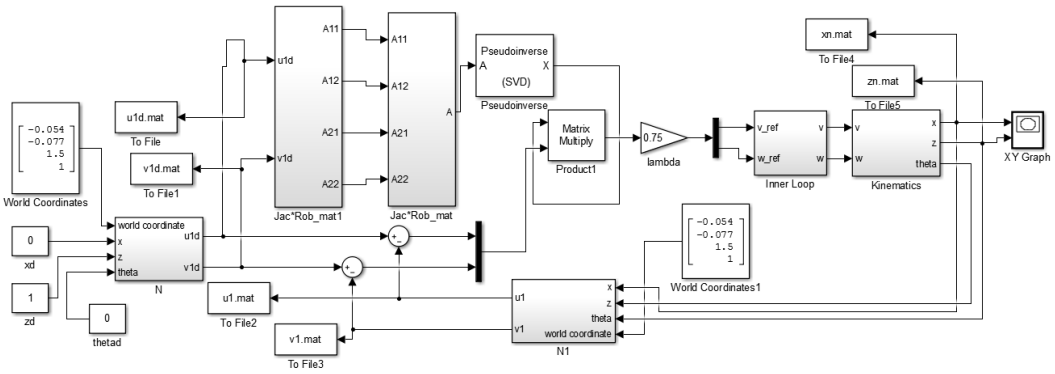


Figure A.1: Simulink Model IBVS - One Marker

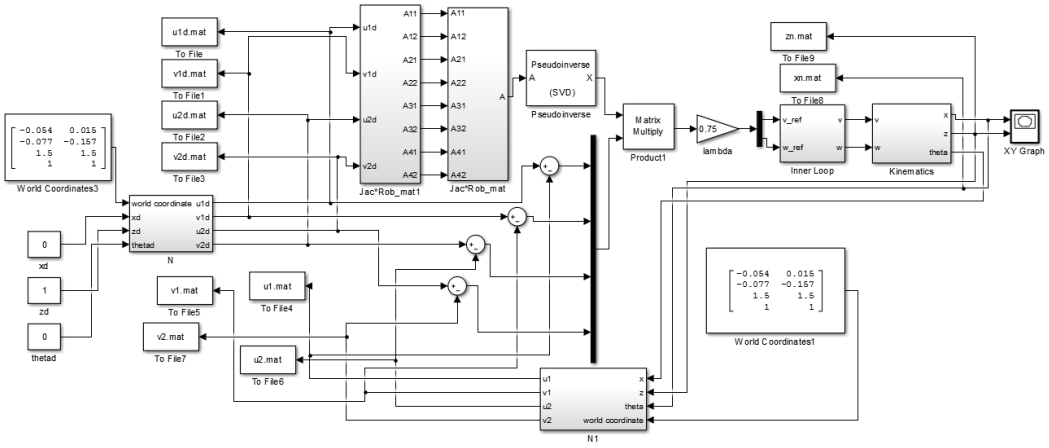


Figure A.2: Simulink Model IBVS - Two Markers

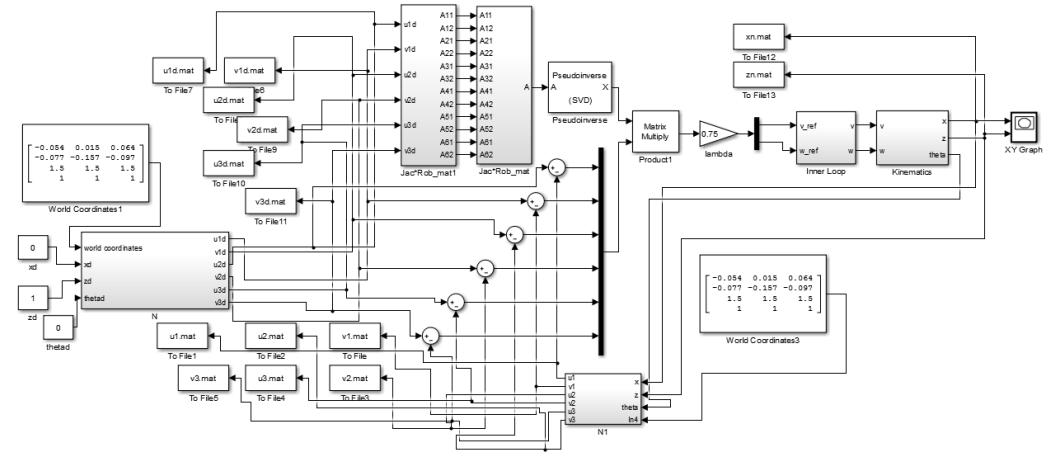


Figure A.3: Simulink Model IBVS - Three Markers

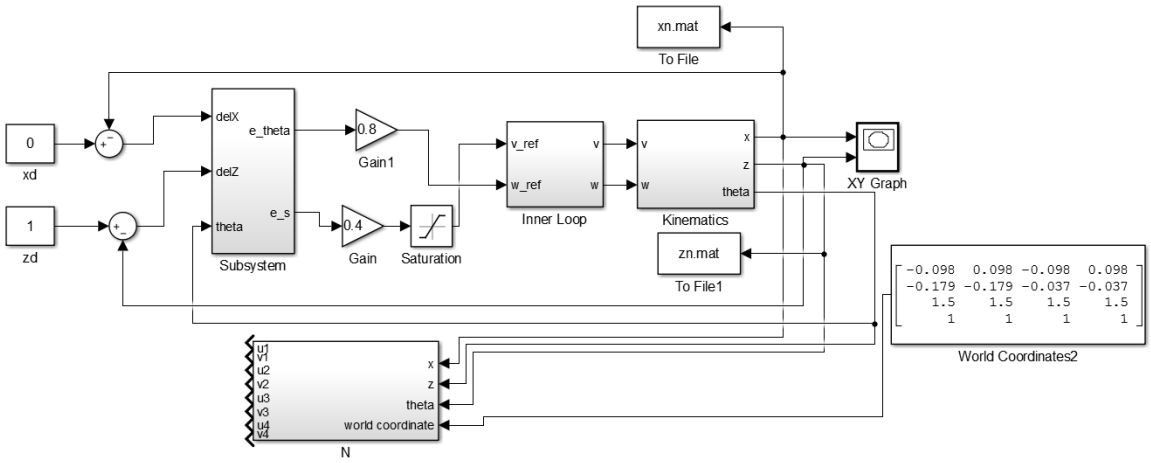


Figure A.4: Simulink Model PBVS

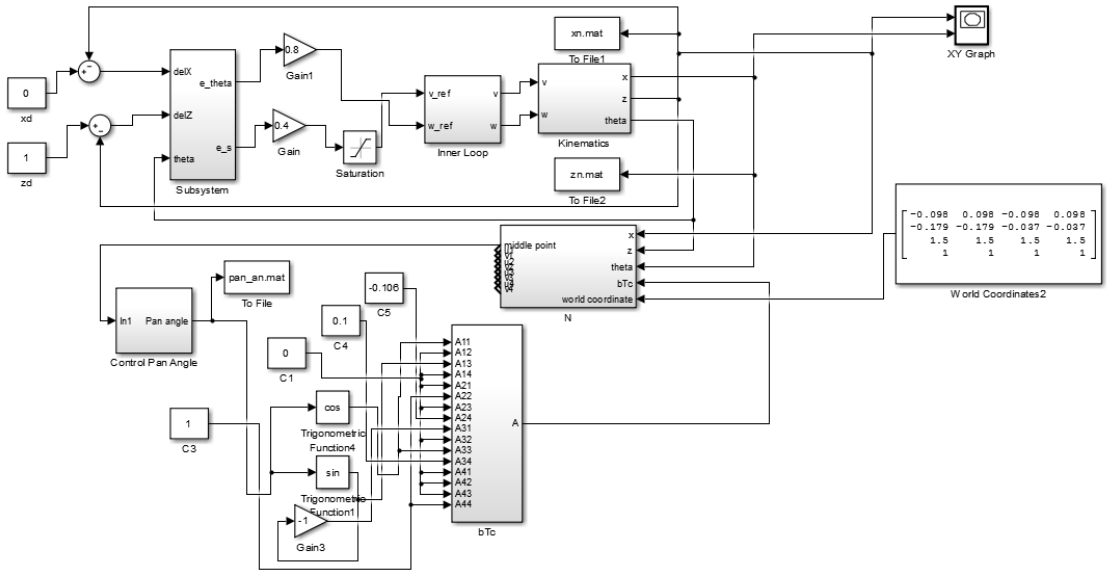


Figure A.5: Simulink Model PBVS with Pan Camera

```
% M-file to generate oval racetrack data
% There are 2 U turns on the track with a curvature
% radius of 1 m each.

clear
close all
clc

% radius
r1 = 1.0;
r2 = 1.0;


```

```
%step size 1 mm
s = 0:0.001:10.282;

s=s';

theta_t=[ 0*ones(2001,1)     %2m, t=2m     1 until 2001
      (s(2002:5142,1)-2)./r1 %3.141m, t=5.141m 2002 until 5142
       pi*ones(2000,1)           %2m, t=7.141m  5143 until 7142
pi+(s(7143:10283,1)-7.141)./r2  ]; %3.141m, t=10.282m  7143 until 10283

x_t = [          0*ones(2001,1)
      r1-r1.*cos(theta_t(2002:5142,1))
                 2*ones(2000,1)
      2-r2+r2.*cos((s(7143:10283,1)-7.141)./r2) ];


z_t = [          s(1:2001,1)
      2+r1.*sin(theta_t(2002:5142,1))
             2-(s(5143:7142,1)-(r1*pi+2))
      -r2.*sin((s(7143:10283,1)-7.141)./r2)    ];


k_t(1:2001,1) = 0;           %curvature
k_t(2002:5142,1) = 1/r1;
k_t(5143:7142,1) = 0;
k_t(7143:10283,1) = 1/r2;

%shift 30 cm
x_sh = x_t(301:10283,1);
x_sh = [  x_sh
          0*ones(300,1)      ];

z_sh = z_t(301:10283,1);
z_sh = [  z_sh
         s(1:300,1)       ];

theta_sh = theta_t(301:10283,1);
theta_sh = [  theta_sh
                2*pi*ones(300,1)     ];

[n,~]=size(s);
i=0:1:n-1;
i = i';

% for a camera based method, matrix X is used
X = [i  x_t  z_t  theta_t   k_t   x_sh   z_sh  theta_sh];

% for a noncamera based method matrix X1 is used
X1 = [i  x_t  z_t  theta_t   k_t  ];

% this X, X1 data needs to be copied into a .dat file
% Then the NEOS server solver (KNITRO) can take
% this data file along with the model file and
% provide a solution

%%
% To plot  a 'real' track, i.e. one that has width
% an outer line and an inner line can be used

%outer line
distance_o = 0.06;
x_o = x_t - distance_o.*cos(theta_t);
z_o = distance_o.*sin(theta_t) + z_t;
theta_o = theta_t;

%inner line
distance_i = -0.06;
```

```
x_i = x_t - distance_i.*cos(theta_t);
z_i = distance_i.*sin(theta_t) + z_t;
theta_i = theta_t;


%%
% Output data from KNITRO solver (or any nonlinear
% optimization solver) goes here

out1_1 = [];

%%
% ****************************************
% For a camera based method this is the order*
% of the output                           *
% ****************************************
vref_1 = outj1_1(:,2);
error_1 = outj1_1(:,3);
v_achieved_1 = outj1_1(:,4);
theta_achieved_1 = outj1_1(:,5);
x_opt_1 = outj1_1(:,6);
z_opt_1 = outj1_1(:,7);
accelref_1 = outj1_1(:,8);
accel_1 = outj1_1(:,9);
jerk_1 = outj1_1(:,10);
jerkr_1 = outj1_1(:,11);
time_sim_1 = outj1_1(:,12);
w_1 = outj1_1(:,13);
xcam_1 = outj1_1(:,14);
zcam_1 = outj1_1(:,15);

% ********************************************
% For a noncamera based method this is the order*
% of the output                               *
% ********************************************
% vref_1 = outj1_1(:,2);          %  0
% tref_1 = outj1_1(:,3);
% v_achieved_1 = outj1_1(:,4);
% theta_achieved_1 = outj1_1(:,5);
% x_opt_1 = outj1_1(:,6);
% z_opt_1 = outj1_1(:,7);
% accelref_1 = outj1_1(:,8);
% accel_1 = outj1_1(:,9);
% jerk_1 = outj1_1(:,10);
% jerkr_1 = outj1_1(:,11);
% time_sim_1 = outj1_1(:,12);
% w_1 = outj1_1(:,13);
% ********************************************

%tref_1 = error_1 + theta_achieved_1;

[m,~] = size(v_achieved_1);
elapsed_distance(1) = 0;
for k=1:1:m-1
   elapsed_distance(k+1,1) = elapsed_distance(k,1) + ...
       (time_sim_1(k+1,1)-time_sim_1(k,1))*v_achieved_1(k,1);
end

figure(1)
plot(s,x_t, 'LineWidth', 1.5)
title(' X(s) ')
xlabel(' Track length [m] ')
ylabel(' [m] ')
grid on

```

```
figure(2)
plot(s,z_t, 'LineWidth', 1.5)
title('Z(s)')
xlabel(' Track length [m]')
ylabel(' [m] ')
grid on

figure(3)
plot(s,theta_t, 'k', s,tref_1, 'r--', 'LineWidth', 2)
title('\theta (s)')
xlabel(' Track length [m]')
ylabel(' [m] ')
grid on
legend('\theta_{ref}  with camera','\theta_{ref}')
set(gca,'fontsize',19)
set(gca,'linewidth',3)

figure(30)
plot(s,x_t, 'k',s,z_t, 'b', s,theta_t, 'r', 'LineWidth', 3)
title('Track parameters')
xlabel(' Track length [m]')
ylabel(' [m],[rad] ')
grid on
legend('x(s)','z(s)','\theta(s)')
set(gca,'fontsize',19)
set(gca,'linewidth',3)

figure(4)
plot( s, vref_1,'k', 'LineWidth', 3)
title(' Optimal Commanded Linear Velocity ')
xlabel('Track length [m]')
ylabel(' v [m/s] ')
grid on
set(gca,'fontsize',19)
set(gca,'linewidth',3)
hold on
x1s = [2,2];
x1e = [5.141,5.141];
x2s = [7.141,7.141];
x2e = [10.282,10.282];

y = [0,0.7];
plot(x1s,y,':m' ,x1e,y,':c' ,x2s,y,':m' ,x2e,y,':c' , 'LineWidth', 2)


sl = tf('s');
g = 0.096; z = 5.40625; %inner loop PI
kp = 1.2; kd = 0.5;     %outer loop PD
P = 9.507/(sl+1.73);
K = g*(sl+z)*100/(sl*(sl+100));
L = P*K;
S = 1/(1+L);
Tv = 1 - S;
W = z/(sl+z);
Tv = Tv*W;
Kouter = kd*(sl+kp/kd)*100*100/(sl+100)^2;
Louter = (1/sl)*Tv*Kouter;
So = 1/(1+Louter);
Tthe = 1 - So;


% Simulation part
t_1 = linspace(0, time_sim_1(end), 10283);
vref_i_1 = interp1(time_sim_1,vref_1,t_1);
tref_i_1 = interp1(time_sim_1,tref_1,t_1);
figure(6)
```

```
212 plot(time_sim_1,vref_1, 'o', t_1,vref_i_1,':.')
213 legend('vref','vref interpolated')
214 figure(7)
215 plot(time_sim_1,tref_1, 'o', t_1,tref_i_1,':.')
216 legend('tref','tref interpolated')
217
218 v_actual_1 = lsim(Tv,vref_i_1,t_1);
219 theta_actual_1 = lsim(Tthe,tref_i_1,t_1);
220
221
222
223 x_dot_1 = v_actual_1.*sin(theta_actual_1);
224 z_dot_1 = v_actual_1.*cos(theta_actual_1);
225 xp = 0;
226 zp = 0;
227 x_real_1(1) = xp;
228 z_real_1(1) = zp;
229 [m,~]=size(x_dot_1);
230 delT = time_sim_1(end)/10283;
231 for i=1:m-1
232     x_real_1(i+1) = delT*x_dot_1(i) + x_real_1(i);
233     z_real_1(i+1) = delT*z_dot_1(i) + z_real_1(i);
234 end
235
236
237 figure(80)
238 plot(x_t,z_t, 'k', 'LineWidth',3)
239 title(' Race Track ')
240 xlabel(' x [m]')
241 ylabel(' z [m]')
242 grid on
243 axis equal
244 set(gca,'fontsize',19)
245 set(gca,'linewidth',3)
246
247 figure(8)
248 plot(x_t,z_t, 'k', x_opt_1, z_opt_1,...
249     'r--', x_real_1, z_real_1, 'b:','LineWidth', 3)
250 legend('Racetrack line',...
251     'Optimal Line (optimization)', ...
252     'Optimal Line (simulation)')
253 title(' Race Track ')
254 xlabel(' x [m]')
255 ylabel(' z [m]')
256 grid on
257 axis equal
258 set(gca,'fontsize',19)
259 set(gca,'linewidth',3)
260
261 figure(9)
262 plot(time_sim_1,vref_1, 'k', time_sim_1,...
263     v_achieved_1, 'r--', t_1,v_actual_1,'b:', 'LineWidth', 3)
264 title(' Optimal Linear Velocities ')
265 xlabel(' Time [sec]')
266 ylabel(' Velocity [m/s] ')
267 grid on
268 legend('v_{ref}',...
269     'v_{actual} (optimization)',...
270     'v_{actual} (simulation)')
271 set(gca,'fontsize',19)
272 set(gca,'linewidth',3)
273
274
275 figure(110)
276 plot(time_sim_1,error_1, 'k', 'LineWidth', 3)
277 title(' e_{\theta} ')
```

```
278 xlabel(' Time[s]')
279 ylabel(' Angle [rad] ')
280 grid on
281 set(gca,'fontsize',19)
282 set(gca,'linewidth',3)
```

# APPENDIX B

# ARDUINO UNO CODE

```
// INNER LOOP (wr, wl) SPEED CONTROL
// PI controller with roll-off and prefilter
// wg = 2 rad/s

#include <Wire.h>
#include <Adafruit_MotorShield.h>
#include "utility/Adafruit_PWMServoDriver.h"
#include <math.h>

//Create the motor shield object with the default I2C address
Adafruit_MotorShield AFMS = Adafruit_MotorShield();
//Or, create it with a different I2C address (say for stacking)
//Adafruit_MotorShield AFMS = Adafruit_MotorShield(0x61);

Adafruit_DCMotor *rightMotor = AFMS.getMotor(2);
Adafruit_DCMotor *leftMotor = AFMS.getMotor(4);

#include <Encoder.h>

Encoder le(2,2);
Encoder re(3,3);

// Variables for storing the calculated velocity
double wR;
double wL;
double wRp=0.0;
double wLp=0.0;
double wLn;
double wRn;
double LdVal = 0;
double RdVal = 0;
double Radius =0.05;
double Length =0.14;

double wd = 0;
double vd = 0.5;
double wdr;
double wdl;
double wdr_p=0;
double wdl_p=0;
double wrf;
double wlf;
double wrf_p=0;
double wlf_p=0;

double CR;
double CR_p=0;
double CR_pp=0;
double CL;
double CL_p=0;
double CL_pp=0;

double Lerror;
double Lerror_p = 0;
double Lerror_pp = 0;
double Rerror;
double Rerror_p = 0;
double Rerror_pp = 0;

int PWMR;
int PWML;

double kp = 0.096;
double ki = 0.519;

double alpha = 100;
double h = ki/kp;

long L;
long R;
```

```
71  long L_last=0;
72  long R_last=0;
73  unsigned long Time=0;
74  unsigned long sample_time=100;
75  double td=0.100;
76
77  void setup()
78  {
79
80    AFMS.begin();
81
82    Serial.begin(9600);
83
84    leftMotor->setSpeed(0);
85      rightMotor->setSpeed(0);
86   leftMotor->run(FORWARD);
87   rightMotor->run(FORWARD);
88
89    leftMotor->run(RELEASE);
90    rightMotor->run(RELEASE);
91
92   delay(1000);
93  }
94
95  void loop()
96  {
97    if (millis()<8000)
98    {
99        if(millis()-Time>sample_time)
100         {
101           Time = millis();
102           GetSpeeds();
103         }
104   }
105
106
107   else
108   {
109     rightMotor->setSpeed(0);
110     leftMotor->setSpeed(0);
111   }
112
113 }
114
115
116 void GetSpeeds()
117 {
118     wdr= (2*vd - Length*wd)/(2*Radius);
119     wdl= (2*vd + Length*wd)/(2*Radius);
120
121     wrf = ( (td*h)*wdr + (td*h)*wdr_p - (td*h - 2)*wrf_p)
122               /(2 + td*h);
123     wlf = ( (td*h)*wdl + (td*h)*wdl_p - (td*h - 2)*wlf_p)
124              /(2 + td*h);
125
126     wrf_p = wrf;
127     wlf_p = wlf;
128     wdr_p = wdr;
129     wdl_p = wdl;
130
131
132     L = le.read();
133     R = re.read();
134
135    LdVal = (double)( L- L_last)/ (td);
136    RdVal = (double)( R -R_last)/(td);
137
```

```
138     wL = LdVal*2*3.14159 /32;
139     wR = RdVal*2*3.14159 /32;
140
141     wLn = (wL + wLp)/2.0;
142     wRn = (wR + wRp)/2.0;
143
144     wLp = wL;
145     wRp = wR;
146
147     Rerror = wrf - wRn;
148     Lerror = wlf - wLn;
149
150     CL = ((alpha*td*td*ki+2*alpha*td*kp)*Lerror +
151           (2*alpha*td*td*ki)*Lerror_p +
152           (alpha*td*td*ki-2*alpha*td*kp)*Lerror_pp +
153           8*CL_p -
154           (4-2*alpha*td)*CL_pp)/(2*alpha*td + 4);
155
156     CR = ((alpha*td*td*ki+2*alpha*td*kp)*Rerror +
157          (2*alpha*td*td*ki)*Rerror_p +
158          (alpha*td*td*ki-2*alpha*td*kp)*Rerror_pp +
159          8*CR_p -
160          (4-2*alpha*td)*CR_pp)/(2*alpha*td + 4);
161
162     CR_pp = CR_p;
163     CR_p = CR;
164     CL_pp = CL_p;
165     CL_p = CL;
166     Lerror_pp = Lerror_p;
167     Lerror_p = Lerror;
168     Rerror_pp = Rerror_p;
169     Rerror_p = Rerror;
170
171     PWMR = int(255.0*CR/5.15);
172     PWML = int(255.0*CL/5.15);
173
174     if (PWMR>=255) {PWMR=255;}
175     else if (PWMR<=0) {PWMR=0;}
176
177     if (PWML>=255) {PWML=255;}
178     else if (PWML<=0) {PWML=0;}
179
180     leftMotor->setSpeed(PWML);
181     leftMotor->run(FORWARD);
182     rightMotor->setSpeed(PWMR);
183     rightMotor->run(FORWARD);
184
185     L_last = L;
186     R_last = R;
187
188    Serial.print("   ");
189    Serial.print( wLn );
190    Serial.print("   ");
191    Serial.print( wRn);
192    Serial.print("   ");
193    Serial.print( PWML );
194    Serial.print("   ");
195    Serial.println(  PWMR);
196
197
198 }
```

```
1  // Arduino code for implementing POSITION
2  // and IMAGE-BASED ROBOT CONTROL
3  // Raspberry Pi sends desired wheels' angular
4  // velocities (and the angle of the pan
```

```
5  // camera when appropiate)
6
7  #include <Wire.h>
8  #include <Adafruit_MotorShield.h>
9  #include <Servo.h>
10 #include "utility/Adafruit_PWMServoDriver.h"
11 #include <math.h>
12 #include <Encoder.h>
13
14 Servo myservo;
15
16 Encoder le(2,2);
17 Encoder re(3,3);
18
19 Adafruit_MotorShield AFMS = Adafruit_MotorShield();
20 Adafruit_DCMotor *rightMotor = AFMS.getMotor(1);
21 Adafruit_DCMotor *leftMotor = AFMS.getMotor(4);
22
23 double wR;
24 double wL;
25 double wLp = 0.0;
26 double wRp = 0.0;
27 double wRn;
28 double wLn;
29 double LdVal = 0;
30 double RdVal = 0;
31 double Radius =0.05;
32 double Length =0.14;
33
34 double wdr;
35 double wdl;
36 double wdr_p=0;
37 double wdl_p=0;
38 double wrf;
39 double wlf;
40 double wrf_p=0;
41 double wlf_p=0;
42
43 int PWMR;
44 int PWML;
45
46 double CR;
47 double CR_p=0;
48 double CR_pp=0;
49 double CL;
50 double CL_p=0;
51 double CL_pp=0;
52
53 double Lerror;
54 double Lerror_p = 0;
55 double Lerror_pp = 0;
56 double Rerror;
57 double Rerror_p = 0;
58 double Rerror_pp = 0;
59
60 double kp = 0.096;
61 double ki = 0.519;
62
63 double alpha = 100;
64 double h = ki/kp;
65
66 long L;
67 long R;
68 long L_last=0;
69 long R_last=0;
70 unsigned long Time=0;
71 unsigned long sample_time=100;
72 double td=0.100;
73
74 const int NUMBER_OF_FIELDS = 3;
75 int fieldIndex = 0;
```

```
76  double values[NUMBER_OF_FIELDS];
77  int sign = 1;
78  double WR = 0;
79  double WL = 0;
80  int angle = 0;
81
82  void setup()
83  {
84    myservo.attach(9);
85
86    Serial.begin(115200);
87    AFMS.begin();
88    rightMotor->setSpeed(0);
89    leftMotor->setSpeed(0);
90    rightMotor->run(FORWARD);
91    leftMotor->run(FORWARD);
92
93    rightMotor->run(RELEASE);
94    leftMotor->run(RELEASE);
95    delay(1000);
96  }
97
98  void loop()
99  {
100
101   if( Serial.available())
102   {
103     char ch = Serial.read();
104     if(ch >= '0' && ch <= '9')
105     {
106
107       if(fieldIndex < NUMBER_OF_FIELDS)
108       {
109         values[fieldIndex] =
110             (values[fieldIndex] * 10) + (ch - '0');
111       }
112     }
113     else if (ch == ',')
114     {
115         values[fieldIndex] =
116            values[fieldIndex] * sign;
117         fieldIndex++;
118         sign = 1;
119     }
120     else if (ch== '-')
121     {
122         sign = -1;
123     }
124     else
125     {
126       values[fieldIndex] = values[fieldIndex]*sign;
127
128       WR = values[0]/100;
129       WL = values[1]/100;
130       angle = values[2];
131
132       for(int i=0;
133            i<min(NUMBER_OF_FIELDS, fieldIndex+1); i++)
134       {
135         values[i] = 0;
136       }
137       fieldIndex = 0;
138       sign = 1;
139     }
140   }
141
```

```
142
143   if(millis()-Time>sample_time)
144   {
145       Time = millis() ;
146       GetSpeed(WR,WL,angle);
147
148   }
149 }
150
151 void GetSpeed(double a,double b, int c)
152
153 {
154    Serial.print(wdl);
155    Serial.print(" ");
156    Serial.print(wdr);
157    Serial.print(" ");
158    Serial.print(wLn);
159    Serial.print(" ");
160    Serial.print(wRn);
161    Serial.print(" ");
162    Serial.print(PWML);
163    Serial.print(" ");
164    Serial.print(PWMR);
165    Serial.println(" ");
166
167
168   angle = c;
169   myservo.write(90 - angle);
170
171   L = le.read();
172     R = re.read();
173
174    LdVal = (double)( L- L_last)/ (td);
175    RdVal = (double)( R -R_last)/(td);
176
177    wL = LdVal*2*3.14159 /32;
178    wR = RdVal*2*3.14159 / 32;
179
180   wLn = (wL + wLp)/2.0;
181    wRn = (wR + wRp)/2.0;
182
183    wLp = wL;
184    wRp = wR;
185
186   wdr=a;
187   wdl=b;
188
189   if (wdr > 46) wdr=46;
190     else if (wdr < -46) wdr = -46;
191
192     if (wdl > 46) wdl=46;
193     else if (wdl < -46) wdl = -46;
194
195   //Prefilter
196     wrf = ( (td*h)*wdr + (td*h)*wdr_p - (td*h -
197            2)*wrf_p)/(2 + td*h);
198     wlf = ( (td*h)*wdl + (td*h)*wdl_p - (td*h -
199            2)*wlf_p)/(2 + td*h);
200
201     wrf_p = wrf;
202     wlf_p = wlf;
203     wdr_p = wdr;
204     wdl_p = wdl;
205
206   Rerror = wrf - wRn;
207   Lerror = wlf - wLn;
208
```

```
209
210     CL = ((alpha*td*td*ki+2*alpha*td*kp)*Lerror +
211           (2*alpha*td*td*ki)*Lerror_p +
212           (alpha*td*td*ki-2*alpha*td*kp)*Lerror_pp +
213           8*CL_p - (4-2*alpha*td)*
214           CL_pp)/(2*alpha*td + 4);
215     CR = ((alpha*td*td*ki+2*alpha*td*kp)*Rerror +
216           (2*alpha*td*td*ki)*Rerror_p +
217           (alpha*td*td*ki-2*alpha*td*kp)*Rerror_pp+
218           8*CR_p - (4-2*alpha*td)*
219           CR_pp)/(2*alpha*td + 4);
220
221     CR_pp = CR_p;
222       CR_p = CR;
223       CL_pp = CL_p;
224       CL_p = CL;
225       Lerror_pp = Lerror_p;
226       Lerror_p = Lerror;
227       Rerror_pp = Rerror_p;
228       Rerror_p = Rerror;
229
230      PWMR = int(255.0*CR/5.15);
231       PWML = int(255.0*CL/5.15);
232
233     if (PWMR>=255) {PWMR=255;}
234     else if (PWMR<0) {PWMR=0;}
235
236     if (PWML>=255) {PWML=255;}
237     else if (PWML<0) {PWML=0;}
238
239    leftMotor->setSpeed(PWML);
240    leftMotor->run(FORWARD);
241    rightMotor->setSpeed(PWMR);
242    rightMotor->run(FORWARD);
243
244     L_last=L;
245     R_last=R;
246
247 }
```

```
1  // Code for implementing
2  // CAMERA-BASED MINIMUM TIME
3  // Raspberry Pi sends desired wheels' angular
4  // velocities
5
6  #include <Wire.h>
7  #include <Adafruit_MotorShield.h>
8  #include "utility/Adafruit_PWMServoDriver.h"
9  #include <math.h>
10 #include <Encoder.h>
11 #include <Adafruit_Sensor.h>
12 #include <Adafruit_BNO055.h>
13 #include <utility/imumaths.h>
14
15 Adafruit_BNO055 bno = Adafruit_BNO055();
16
17 Encoder le(2,2);
18 Encoder re(3,3);
19
20 Adafruit_MotorShield AFMS = Adafruit_MotorShield();
21 Adafruit_DCMotor *rightMotor = AFMS.getMotor(2);
22 Adafruit_DCMotor *leftMotor = AFMS.getMotor(4);
23
24 double wR;
25 double wL;
26 double wRn;
```

```
27 double wLn;
28 double wRp = 0;
29 double wLp = 0;
30 double LdVal = 0;
31 double RdVal = 0;
32 double Radius =0.05;
33 double Length =0.14;
34
35 double theta;
36 double thetap = 0.0;
37 double angularV;
38
39 double wdr;
40 double wdl;
41 double wdr_p=0;
42 double wdl_p=0;
43 double wrf;
44 double wlf;
45 double wrf_p=0;
46 double wlf_p=0;
47
48 int PWMR;
49 int PWML;
50
51 double CR;
52 double CR_p=0;
53 double CR_pp=0;
54 double CL;
55 double CL_p=0;
56 double CL_pp=0;
57
58 double Lerror;
59 double Lerror_p = 0;
60 double Lerror_pp = 0;
61 double Rerror;
62 double Rerror_p = 0;
63 double Rerror_pp = 0;
64
65 double kp = 0.096;
66 double ki = 0.519;
67
68 double alpha = 100;
69 double h = ki/kp;
70
71 long L;
72 long R;
73 long L_last = 0;
74 long R_last = 0;
75 unsigned long Time=0;
76 unsigned long sample_time=50;
77 double td=0.050;
78
79 const int NUMBER_OF_FIELDS = 3;
80 int fieldIndex = 0;
81 double values[NUMBER_OF_FIELDS];
82 int sign = 1;
83 double WR = 0;
84 double WL = 0;
85 double etheta = 0;
86
87 void setup()
88 {
89   bno.begin();
90
91   Serial.begin(115200);
92   AFMS.begin();
93   rightMotor->setSpeed(0);
94   leftMotor->setSpeed(0);
95   rightMotor->run(FORWARD);
96   leftMotor->run(FORWARD);
```

```
97
98    rightMotor->run(RELEASE);
99    leftMotor->run(RELEASE);
100
101   delay(1000);
102 }
103
104 void loop()
105 {
106
107   if( Serial.available())
108   {
109     char ch = Serial.read();
110     if(ch >= '0' && ch <= '9')
111     {
112
113       if(fieldIndex < NUMBER_OF_FIELDS)
114       {
115         values[fieldIndex]=(values[fieldIndex]*10)+
116         (ch - '0');
117       }
118     }
119     else if (ch == ',')
120     {
121         values[fieldIndex] = values[fieldIndex] * sign;
122         fieldIndex++;
123         sign = 1;
124     }
125     else if (ch== '-')
126     {
127         sign = -1;
128     }
129     else
130     {
131
132       values[fieldIndex] = values[fieldIndex] * sign;
133
134       WR = values[0]/100;
135       WL = values[1]/100;
136       etheta = values[2]/100;
137
138
139       for(int i=0; i < min(NUMBER_OF_FIELDS,
140                fieldIndex+1); i++)
141       {
142
143         values[i] = 0;
144       }
145       fieldIndex = 0;
146       sign = 1;
147
148     }
149   }
150
151
152   if(millis()-Time>sample_time)
153   {
154       Time = millis() ;
155       GetSpeed(WR,WL);
156
157   }
158 }
159
160 void GetSpeed(double a,double b)
161 {
162    Serial.print(thetap);
```

```
163     Serial.print(" ");
164     Serial.print(etheta);
165     Serial.print(" ");
166     Serial.print(wdl);
167     Serial.print(" ");
168     Serial.print(wdr);
169     Serial.print(" ");
170     Serial.print(wLn);
171     Serial.print(" ");
172     Serial.print(wRn);
173     Serial.print(" ");
174     Serial.print(PWML);
175     Serial.print(" ");
176     Serial.print(PWMR);
177     Serial.println(" ");
178
179
180     L = le.read();
181     R = re.read();
182
183     LdVal = (double)( L- L_last)/ (td);
184     RdVal = (double)( R -R_last)/(td);
185
186     wL = LdVal*2*3.14159 /32;
187     wR = RdVal*2*3.14159 / 32;
188
189     wLn = (wL + wLp)/2.0;
190     wRn = (wR + wRp)/2.0;
191
192     wLp = wL;
193     wRp = wR;
194
195    wdr=a;
196    wdl=b;
197
198      if (wdr > 46) wdr=46;
199      else if (wdr < -46) wdr = -46;
200
201      if (wdl > 46) wdl=46;
202      else if (wdl < -46) wdl = -46;
203
204      //Prefilter
205      wrf = ( (td*h)*wdr + (td*h)*wdr_p - (td*h -
206             2)*wrf_p)/(2 + td*h);
207      wlf = ( (td*h)*wdl + (td*h)*wdl_p - (td*h -
208               2)*wlf_p)/(2 + td*h);
209
210      wrf_p = wrf;
211      wlf_p = wlf;
212      wdr_p = wdr;
213      wdl_p = wdl;
214
215      Rerror = wrf - wRn;
216      Lerror = wlf - wLn;
217
218     CL = ((alpha*td*td*ki+2*alpha*td*kp)*Lerror +
219           (2*alpha*td*td*ki)*Lerror_p +
220           (alpha*td*td*ki-2*alpha*td*kp)*Lerror_pp +
221           8*CL_p - (4-2*alpha*td)*
222           CL_pp)/(2*alpha*td + 4);
223     CR = ((alpha*td*td*ki+2*alpha*td*kp)*Rerror +
224           (2*alpha*td*td*ki)*Rerror_p +
225           (alpha*td*td*ki-2*alpha*td*kp)*Rerror_pp +
226           8*CR_p - (4-2*alpha*td)*
227           CR_pp)/(2*alpha*td + 4);
228
229     CR_pp = CR_p;
```

```
      CR_p = CR;
      CL_pp = CL_p;
      CL_p = CL;
      Lerror_pp = Lerror_p;
      Lerror_p = Lerror;
      Rerror_pp = Rerror_p;
      Rerror_p = Rerror;

    PWMR = int(255.0*CR/5.15);
     PWML = int(255.0*CL/5.15);

   if (PWMR>=255) {PWMR=255;}
   else if (PWMR<0) {PWMR=0;}

   if (PWML>=255) {PWML=255;}
   else if (PWML<0) {PWML=0;}

     leftMotor->setSpeed(PWML);
     leftMotor->run(FORWARD);
     rightMotor->setSpeed(PWMR);
     rightMotor->run(FORWARD);

   L_last=L;
   R_last=R;

  imu::Vector<3> gyro =
      bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);
  angularV = double(sin(0.2516)*gyro.y() +
                cos(0.2516)*gyro.z());
  theta = angularV*td + thetap;
  thetap = theta;

}
```

```
// Code for implementing
// NONCAMERA-BASED MINIMUM TIME
// Outer and inner loop functionalities
// implemented in arduino
// Sampling rate for the outer loop is half
// of the inner loop sampling rate

#include <SPI.h>
#include <Wire.h>
#include <Adafruit_MotorShield.h>
#include "utility/Adafruit_PWMServoDriver.h"
#include <math.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
#include <Encoder.h>


Adafruit_BNO055 bno = Adafruit_BNO055();

//Create the motor shield object with the default I2C address
Adafruit_MotorShield AFMS = Adafruit_MotorShield();
//Or, create it with a different I2C address (say for stacking)
//Adafruit_MotorShield AFMS = Adafruit_MotorShield(0x61);

Adafruit_DCMotor *rightMotor = AFMS.getMotor(2);
Adafruit_DCMotor *leftMotor = AFMS.getMotor(4);

Encoder le(2, 2);
Encoder re(3, 3);

int i = 0;
```

```
33
34 double wR;
35 double wL;
36 double wRn;
37 double wLn;
38 double wRp = 0.0;
39 double wLp = 0.0;
40 double LdVal = 0;
41 double RdVal = 0;
42 double Radius = 0.05;
43 double Length = 0.14;
44
45 double vd;
46 double thetad;
47
48 //coefficients for thetad
49 double c11 = -1.078494640028155 , c12 = 0.589828409894448,
50        c13 = -0.028744221309861, c14 = -0.000678949378683;
51 double c21 = 2.033372269393586, c22 = -8.346480116404441,
52        c23 = 12.046173786017038, c24 = -7.191684895278207,
53        c25 = 1.494624926389315;
54 double c31 = 0.012500371135158, c32 = -0.159687067816037,
55        c33 = 0.715195483299841, c34 = -0.262371325614063,
56        c35 = -0.787878835452174;
57 double c41 = 5.381935271613, c42 = -102.140449409786,
58        c43 = 724.621486554530, c44 = -2276.984449879569,
59        c45 = 2676.560312474572;
60 double c51 = -1.2560789229968, c52 = 21.4218449225788,
61        c53 = -121.2496356598690, c54 = 231.0526945780721;
62 double c61 = -1.0404967372400 , c62 = 19.0017252985985,
63        c63 = -115.6505979459060, c64 = 237.7442418699383;
64 double c71 = 0.001924563252541, c72 = -0.056484304203718,
65        c73 = 0.574924248356445, c74 = -1.228299116582199,
66        c75 = -1.345608340824477;
67
68 //vd polynomial
69 double q11 = 4002938.341422841, q12 = -786360.729838765,
70        q13 = 57850.723212587, q14 = -1986.346301998,
71        q15 = 33.002467506, q16 = 0.248241748;
72
73
74 double linearV;
75 double angularV;
76 double angularV_p = 0;
77 double theta;
78 double thetap = 0;
79 double distance;
80 double distance_p = 0;
81
82 double wdr;
83 double wdl;
84 double wdr_p = 0;
85 double wdl_p = 0;
86 double wrf;
87 double wlf;
88 double wrf_p = 0;
89 double wlf_p = 0;
90
91 double CR;
92 double CR_p = 0;
93 double CR_pp = 0;
94 double CL;
95 double CL_p = 0;
96 double CL_pp = 0;
97
98 double Lerror;
99 double Lerror_p = 0;
100 double Lerror_pp = 0;
101 double Rerror;
102 double Rerror_p = 0;
103 double Rerror_pp = 0;
```

```
104
105 int PWMR;
106 int PWML;
107
108 double okp = 0.869;
109 double okd = 0.396;
110 double wd;
111 double wdp = 0;
112 double wdpp = 0;
113 double wdppp = 0;
114 double thetae;
115 double thetaep = 0;
116 double thetaepp = 0;
117 double thetaeppp = 0;
118 double alphao = 100;
119
120
121 double ikp = 0.096;
122 double iki = 0.519;
123
124 double h = iki / ikp;
125
126 long L;
127 long R;
128 long L_last = 0;
129 long R_last = 0;
130 unsigned long Time = 0;
131 unsigned long sample_time = 100;
132 double td = 0.100;
133 double to = td*2;
134 double XX;
135
136 void setup()
137 {
138   bno.begin();
139
140   AFMS.begin();
141
142
143   leftMotor->setSpeed(0);
144   rightMotor->setSpeed(0);
145   leftMotor->run(FORWARD);
146   rightMotor->run(FORWARD);
147
148   leftMotor->run(RELEASE);
149   rightMotor->run(RELEASE);
150
151   delay(1000);
152
153
154   Serial.begin(115200);
155 }
156
157 void loop()
158 {
159
160   if (distance_p > 10.2202)
161   {
162     leftMotor->setSpeed(0);
163     leftMotor->run(FORWARD);
164     rightMotor->setSpeed(0);
165     rightMotor->run(FORWARD);
166
167   }
168
169
170   else
171   {
172     Outer_loop();
```

```
173
174       for (i = 0; i < 2; i++)
175       {
176         if (millis() - Time > sample_time)
177         {
178           Time = millis();
179           Inner_loop();
180         }
181
182         else
183         {
184           i = i - 1;
185         }
186       }
187
188       Update();
189     }
190   }
191
192
193   void Outer_loop()
194   {
195     XX = distance_p;
196     thetad = c11 * XX * XX * XX + c12 * XX * XX + c13 * XX + c14;
197
198     if (XX >  0.4190 && XX <= 1.7891)
199       thetad = c21 * XX * XX * XX * XX + c22 * XX * XX * XX +
200                c23 * XX * XX + c24 * XX + c25;
201     else if (XX > 1.7891 &&  XX <= 4.3813)
202       thetad = c31 * XX * XX * XX * XX + c32 * XX * XX * XX +
203                c33 * XX * XX + c34 * XX + c35;
204     else if (XX > 4.3813 &&  XX <= 4.9922)
205       thetad = c41 * XX * XX * XX * XX + c42 * XX * XX * XX +
206                c43 * XX * XX + c44 * XX + c45;
207     else if (XX > 4.9922 &&  XX <= 5.6083)
208       thetad = c51 * XX * XX * XX + c52 * XX * XX +
209                c53 * XX + c54;
210     else if (XX > 5.6083 &&  XX <= 6.6083)
211       thetad = c61 * XX * XX * XX + c62 * XX * XX +
212                c63 * XX + c64;
213     else if (XX > 6.6083 )
214       thetad = c71 * XX * XX * XX * XX + c72 * XX * XX * XX +
215                c73 * XX * XX + c74 * XX + c75;
216
217     thetae = thetad - thetap;
218
219     wd = (   (2 * okd * alphao * alphao * to * to +
220              okp * alphao * alphao * to * to * to) * thetae +
221              (2 * okd * alphao * alphao * to * to +
222              3 * okp * alphao * alphao * to * to * to) * thetaep +
223              (3 * okp * alphao * alphao * to * to * to -
224              2 * okd * alphao * alphao * to * to) * thetaepp +
225              (okp * alphao * alphao * to * to * to -
226              2 * okd * alphao * alphao * to * to) * thetaeppp -
227              (to * (3 * alphao * alphao * to * to +
228              4 * alphao * to - 4)) * wdp -
229              (to * (3 * alphao * alphao * to * to -
230              2 * alphao * to - 4)) * wdpp -
231              (to * (alphao * alphao * to * to -
232              2 * alphao * to + 4)) * wdppp    ) / (to * (4 +
233              4 * alphao * to + alphao * alphao * to * to));
234
235   }
236
237   void Inner_loop()
238   {
239     XX = distance_p;
```

```
240     //
241     vd = q11 * XX * XX * XX * XX * XX + q12 * XX * XX * XX * XX +
242           q13 * XX * XX * XX + q14 * XX * XX + q15 * XX + q16;
243
244     if (XX > 0.0690)
245       vd = 0.5;
246
247     Serial.print(distance_p);
248     Serial.print(" ");
249     Serial.print(thetad);
250     Serial.print(" ");
251     Serial.print(vd);
252     Serial.print(" ");
253     Serial.print(thetap);
254     Serial.print(" ");
255     Serial.print(linearV);
256     Serial.print(" ");
257     Serial.print(PWML);
258     Serial.print(" ");
259     Serial.println(PWMR);
260
261     wdr = (2 * vd - Length * wd) / (2 * Radius);
262     wdl = (2 * vd + Length * wd) / (2 * Radius);
263
264     if (wdr > 46) wdr = 46;
265     else if (wdr < -46) wdr = -46;
266
267
268     if (wdl > 46) wdl = 46;
269     else if (wdl < -46) wdl = -46;
270
271     //Prefilter z/(s+z)
272     wrf = ( (td * h) * wdr + (td * h) * wdr_p - (td * h -
273               2) * wrf_p) / (2 + td * h);
274     wlf = ( (td * h) * wdl + (td * h) * wdl_p - (td * h -
275               2) * wlf_p) / (2 + td * h);
276
277     wrf_p = wrf;
278     wlf_p = wlf;
279     wdr_p = wdr;
280     wdl_p = wdl;
281
282
283
284     L = le.read();
285     R = re.read();
286
287     LdVal = (double)( L - L_last) / (td);
288     RdVal = (double)( R - R_last) / (td);
289
290     wL = LdVal * 2 * 3.1416 / 32; //32
291     wR = RdVal * 2 * 3.1416 / 32;
292
293     wLn = (wL + wLp) / 2.0;
294     wRn = (wR + wRp) / 2.0;
295
296     wLp = wL;
297     wRp = wR;
298
299     linearV = (wRn * Radius + wLn * Radius) / 2;
300
301     Rerror = wrf - wRn;
302     Lerror = wlf - wLn;
303
304     CL = ((100 * td * td * iki + 200 * td * ikp)*Lerror+
305           (200*td*td*iki)*Lerror_p+(100*td*td*iki-
306            200 * td * ikp) * Lerror_pp + 8 * CL_p -
```

```
307           (4 - 200 * td) * CL_pp) / (200 * td + 4);
308   CR = ((100 * td * td * iki + 200 * td * ikp)*Rerror+
309         (200*td*td*iki)*Rerror_p+(100*td*td*iki-
310         200 * td * ikp) * Rerror_pp + 8 * CR_p -
311         (4 - 200 * td) * CR_pp) / (200 * td + 4);
312
313   CR_pp = CR_p;
314   CR_p = CR;
315   CL_pp = CL_p;
316   CL_p = CL;
317   Lerror_pp = Lerror_p;
318   Lerror_p = Lerror;
319   Rerror_pp = Rerror_p;
320   Rerror_p = Rerror;
321
322   PWMR = int(255.0 * CR / 5.15);
323   PWML = int(255.0 * CL / 5.15);
324
325   if (PWMR >= 255) {
326     PWMR = 255;
327   }
328   else if (PWMR <= 0) {
329     PWMR = 0;
330   }
331
332   if (PWML >= 255) {
333     PWML = 255;
334   }
335   else if (PWML <= 0) {
336     PWML = 0;
337   }
338
339   leftMotor->setSpeed(PWML);
340   leftMotor->run(FORWARD);
341   rightMotor->setSpeed(PWMR);
342   rightMotor->run(FORWARD);
343
344   L_last = L;
345   R_last = R;
346
347   distance = td * linearV + distance_p;
348   distance_p = distance;
349
350 }
351
352
353 void Update()
354 {
355
356   wdppp = wdpp;
357   wdpp = wdp;
358   wdp = wd;
359   thetaeppp = thetaepp;
360   thetaepp = thetaep;
361   thetaep = thetae;
362
363   imu::Vector<3> gyro = bno.getVector
364             (Adafruit_BNO055::VECTOR_GYROSCOPE);
365   angularV = double(sin(0.2516) * gyro.y() +
366                 cos(0.2516) * gyro.z());
367   theta = angularV * to + thetap;
368   thetap = theta;
369
370 }
```

# APPENDIX C

# RASPBERRY PI PYTHON CODE

```
# IMAGE BASED robot control using 1 blue dot on
# the target

import cv2
from numpy import linalg as LA
import numpy as np
import io
import picamera
import serial
import matplotlib.pyplot as plt
import pylab as plab
import time


ser = serial.Serial('/dev/ttyACM0', 115200)
ser.write('0,0 \n')

def getImage():

        cap.capture(stream, format = 'jpeg', \
                    use_video_port = True)
        frame = np.fromstring(stream.getvalue(), \
                              dtype = np.uint8)
        stream.seek(0)
        frame = cv2.imdecode(frame,1)
        return frame

datau1 = []
datav1 = []

u1_error = []
v1_error = []

j = 0

end = '\n'
comma = ','

fs_u = 327.2677
fs_v = 326.8835
z = 0.40
o_x = 152
o_y = 120
b_Z_c = 0.10
R = 0.05
L = 0.14

gain = 0.75

cap = picamera.PiCamera()
cap.vflip = True
cap.hflip = True
cap.resolution = (320,240)
cap.contrast = 0
cap.saturation = 0


stream = io.BytesIO()

frame = cv2.imread("/home/pi/visual_servoing/1dot.jpg")

src_hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)


#blue  (day)
#lower_yellow = np.array([91,82,46])
#upper_yellow = np.array([111,255,255])

#blue (night)
```

```
lower_yellow = np.array([45,90,27])
upper_yellow = np.array([99,255,71])

output1 = cv2.inRange(src_hsv, lower_yellow,\
                      upper_yellow)

erode = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,\
                                  (3,3))
dilate = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,\
                                   (8,8))
output1 = cv2.erode(output1, erode, iterations = 1)
output1 = cv2.dilate(output1, dilate, iterations = 1)

_,contours,_ = cv2.findContours(output1,cv2.RETR_TREE,\
                                cv2.CHAIN_APPROX_SIMPLE)

if len(contours) == 1:

        cv2.drawContours(frame, contours, -1, \
                         (0,255,0), 2)

        m1 = cv2.moments(contours[0])
        u1d = int(m1['m10']/m1['m00'])
        v1d = int(m1['m01']/m1['m00'])

        cv2.putText(frame,"(1)"+str(u1d)+","+str(v1d), \
                    (u1d,v1d+30), 1, 1, (0,255,0), 2, 8)

        u1d = u1d - o_x
        v1d = v1d - o_y

cv2.imshow('desired frame',frame)

Int_matrix = np.matrix([ [-fs_u/z,0,u1d/z ,u1d*v1d/fs_u , \
                          -(fs_u + u1d*u1d/fs_u) , v1d ],\
                         [0, -fs_v/z , v1d/z , \
                          fs_v + v1d*v1d/fs_v , \
                          -u1d*v1d/fs_v , -u1d ]  ])
Robot_jacobian = np.matrix([ [0, b_Z_c], [0,0], [1,0], \
                             [0,0], [0,1], [0,0]  ])
Wheels_matrix = np.matrix([ [R/2 , R/2], [-R/L, R/L]   ])
Composite_matrix = Int_matrix*Robot_jacobian*Wheels_matrix
Comp_inverse = LA.pinv(Composite_matrix)

while(1):
    frame = getImage()

    src_hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    output2 = cv2.inRange(src_hsv, lower_yellow,\
                          upper_yellow)

    output2 = cv2.erode(output2, erode, iterations = 1)
    output2 = cv2.dilate(output2, dilate, iterations = 1)

    cv2.imshow('eroded',output2)

    _,contours, _ = cv2.findContours(output2,\
                                     cv2.RETR_TREE,\
                                cv2.CHAIN_APPROX_SIMPLE)

    if len(contours) == 1:

        cv2.drawContours(frame, contours, -1, \
                         (0,255,0), 2)

```

```
136             m1 = cv2.moments(contours[0])
137             u1 = int(m1['m10']/m1['m00'])
138             v1 = int(m1['m01']/m1['m00'])
139 
140             datau1.insert(j,u1)
141             datav1.insert(j,v1)
142 
143             cv2.putText(frame,"(1)"+str(u1)+","+str(v1),\
144                         (u1,v1+30), 1, 1, (0,255,0), 2, 8)
145 
146             u1 = u1 - o_x
147             v1 = v1 - o_y
148 
149             ue1 = u1d - u1
150             ve1 = v1d - v1
151 
152             u1_error.insert(j,ue1)
153             v1_error.insert(j,ve1)
154 
155             error_vector = np.matrix([[ue1],[ve1] ])
156 
157             wheels_angular_velocity = gain*Comp_inverse*\
158                                       error_vector
159             wr = int(100*round(wheels_angular_velocity.\
160                                item(0),2))
161             wl = int(100*round(wheels_angular_velocity.\
162                                item(1),2))
163 
164             if wr <= 150 and wl <= 150:
165                     wr = 0
166                     wl = 0
167 
168             print ue1,ve1
169 
170             if wr > 4600:
171                 wr = 4600
172             elif wr < -4600:
173                 wr = -4600
174 
175             if wl > 4600:
176                 wl = 4600
177             elif wl < -4600:
178                 wl = -4600
179 
180 
181             swr = str(wr)
182             swl = str(wl)
183             string = swr + comma + swl + end
184             ser.write(string)
185 
186             j=j+1
187 
188         else:
189             ser.write('0,0 \n')
190 
191 
192         cv2.imshow('frame',frame)
193 
194         k = cv2.waitKey(1) & 0xFF
195         if k == 27:
196             break
197 
198     cv2.destroyAllWindows()
199 
200     ser.write('0,0 \n')
201     ser.close()
202     cap.close()
```

```
# IMAGE BASED robot control using 2 blue dots on
# the target

import cv2
from numpy import linalg as LA
import numpy as np
import io
import picamera
import serial
import matplotlib.pyplot as plt
import pylab as plab
import time


ser = serial.Serial('/dev/ttyACM0', 115200)
ser.write('0,0 \n')

def getImage():

        cap.capture(stream, format = 'jpeg', \
                    use_video_port = True)
        frame = np.fromstring(stream.getvalue(), \
                              dtype = np.uint8)
        stream.seek(0)
        frame = cv2.imdecode(frame,1)
        return frame

datau1 = []
datav1 = []
datau2 = []
datav2 = []
datawl = []
datawr = []

u1_error = []
v1_error = []
u2_error = []
v2_error = []

j = 0

end = '\n'
comma = ','

fs_u = 327.267
fs_v = 326.883
z = 40.0
o_x = 152
o_y = 120
b_Z_c = 10.0
R = 5.0
L = 14.0

gain = 0.75

cap = picamera.PiCamera()
cap.vflip = True
cap.hflip = True
cap.resolution = (320,240)
cap.contrast = 0
cap.saturation = 0

stream = io.BytesIO()

frame=cv2.imread("/home/pi/visual_servoing/2dot.jpg")

src_hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

#blue  day
```

```
#lower_yellow = np.array([93,122,58])
#upper_yellow = np.array([111,255,255])

#blue  night
lower_yellow = np.array([45,90,27])
upper_yellow = np.array([99,255,71])

output1 = cv2.inRange(src_hsv, lower_yellow, \
                      upper_yellow)

erode = cv2.getStructuringElement(cv2.\
                                  MORPH_ELLIPSE,(3,3))
dilate = cv2.getStructuringElement(cv2.\
                                   MORPH_ELLIPSE,(8,8))
output1 = cv2.erode(output1, erode, iterations = 1)
output1 = cv2.dilate(output1, dilate, iterations = 1)

_,contours,_=cv2.findContours(output1,\cv2.RETR_TREE,\
                          cv2.CHAIN_APPROX_SIMPLE)

if len(contours) == 2:

        cv2.drawContours(frame, contours, -1, \
                         (0,255,0), 2)

        m1 = cv2.moments(contours[0])
        u1d = int(m1['m10']/m1['m00'])
        v1d = int(m1['m01']/m1['m00'])

        m2 = cv2.moments(contours[1])
        u2d = int(m2['m10']/m2['m00'])
        v2d = int(m2['m01']/m2['m00'])

        #Reorder points
        for i in range(2):
            if u1d > u2d:
                temp = u1d
                u1d = u2d
                u2d = temp
                temp = v1d
                v1d = v2d
                v2d = temp

        cv2.putText(frame,"(1)"+str(u1d)+","+\
                    str(v1d), (u1d,v1d+30), 1,\
                    1, (0,255,0), 2, 8)
        cv2.putText(frame,"(2)"+str(u2d)+","+\
                    str(v2d), (u2d,v2d+30), 1,\
                    1, (0,255,0), 2, 8)

        u1d = u1d - o_x
        u2d = u2d - o_x

        v1d = v1d - o_y
        v2d = v2d - o_y

cv2.imshow('desired frame',frame)

Int_matrix = np.matrix([ [-fs_u/z, 0, u1d/z , \
                          u1d*v1d/fs_u , \
                          -(fs_u + u1d*u1d/fs_u) \
                          , v1d ], [0, -fs_v/z , \
                                    v1d/z , fs_v +\
                                    v1d*v1d/fs_v , \
                                    -u1d*v1d/fs_v ,\
                                    -u1d ] , \
                         [-fs_u/z, 0, u2d/z , \
```

```
                                u2d*v2d/fs_u  , -(fs_u +\
                                      u2d*u2d/fs_u) , v2d ],\
                               [0, -fs_v/z , v2d/z , \
                                fs_v + v2d*v2d/fs_v ,\
                               -u2d*v2d/fs_v , -u2d ] ])
Robot_jacobian = np.matrix([ [0, b_Z_c], [0,0], [1,0],\
                                  [0,0], [0,1], [0,0]  ])
Wheels_matrix = np.matrix([ [R/2 , R/2], [-R/L, R/L]    ])
Composite_matrix = Int_matrix*Robot_jacobian*Wheels_matrix
Comp_inverse = LA.pinv(Composite_matrix)

while(1):

    frame = getImage()

    src_hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    output2 = cv2.inRange(src_hsv, lower_yellow,\
                          upper_yellow)

    output2 = cv2.erode(output2, erode, iterations = 1)
    output2 = cv2.dilate(output2, dilate, iterations = 1)

    cv2.imshow('eroded',output2)
    _,contours, _ = cv2.findContours(output2,cv2.\
                                     RETR_TREE,cv2.\
                                     CHAIN_APPROX_SIMPLE)

    if len(contours) == 2:

        cv2.drawContours(frame, contours, -1,\
                         (0,255,0), 2)

        m1 = cv2.moments(contours[0])
        u1 = int(m1['m10']/m1['m00'])
        v1 = int(m1['m01']/m1['m00'])

        m2 = cv2.moments(contours[1])
        u2 = int(m2['m10']/m2['m00'])
        v2 = int(m2['m01']/m2['m00'])

        #Reorder points
        for i in range(2):
            if u1 > u2:
                temp = u1
                u1 = u2
                u2 = temp
                temp = v1
                v1 = v2
                v2 = temp

        datau1.insert(j,u1)
        datav1.insert(j,v1)
        datau2.insert(j,u2)
        datav2.insert(j,v2)

        cv2.putText(frame,"(1)"+str(u1)+","+str(v1),\
                    (u1,v1+30), 1, 1, (0,255,0), 2, 8)
        cv2.putText(frame,"(2)"+str(u2)+","+str(v2),\
                    (u2,v2+30), 1, 1, (0,255,0), 2, 8)

        u1 = u1 - o_x
        u2 = u2 - o_x

        v1 = v1 - o_y
        v2 = v2 - o_y

```

```
204             ue1 = u1d - u1
205             ve1 = v1d - v1
206             ue2 = u2d - u2
207             ve2 = v2d - v2
208 
209             u1_error.insert(j,ue1)
210             v1_error.insert(j,ve1)
211             u2_error.insert(j,ue2)
212             v2_error.insert(j,ve2)
213 
214             error_vector = np.matrix([[ue1],[ve1], \
215                                       [ue2],[ve2] ])
216 
217             wheels_angular_velocity = gain*\
218                                       Comp_inverse*error_vector
219             wr = int(100*round(wheels_angular_velocity.\
220                                item(0),2))
221             wl = int(100*round(wheels_angular_velocity.\
222                                item(1),2))
223 
224             if wr <= 150 and wl <= 150:
225                     wr = 0
226                     wl = 0
227 
228             print ue1,ve1,ue2,ve2
229 
230             if wr > 4600:
231                 wr = 4600
232             elif wr < -4600:
233                 wr = -4600
234 
235             if wl > 4600:
236                 wl = 4600
237             elif wl < -4600:
238                 wl = -4600
239 
240             swr = str(wr)
241             swl = str(wl)
242             string = swr + comma + swl + end
243             ser.write(string)
244 
245             j=j+1
246 
247         else:
248             ser.write('0,0 \n')
249 
250 
251         cv2.imshow('frame',frame)
252 
253 
254         k = cv2.waitKey(1) & 0xFF
255         if k == 27:
256             break
257 
258 cv2.destroyAllWindows()
259 
260 ser.write('0,0 \n')
261 ser.close()
262 cap.close()
```

```
1 # IMAGE BASED robot control using 3 blue dots on
2 # the target
3 
4 import cv2
5 from numpy import linalg as LA
6 import numpy as np
7 import io
```

```
8  import picamera
9  import serial
10 import matplotlib.pyplot as plt
11 import pylab as plab
12 import time
13
14
15 ser = serial.Serial('/dev/ttyACM0', 115200)
16 ser.write('0,0 \n')
17
18 def getImage():
19
20         cap.capture(stream, format = 'jpeg', \
21                     use_video_port = True)
22         frame = np.fromstring(stream.getvalue(), \
23                               dtype = np.uint8)
24         stream.seek(0)
25         frame = cv2.imdecode(frame,1)
26         return frame
27
28 datau1 = []
29 datav1 = []
30 datau2 = []
31 datav2 = []
32 datau3 = []
33 datav3 = []
34
35 u1_error = []
36 v1_error = []
37 u2_error = []
38 v2_error = []
39 u3_error = []
40 v3_error = []
41 j = 0
42
43 end = '\n'
44 comma = ','
45
46 fs_u = 327.267
47 fs_v = 326.883
48 z = 40.0
49 o_x = 152
50 o_y = 120
51 b_Z_c = 10.0
52 R = 5.0
53 L = 14.0
54
55 gain = 0.75
56
57 cap = picamera.PiCamera()
58 cap.vflip = True
59 cap.hflip = True
60 cap.resolution = (320,240)
61 cap.contrast = 0
62 cap.saturation = 0
63
64 stream = io.BytesIO()
65
66 frame = cv2.imread("/home/pi/visual_servoing/3dot.jpg")
67
68 src_hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
69
70 #blue  day
71 #lower_yellow = np.array([91,82,46])
72 #upper_yellow = np.array([111,255,255])
73
74 #blue  night
75 lower_yellow = np.array([45,90,27])
```

```
upper_yellow = np.array([99,255,71])

output1 = cv2.inRange(src_hsv, lower_yellow, \
                      upper_yellow)

erode = cv2.getStructuringElement(cv2.\
                                  MORPH_ELLIPSE,(3,3))
dilate = cv2.getStructuringElement(cv2.\
                                   MORPH_ELLIPSE,(8,8))
output1 = cv2.erode(output1, erode, iterations = 1)
output1 = cv2.dilate(output1, dilate, iterations = 1)

_,contours, _ = cv2.findContours(output1,cv2.\
                                 RETR_TREE,cv2.\
                                 CHAIN_APPROX_SIMPLE)

if len(contours) == 3:

        cv2.drawContours(frame, contours, -1, \
                         (0,255,0), 2)

        m1 = cv2.moments(contours[0])
        u1d = int(m1['m10']/m1['m00'])
        v1d = int(m1['m01']/m1['m00'])

        m2 = cv2.moments(contours[1])
        u2d = int(m2['m10']/m2['m00'])
        v2d = int(m2['m01']/m2['m00'])

        m3 = cv2.moments(contours[2])
        u3d = int(m3['m10']/m3['m00'])
        v3d = int(m3['m01']/m3['m00'])

        #Reorder points
        for i in range(3):
            if u1d > u2d:
                temp = u1d
                u1d = u2d
                u2d = temp
                temp = v1d
                v1d = v2d
                v2d = temp
            if u2d > u3d:
                temp = u2d
                u2d = u3d
                u3d = temp
                temp = v2d
                v2d = v3d
                v3d = temp

        cv2.putText(frame,"(1)"+str(u1d)+","+\
                    str(v1d), (u1d,v1d+30), 1, \
                    1, (0,255,0), 2, 8)
        cv2.putText(frame,"(2)"+str(u2d)+","+\
                    str(v2d), (u2d,v2d+30), 1,\
                    1, (0,255,0), 2, 8)
        cv2.putText(frame,"(3)"+str(u3d)+","+\
                    str(v3d), (u3d-30,v3d+60), \
                    1, 1, (0,255,0), 2, 8)

        u1d = u1d - o_x
        u2d = u2d - o_x
        u3d = u3d - o_x
        v1d = v1d - o_y
        v2d = v2d - o_y
        v3d = v3d - o_y

cv2.imshow('desired frame',frame)
```

```
144
145 Int_matrix = np.matrix([ [-fs_u/z, 0, u1d/z , \
146                           u1d*v1d/fs_u , -(fs_u \
147                         + u1d*u1d/fs_u) , v1d ],\
148                          [0, -fs_v/z , v1d/z , \
149                           fs_v + v1d*v1d/fs_v ,\
150                           -u1d*v1d/fs_v , -u1d ] , \
151                          [-fs_u/z, 0, u2d/z , \
152                           u2d*v2d/fs_u , -(fs_u +\
153                         u2d*u2d/fs_u) , v2d ], [0, \
154                         -fs_v/z , v2d/z , fs_v + \
155                         v2d*v2d/fs_v , -u2d*v2d/fs_v ,\
156                         -u2d ], [-fs_u/z, 0, u3d/z , \
157                         u3d*v3d/fs_u , -(fs_u + \
158                         u3d*u3d/fs_u) , v3d ], [0, \
159                         -fs_v/z , v3d/z , fs_v + \
160                                 v3d*v3d/fs_v , \
161                         -u3d*v3d/fs_v , -u3d ] ])
162 Robot_jacobian = np.matrix([ [0, b_Z_c], [0,0], [1,0],\
163                              [0,0], [0,1], [0,0] ])
164 Wheels_matrix = np.matrix([ [R/2 , R/2], [-R/L, R/L] ])
165 Composite_matrix = Int_matrix*Robot_jacobian*\
166                    Wheels_matrix
167 Comp_inverse = LA.pinv(Composite_matrix)
168
169 while(1):
170
171     frame = getImage()
172
173     src_hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
174
175     output2 = cv2.inRange(src_hsv, lower_yellow,\
176                           upper_yellow)
177
178     output2 = cv2.erode(output2, erode, iterations = 1)
179     output2 = cv2.dilate(output2, dilate, iterations = 1)
180
181     cv2.imshow('eroded',output2)
182     _,contours, _ = cv2.findContours(output2,cv2.\
183                                      RETR_TREE,cv2.\
184                                      CHAIN_APPROX_SIMPLE)
185
186     if len(contours) == 3:
187
188         cv2.drawContours(frame, contours, -1, \
189                          (0,255,0), 2)
190
191         m1 = cv2.moments(contours[0])
192         u1 = int(m1['m10']/m1['m00'])
193         v1 = int(m1['m01']/m1['m00'])
194
195         m2 = cv2.moments(contours[1])
196         u2 = int(m2['m10']/m2['m00'])
197         v2 = int(m2['m01']/m2['m00'])
198
199         m3 = cv2.moments(contours[2])
200         u3 = int(m3['m10']/m3['m00'])
201         v3 = int(m3['m01']/m3['m00'])
202
203         #Reorder points
204         for i in range(2):
205             if u1 > u2:
206                 temp = u1
207                 u1 = u2
208                 u2 = temp
209                 temp = v1
```

```
                v1 = v2
                v2 = temp
            if u2 > u3:
                temp = u2
                u2 = u3
                u3 = temp
                temp = v2
                v2 = v3
                v3 = temp

        datau1.insert(j,u1)
        datav1.insert(j,v1)
        datau2.insert(j,u2)
        datav2.insert(j,v2)
        datau3.insert(j,u3)
        datav3.insert(j,v3)

        cv2.putText(frame,"(1)"+str(u1)+","+str(v1),\
                    (u1,v1+30), 1, 1, (0,255,0), 2, 8)
        cv2.putText(frame,"(2)"+str(u2)+","+str(v2),\
                    (u2,v2+30), 1, 1, (0,255,0), 2, 8)
        cv2.putText(frame,"(3)"+str(u3)+","+str(v3),\
                    (u3,v3+30), 1, 1, (0,255,0), 2, 8)

        u1 = u1 - o_x
        u2 = u2 - o_x
        u3 = u3 - o_x
        v1 = v1 - o_y
        v2 = v2 - o_y
        v3 = v3 - o_y

        ue1 = u1d - u1
        ve1 = v1d - v1
        ue2 = u2d - u2
        ve2 = v2d - v2
        ue3 = u3d - u3
        ve3 = v3d - v3

        u1_error.insert(j,ue1)
        v1_error.insert(j,ve1)
        u2_error.insert(j,ue2)
        v2_error.insert(j,ve2)
        u3_error.insert(j,ue3)
        v3_error.insert(j,ve3)

        error_vector = np.matrix([[ue1],[ve1], [ue2],\
                                  [ve2], [ue3],[ve3] ])

        wheels_angular_velocity = gain*Comp_inverse*\
                                  error_vector
        wr = int(100*round(wheels_angular_velocity.\
                           item(0),2))
        wl = int(100*round(wheels_angular_velocity.\
                           item(1),2))

        if wr <= 150 and wl <= 150:
                wr = 0
                wl = 0

        print ue1,ve1,ue2,ve2,ue3,ve3

        if wr > 4600:
            wr = 4600
        elif wr < -4600:
            wr = -4600

        if wl > 4600:
            wl = 4600
        elif wl < -4600:
```

```
279                 wl = -4600
280
281             swr = str(wr)
282             swl = str(wl)
283             string = swr + comma + swl + end
284             ser.write(string)
285
286             j=j+1
287
288         else:
289             ser.write('0,0 \n')
290
291
292         cv2.imshow('frame',frame)
293
294
295         k = cv2.waitKey(1) & 0xFF
296         if k == 27:
297             break
298
299     cv2.destroyAllWindows()
300
301     ser.write('0,0 \n')
302     ser.close()
303     cap.close()
```

```
1   # POSITION BASED robot control
2   # Sends desired angular velocities
3   # for both left and right wheel to
4   # Arduino
5
6   import cv2
7   from numpy import linalg as LA
8   import numpy as np
9   import math
10  import io
11  import picamera
12  import serial
13  import time
14
15  ser = serial.Serial('/dev/ttyACM0', 115200)
16  ser.write('0,0 \n')
17
18  def getImage():
19
20      cap.capture(stream, format = 'jpeg', \
21                  use_video_port = True)
22      frame = np.fromstring(stream.getvalue(),\
23                            dtype = np.uint8)
24      stream.seek(0)
25      frame = cv2.imdecode(frame,1)
26      return frame
27
28  datax = []
29  dataz = []
30
31  j = 0
32
33  end = '\n'
34  comma = ','
35
36  mtx = np.matrix([ [327.26773097, 0.0, \
37                     152.4401473], [0.0, \
38                      326.88353638,\
39                      120.22141464], [0.0,\
40                                0.0, 1.0]    ])
41  dist = np.matrix([ [-0.0233138421, \
```

```
42                      1.14789142,  -0.000356860444,\
43                   -0.00891682674,  -5.75034097]   ])
44
45  col = 8
46  row = 6
47
48  criteria = (cv2.TERM_CRITERIA_EPS + cv2.\
49              TERM_CRITERIA_MAX_ITER, 30, 0.001)
50  objp = np.zeros( (row*col,3), np.float32 )
51  objp[:,:2] = np.mgrid[0:col,0:row].T.reshape(-1,2)
52
53  square_size = 2.8
54  objp *= square_size
55
56  axis = np.float32([ [2.8,0,0], [0,2.8,0],\
57                      [0,0,-2.8]   ]).reshape(-1,3)
58
59  b_Z_c = 10.0
60  R = 5.0
61  L = 14.0
62
63  kv = 0.4
64  kw = 0.8
65
66  cap = picamera.PiCamera()
67  cap.vflip = True
68  cap.hflip = True
69  cap.resolution = (320,240)
70  cap.contrast = 0
71  cap.saturation = 0
72
73  stream = io.BytesIO()
74
75  Wheels_matrix = np.matrix([    [R/2, R/2], \
76                                 [-R/L, R/L]   ])
77
78  xd = -9.8
79  zd = 50.0
80
81  while(1):
82
83      frame = getImage()
84
85      gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
86
87      ret, corners = cv2.findChessboardCorners(gray,\
88                                      (col,row), None)
89
90      if ret == True:
91
92          retval, rvecs, tvecs = cv2.solvePnP(objp,\
93                                  corners, mtx, dist)
94
95          imgpts, jac = cv2.projectPoints(axis, \
96                          rvecs, tvecs, mtx, dist)
97
98          corner = tuple(corners[0].ravel())
99          cv2.line(frame, corner, \
100             tuple(imgpts[0].ravel()), (255,0,0), 5)
101         cv2.line(frame, corner, \
102             tuple(imgpts[1].ravel()), (0,255,0), 5)
103         cv2.line(frame, corner, \
104             tuple(imgpts[2].ravel()), (0,0,255), 5)
105
106         x_temp = tvecs.item(0)
107         y_temp = tvecs.item(1)
108         z_temp = tvecs.item(2)
109
```

```
110             theta = -rvecs.item(1)
111
112             pos = np.matrix([   [x_temp],[z_temp]    ])
113             rot_mat = np.matrix([     [math.cos(theta),\
114                                        math.sin(theta)],\
115                                       [-math.sin(theta),\
116                                        math.cos(theta)]   ])
117             real_pos = rot_mat*pos
118
119             xcam = real_pos.item(0)
120             zcam = real_pos.item(1)
121
122             xcar = xcam + math.sin(theta)*10.0
123             zcar = zcam + math.cos(theta)*10.0
124
125             print xcar, zcar
126
127             xe = xcar - xd
128             ze = zcar - zd
129             distance = math.sqrt(xe*xe + ze*ze)
130
131             beta = math.atan2(xe,ze)
132
133             etheta = beta - theta
134
135             es = distance*math.cos(etheta)
136
137             vref = kv*es
138
139             if vref > 20.0:
140                 vref = 20.0
141
142             wref = kw*etheta
143
144             if etheta > 3.14159/2:
145                 wref = 0
146             elif etheta < -3.14159/2:
147                 wref = 0
148
149             vel = np.matrix([   [vref],[wref]   ])
150
151             wheels_angular_velocity = LA.inv(Wheels_matrix)*\
152                                       vel
153             wr = int(100*round(wheels_angular_velocity.\
154                                item(0),2))
155             wl = int(100*round(wheels_angular_velocity.\
156                                item(1),2))
157
158             if distance < 10:
159                 wr = 0
160                 wl = 0
161
162             if wr > 4600:
163                 wr = 4600
164             elif wr < -4600:
165                 wr = -4600
166
167             if wl > 4600:
168                 wl = 4600
169             elif wl < -4600:
170                 wl = -4600
171
172             datax.insert(j,round(xcar,2))
173             dataz.insert(j,round(zcar,2))
174
175             swr = str(wr)
176             swl = str(wl)
177             string = swr + comma + swl + end
178             ser.write(string)
```

```
179
180             j=j+1
181
182         else:
183             ser.write('0,0 \n')
184
185         cv2.imshow('frame', frame)
186
187         k = cv2.waitKey(1) & 0xFF
188         if k == 27:
189             break
190
191     cv2.destroyAllWindows()
192
193     ser.write('0,0 \n')
194     ser.close()
195     cap.close()
```

```
1   # POSITION BASED robot control
2   # with pan camera
3   # Sends desired angular velocities
4   # for both left and right wheel and
5   # desired angle for the pan camera
6   # to Arduino
7
8   import cv2
9   from numpy import linalg as LA
10  import numpy as np
11  import math
12  import io
13  import picamera
14  import serial
15  import time
16
17
18  ser = serial.Serial('/dev/ttyACM0', 115200)
19  ser.write('0,0,0 \n')
20
21  def getImage():
22
23      cap.capture(stream, format = 'jpeg',\
24                  use_video_port = True)
25      frame = np.fromstring(stream.getvalue(),\
26                            dtype = np.uint8)
27      stream.seek(0)
28      frame = cv2.imdecode(frame,1)
29      return frame
30
31  datax = []
32  dataz = []
33  datapan = []
34
35  j = 0
36
37  end = '\n'
38  comma = ','
39
40  mtx = np.matrix([ [327.26773097, 0.0,\
41                  152.4401473], [0.0, 326.88353638,\
42                  120.22141464], [0.0, 0.0, 1.0]    ])
43  dist = np.matrix([ [-0.0233138421, 1.14789142,\
44                  -0.000356860444, -0.00891682674,\
45                  -5.75034097]   ])
46
47  col = 8
48  row = 6
49
```

```
50  criteria = (cv2.TERM_CRITERIA_EPS + cv2.\
51              TERM_CRITERIA_MAX_ITER, 30, 0.001)
52  objp = np.zeros( (row*col,3), np.float32 )
53  objp[:,:2] = np.mgrid[0:col,0:row].T.\
54               reshape(-1,2)
55
56  square_size = 2.8
57  objp *= square_size
58
59  axis = np.float32([ [2.8,0,0], [0,2.8,0], \
60                      [0,0,-2.8]  ]).reshape(-1,3)
61
62  b_Z_c = 10.0
63  R = 5.0
64  L = 14.0
65
66  kv = 0.4
67  kw = 0.8
68
69  gain = 0.001
70  gaini = 0.003
71  alpha = 100.0
72
73  td = 0.25
74  anglep = 0
75  anglepp = 0
76  errorp = 0
77  errorpp = 0
78
79  cap = picamera.PiCamera()
80  cap.vflip = True
81  cap.hflip = True
82  cap.resolution = (320,240)
83  cap.contrast = 0
84  cap.saturation = 0
85
86  stream = io.BytesIO()
87
88  Wheels_matrix = np.matrix([    [R/2, R/2], \
89                                 [-R/L, R/L]  ])
90
91  xd = -9.8
92  zd = 50.0
93
94  while(1):
95
96      frame = getImage()
97
98      gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
99
100     ret, corners = cv2.findChessboardCorners(gray,\
101                                  (col,row), None)
102
103     if ret == True:
104
105         retval, rvecs, tvecs = cv2.solvePnP(objp,\
106                                  corners, mtx, dist)
107
108         imgpts, jac = cv2.projectPoints(axis,\
109                           rvecs, tvecs, mtx, dist)
110
111         corner = tuple(corners[0].ravel())
112         cv2.line(frame, corner, \
113             tuple(imgpts[0].ravel()), (255,0,0), 5)
114         cv2.line(frame, corner, \
115             tuple(imgpts[1].ravel()), (0,255,0), 5)
116         cv2.line(frame, corner, \
117             tuple(imgpts[2].ravel()), (0,0,255), 5)
```

```
118
119
120             corner_pos = corners[3].item(0)
121             error = corner_pos - 152.0
122
123             angle = ((alpha*td*td*gaini+\
124                       2*alpha*td*gain)*error + \
125                      (2*alpha*td*td*gaini)*errorp +\
126                      (alpha*td*td*gaini-\
127                       2*alpha*td*gain)*errorpp + \
128                      8*anglep - (4-\
129                       2*alpha*td)*anglepp)/(2*alpha*td+\
130                                            4)
131
132             if angle > 40*3.14159/180:
133                 angle = 40*3.14159/180
134             elif angle < -40*3.14159/180:
135                 angle = -40*3.14159/180
136
137             errorpp = errorp
138             errorp = error
139             anglepp = anglep
140             anglep = angle
141
142             angled = int(angle*180/3.14159)
143
144             x_temp = tvecs.item(0)
145             y_temp = tvecs.item(1)
146             z_temp = tvecs.item(2)
147
148             theta = -rvecs.item(1)
149
150             pos = np.matrix([   [x_temp],[z_temp]    ])
151
152             rot_mat = np.matrix([     [math.cos(theta),\
153                                        math.sin(theta)],\
154                                  [-math.sin(theta), \
155                                   math.cos(theta)]   ])
156             real_pos = rot_mat*pos
157
158             xcam = real_pos.item(0)
159             zcam = real_pos.item(1)
160
161             orientation_car = theta - angle
162
163             xcar = xcam + math.sin(orientation_car)*10.0
164             zcar = zcam + math.cos(orientation_car)*10.0
165
166             print round(xcar,2), round(zcar,2)
167
168             xe = xcar - xd
169             ze = zcar - zd
170             distance = math.sqrt(xe*xe + ze*ze)
171
172             beta = math.atan2(xe,ze)
173
174             etheta = beta - orientation_car
175
176             es = distance*math.cos(etheta)
177
178             vref = kv*es
179             if vref > 20.0:
180                 vref = 20.0
181
182             wref = kw*etheta
183
184             if etheta > 3.14159/2:
185                 wref = 0
```

```
186             elif etheta < -3.14159/2:
187                 wref = 0
188
189             vel = np.matrix([   [vref],[wref]   ])
190
191             wheels_angular_velocity = \
192                             LA.inv(Wheels_matrix)*vel
193             wr = int(100*round(wheels_angular_velocity.\
194                                 item(0),2))
195             wl = int(100*round(wheels_angular_velocity.\
196                                 item(1),2))
197
198             if distance < 10:
199                 wr = 0
200                 wl = 0
201
202             if wr > 4600:
203                 wr = 4600
204             elif wr < -4600:
205                 wr = -4600
206
207             if wl > 4600:
208                 wl = 4600
209             elif wl < -4600:
210                 wl = -4600
211
212             datax.insert(j,round(xcar,2))
213             dataz.insert(j,round(zcar,2))
214             datapan.insert(j,round(angled,2))
215
216             swr = str(wr)
217             swl = str(wl)
218             sangled = str(angled)
219             string = swr + comma + swl + comma + \
220                       sangled + end
221             ser.write(string)
222
223             j = j+1
224
225         else:
226             sanglep_d = str(int(anglep*180/3.14159))
227             string = '0' + comma + '0' + comma + \
228                       sanglep_d + end
229
230             ser.write(string)
231
232         cv2.imshow('frame', frame)
233
234         k = cv2.waitKey(1) & 0xFF
235         if k == 27:
236             break
237
238 cv2.destroyAllWindows()
239
240 ser.write('0,0,0 \n')
241 ser.close()
242 cap.close()
```

```
1 # CAMERA BASED MINIMUM TIME
2 # python code
3 # Sends desired angular velocities
4 # for both left and right wheel
5 # to Arduino
6
7 import cv2
8 from numpy import linalg as LA
9 import numpy as np
```

```
10 import io
11 import picamera
12 import serial
13 import matplotlib.pyplot as plt
14 import pylab as plab
15 import time
16 import math
17
18 ser = serial.Serial('/dev/ttyACM0', 115200)
19 ser.write('0,0,0 \n')
20
21 def getImage():
22
23         cap.capture(stream, format = 'jpeg', \
24                     use_video_port = True)
25         frame = np.fromstring(stream.getvalue(),\
26                               dtype = np.uint8)
27         stream.seek(0)
28         frame = cv2.imdecode(frame,1)
29         return frame
30
31 stream = io.BytesIO()
32
33 end = '\n'
34 comma = ','
35
36 j = 0
37 i = 0
38
39 R = 0.05
40 L = 0.14
41
42 kp = 1.2
43 kd = 0.5
44 td = 0.125
45 alpha = 100.0
46
47 thetae_ppp = 0.0
48 thetae_pp = 0.0
49 thetae_p = 0.0
50 wd_ppp = 0.0
51 wd_pp = 0.0
52 wd_p = 0.0
53
54 cap = picamera.PiCamera()
55 cap.vflip = True
56 cap.hflip = True
57 cap.resolution = (320,240)
58 cap.contrast = 0
59 cap.saturation = 0
60
61 erode = cv2.getStructuringElement(cv2.MORPH_RECT,\
62                                   (5,5))
63 dilate = cv2.getStructuringElement(cv2.MORPH_RECT,\
64                                    (6,6))
65
66 #day
67 lower = np.matrix([   [ 10,141,141 ], [ 161,119,154] ])
68 upper = np.matrix([   [ 30,255,255 ], [ 179,255,255] ])
69
70 #night
71 #lower = np.matrix([   [ 10,196,120 ], [ 0,160,130] ])
72 #upper = np.matrix([   [ 30,255,255 ], [ 14,255,255]])
73
74 while(1):
75
76     frame = getImage()
77
78     frame2 = frame
```

```
79
80      roi = frame[60:80, 0:320].copy()
81
82      roi2 = frame2[60:80, 0:320].copy()
83
84      gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
85
86      ret, output = cv2.threshold(gray, 70, 255, \
87                            cv2.THRESH_BINARY_INV)
88
89      output = cv2.erode(output, erode, \
90                  iterations = 1)
91      output = cv2.dilate(output, dilate, \
92                  iterations = 1)
93
94      _,contours,_= cv2.findContours(output,\
95          cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
96
97      areas = [cv2.contourArea(c) for c \
98               in contours]
99
100     if not not areas:
101
102         max_index = np.argmax(areas)
103         cnt = contours[max_index]
104
105         cv2.drawContours(roi, [cnt], 0, \
106                           (0,0,255), 2)
107
108         m1 = cv2.moments(contours[max_index])
109         u1 = int(m1['m10']/m1['m00'])
110         v1 = int(m1['m01']/m1['m00'])
111
112         error = u1 - 160.0
113         lat = error*20.1616/320.0
114
115         thetae = math.atan2(lat,30)
116
117         wd = ((2*kd*alpha*alpha*td*td+\
118                 kp*alpha*alpha*td*td*td)*\
119               thetae + (2*kd*alpha*alpha*td*td+\
120           3*kp*alpha*alpha*td*td*td)*thetae_p + \
121          (3*kp*alpha*alpha*td*td*td-\
122           2*kd*alpha*alpha*td*td)*thetae_pp +\
123           (kp*alpha*alpha*td*td*td-\
124            2*kd*alpha*alpha*td*td)*thetae_ppp - \
125           (td*(2*alpha*alpha*td*td-8)+\
126            td*(4+4*alpha*td+alpha*alpha*td*td))*wd_p\
127            -(td*(alpha*alpha*td*td-4*alpha*td+4)+\
128            td*(2*alpha*alpha*td*td-8))*wd_pp - \
129            (td*(alpha*alpha*td*td-4*alpha*td+4))\
130            *wd_ppp)/(td*(4+4*alpha*td+\
131                          alpha*alpha*td*td))
132
133         thetae_ppp = thetae_pp
134         thetae_pp = thetae_p
135         thetae_p = thetae
136         wd_ppp = wd_pp
137         wd_pp = wd_p
138         wd_p = wd
139
140         vd = vd_array.item(j)
141
142         WL = (2*vd+L*wd)/(2*R)
143         WR = (2*vd-L*wd)/(2*R)
144         WL = int(100*round(WL,2))
145         WR = int(100*round(WR,2))
```

```
146
147             print j, vd, thetae, wd
148
149             if WR > 4600:
150                 WR = 4600
151             elif WR < -4600:
152                 WR = -4600
153
154             if WL > 4600:
155                 WL = 4600
156             elif WL < -4600:
157                 WL = -4600
158
159
160             spwmr = str(WR)
161             spwml = str(WL)
162             stheta = str(int(100*thetae))
163             string = spwmr + comma + spwml + comma + \
164                      stheta + end
165             ser.write(string)
166
167
168             hsv = cv2.cvtColor(roi2, cv2.COLOR_BGR2HSV)
169
170             out2 = cv2.inRange(hsv, lower[i,:], \
171                                upper[i,:])
172
173             out2 = cv2.erode(out2, erode,\
174                              iterations = 1)
175             out2 = cv2.dilate(out2, dilate,\
176                               iterations = 1)
177
178             _, c_dot, _ = cv2.findContours(out2,\
179                  cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
180
181             if len(c_dot) >= 1:
182
183                 cv2.drawContours(roi2, c_dot, \
184                                  -1, (0,255,0), 2)
185
186                 j = j + 1
187                 i = i + 1
188
189                 if j > 3:
190                     j = 0
191                     i = 0
192
193                 if i > 1:
194                     i = 0
195
196         else:
197             ser.write('0,0,0 \n')
198
199
200         cv2.imshow('frame',frame)
201         cv2.imshow('roi', roi)
202         cv2.imshow('roi2', roi2)
203
204         k = cv2.waitKey(1) & 0xFF
205         if k == 27:
206             break
207
208 cv2.destroyAllWindows()
209
210 ser.write('0,0,0 \n')
211 ser.close()
212 cap.close()
```

```
# This python code is used to receive on the
# Raspberry Pi 2 data sent from Arduino board

import cv2
from numpy import linalg as LA
import numpy as np
import io
import picamera
import serial
import matplotlib.pyplot as plt
import pylab as plab
import time
import math

ser = serial.Serial('/dev/ttyACM0', 115200, timeout=1)


data = 'data'

text = open(data, 'w')

while(1):

    a = ser.readline()
    a = a.rstrip()

    if len(a):
        text.write(a+'\n')

    k = cv2.waitKey(1) & 0xFF
    if k == 27:
        break

cv2.destroyAllWindows()

text.close()
```

# APPENDIX D

# AMPL CODE

```
1
2  # CAMERA BASED MINIMUM TIME
3  # AMPL code
4
5  param N;   # number of integration steps
6  param ds:= 0.001;    # step size
7  set kset ordered:= 0..N;    # defining a set from 0 to N
8
9  # declaration of racetrack parameters
10
11 param x_t{kset};
12 param z_t{kset};
13 param theta_t{kset};
14 param k_t{kset};
15 param x_sh{kset};
16 param z_sh{kset};
17 param theta_sh{kset};
18
19 # declaration of robot parameters
20
21 param L;
22 param r;
23
24 # declaration of variables
25
26 var vref{kset};
27
28 var scale_f{kset};
29 var distance{kset};
30 var temp_dist_cam{kset};
31 var distance_cam{kset};
32
33 var accelr{kset};
34 var accel{kset};
35 var w{kset};
36 var ang_accel{kset};
37 var jerkr{kset};
38 var jerk{kset};
39
40 var xpos{kset};
41 var zpos{kset};
42 var x1{kset};
43 var x2{kset};
44 var x3{kset};
45 var x4{kset};
46 var x5{kset};
47 var v{kset};
48 var theta{kset};
49 var time{kset};
50
51 var xcam{kset};
52 var zcam{kset};
53 var error_ang{kset};
54
55 minimize obj: time[N];
56
57 subject to
58 ## Cruise control state space with kinematics and scale factor
59 c1{k in 1..N}: (xpos[k] - xpos[k-1])/ds =
60                          scale_f[k-1]*(v[k-1]*sin(theta[k-1]));
61 c2{k in 1..N}: (zpos[k] - zpos[k-1])/ds =
62                          scale_f[k-1]*(v[k-1]*cos(theta[k-1]));
63 c3{k in 1..N}: (x1[k] - x1[k-1])/ds = scale_f[k-1]*(-2.616*x1[k-1]-
64                                        2.489*x2[k-1]+2*vref[k-1]);
65 c4{k in 1..N}: (x2[k] - x2[k-1])/ds = scale_f[k-1]*(2*x1[k-1]);
66 c5{k in 0..N}: v[k] = 1.245*x2[k];
67
```

```
68  c6{k in 1..N}: (x3[k] - x3[k-1])/ds = scale_f[k-1]*(-2.616*x3[k-1]-
69                                        2.489*x4[k-1]+2*error_ang[k-1]);
70  c7{k in 1..N}: (x4[k] - x4[k-1])/ds = scale_f[k-1]*(2*x3[k-1]);
71  c8{k in 1..N}: (x5[k] - x5[k-1])/ds = scale_f[k-1]*(x4[k-1]);
72  c9{k in 0..N}: theta[k] = 0.6223*x4[k]+1.494*x5[k];
73
74  c14{k in 1..N}: (time[k] - time[k-1])/ds = scale_f[k-1];
75
76
77  c80{k in 0..N}: scale_f[k] = (1 - (distance[k]*k_t[k]) )/
78                                        (v[k]*cos(theta[k]-theta_t[k]));
79
80
81  ##Acceleration definition
82  c15{k in 1..N}: (vref[k]-vref[k-1])/ds = (scale_f[k-1])*accelr[k-1];
83  c16{k in 1..N}: (v[k]-v[k-1])/ds = (scale_f[k-1])*accel[k-1];
84
85  c60{k in 1..N}: (theta[k]-theta[k-1])/ds = (scale_f[k-1])*w[k-1];
86
87  c17{k in 0..N}:   v[k] <= 2.3;
88  c18{k in 0..N}:   v[k] >= 0.00001;
89  c21{k in 0..N}:   vref[k] <= 0.5;
90  c22{k in 0..N}:   vref[k] >= 0.00001;
91
92
93  # Camera based constraint definition
94  c1000{k in 0..N}: xcam[k] = xpos[k] + 0.30*sin(theta[k]);
95  c1001{k in 0..N}: zcam[k] = zpos[k] + 0.30*cos(theta[k]);
96
97  c1002{k in kset}: temp_dist_cam[k] =
98                      (x_sh[k] - xcam[k])*cos(theta_sh[k]) +
99                         (zcam[k] - z_sh[k])*sin(theta_sh[k]);
100 c1003{k in kset}: distance_cam[k] = temp_dist_cam[k]/
101                                     cos(theta_sh[k]-theta[k]);
102 c1004{k in kset}: distance_cam[k] >= - 0.10080;
103 c1005{k in kset}: distance_cam[k] <=   0.10080;
104 c1006{k in kset}: error_ang[k] = atan(distance_cam[k]/0.30);
105
106
107 # To avoid singularity
108 c1007{k in kset}: theta_sh[k]-theta[k]<=  1.5;
109 c1008{k in kset}: theta_sh[k]-theta[k]>= -1.5;
110
111 ## Path constraint
112
113 # To avoid singularity
114 c25{k in kset}: theta[k]-theta_t[k]<=1.5;
115 c26{k in kset}: theta[k]-theta_t[k]>=-1.5;
116
117 c90{k in kset}: distance[k] =(xpos[k] - x_t[k])*cos(theta_t[k]) -
118                                (zpos[k] - z_t[k])*sin(theta_t[k]);
119
120 #Initial conditions
121 c29: time[0]=0;
122 c30: xpos[0]=0;
123 c31: zpos[0]=0;
124 c32: theta[0]=0;
125 c34: x1[0]=0;
126 c35: x2[0]=0.0080;
127 c36: x3[0]=0;
128 c37: x4[0]=0;
129 c38: x5[0]=0;
130
131 # Max wheel speeds
132 c47{k in 1..N}: v[k]/r + L*(w[k])/(2*r) <= 46;
133 c48{k in 1..N}: v[k]/r + L*(w[k])/(2*r) >= 0;
134 c49{k in 1..N}: v[k]/r - L*(w[k])/(2*r) <= 46;
```

```
c50{k in 1..N}: v[k]/r - L*(w[k])/(2*r) >= 0;

#Accel
c51{k in 0..N}: accelr[k] <= 1;
c52{k in 0..N}: accelr[k] >= -1;
c53{k in 0..N}: accel[k] <= 1;
c54{k in 0..N}: accel[k] >= -1;

#Jerk
c55{k in 1..N}: (accelr[k]-accelr[k-1])/ds =
                                (scale_f[k-1])*jerkr[k-1];
c57{k in 1..N}: (accel[k]-accel[k-1])/ds =
                                (scale_f[k-1])*jerk[k-1];

c67{k in 0..N}: jerkr[k] <= 1;
c68{k in 0..N}: jerkr[k] >= -1;
c69{k in 0..N}: jerk[k] <= 1;
c70{k in 0..N}: jerk[k] >= -1;

##############################
data;

param L:=0.14;
param r:=0.05;

param N := 10282;

param:      x_t     z_t     theta_t   k_t   x_sh   z_sh   theta_sh:=
# Data from matlab file 'generate_racetrack_data.m'
# goes here

##############################
solve;
display vref,error_ang, v, theta, xpos, zpos,
          accelr, accel, jerk, jerkr, time,
           w, xcam, zcam;
```

```
1
2  # NON CAMERA BASED MINIMUM TIME
3  # AMPL code (simulation)
4
5  param N;  # number of integration steps
6  param ds:= 0.001;  # step size
7  set kset ordered:= 0..N; # defining a set from 0 to N
8
9  # declaration of racetrack parameters
10
11 param x_t{kset};
12 param z_t{kset};
13 param theta_t{kset};
14 param k_t{kset};
15
16 # declaration of robot parameters
17
18 param L;
19 param r;
20
21 # declaration of variables
22
23 var vref{kset};
24 var tref{kset};
25
26 var scale_f{kset};
27 var distance{kset};
28
29 var accelr{kset};
30 var accel{kset};
```

```
31 var w{kset};
32 var wref{kset};
33 var ang_accelr{kset};
34 var ang_accel{kset};
35 var jerkr{kset};
36 var jerk{kset};
37
38 var xpos{kset};
39 var zpos{kset};
40 var x1{kset};
41 var x2{kset};
42 var x3{kset};
43 var x4{kset};
44 var x5{kset};
45 var v{kset};
46 var theta{kset};
47 var time{kset};
48
49 minimize obj: time[N];
50
51 subject to
52 ## Cruise control state space with kinematics and scale factor
53 c1{k in 1..N}: (xpos[k] - xpos[k-1])/ds =
54                          scale_f[k-1]*(v[k-1]*sin(theta[k-1]));
55 c2{k in 1..N}: (zpos[k] - zpos[k-1])/ds =
56                          scale_f[k-1]*(v[k-1]*cos(theta[k-1]));
57 c3{k in 1..N}: (x1[k] - x1[k-1])/ds =
58         scale_f[k-1]*(-2.616*x1[k-1]-2.489*x2[k-1]+2*vref[k-1]);
59 c4{k in 1..N}: (x2[k] - x2[k-1])/ds = scale_f[k-1]*(2*x1[k-1]);
60 c5{k in 0..N}: v[k] = 1.244*x2[k];
61
62 c6{k in 1..N}: (x3[k] - x3[k-1])/ds = scale_f[k-1]*(-2.556*x3[k-1]-
63                              1.703*x4[k-1]-1.08*x5[k-1]+tref[k-1]);
64 c7{k in 1..N}: (x4[k] - x4[k-1])/ds = scale_f[k-1]*(4*x3[k-1]);
65 c8{k in 1..N}: (x5[k] - x5[k-1])/ds = scale_f[k-1]*(x4[k-1]);
66 c9{k in 0..N}: theta[k] = 0.4924*x4[k]+1.08*x5[k];
67
68 c14{k in 1..N}: (time[k] - time[k-1])/ds = scale_f[k-1];
69
70 c80{k in 0..N}: scale_f[k] = (1 - (distance[k]*k_t[k]) )/
71                                    (v[k]*cos(theta[k]-theta_t[k]));
72
73 # Acceleration definition
74 c15{k in 1..N}: (vref[k]-vref[k-1])/ds = (scale_f[k-1])*accelr[k-1];
75 c16{k in 1..N}: (v[k]-v[k-1])/ds = (scale_f[k-1])*accel[k-1];
76
77 c59{k in 1..N}: (tref[k]-tref[k-1])/ds = (scale_f[k-1])*wref[k-1];
78 c60{k in 1..N}: (theta[k]-theta[k-1])/ds = (scale_f[k-1])*w[k-1];
79 c61{k in 1..N}: (wref[k]-wref[k-1])/ds =
80                                      (scale_f[k-1])*ang_accelr[k-1];
81 c62{k in 1..N}: (w[k]-w[k-1])/ds = (scale_f[k-1])*ang_accel[k-1];
82
83 c17{k in 0..N}:   v[k] <= 2.3;
84 c18{k in 0..N}:   v[k] >= 0.00001;
85 c21{k in 0..N}:   vref[k] <= 0.5;
86 c22{k in 0..N}:   vref[k] >= 0.00001;
87 c23{k in 0..N}:   wref[k] <= 10;
88 c24{k in 0..N}:   wref[k] >=  -10;
89
90 ## Path constraint
91
92 # To avoid singularity
93 c25{k in kset}: theta[k]-theta_t[k]<=1.5;
94 c26{k in kset}: theta[k]-theta_t[k]>=-1.5;
95
96 # Distance to racetrack
```

```
c90{k in kset}: distance[k] =(xpos[k] - x_t[k])*cos(theta_t[k])-
                              (zpos[k] - z_t[k])*sin(theta_t[k]);
c27{k in kset}: distance[k] <= 0.01;
c28{k in kset}: distance[k] >= -0.01;

## Initial conditions
c29: time[0]=0;
c30: xpos[0]=0;
c31: zpos[0]=0;
c32: theta[0]=0;
c34: x1[0]=0;
c35: x2[0]=0.0080;
c36: x3[0]=0;
c37: x4[0]=0;
c38: x5[0]=0;

# Max wheel speeds
c47{k in 1..N}: v[k]/r + L*(w[k])/(2*r) <= 46;
c48{k in 1..N}: v[k]/r + L*(w[k])/(2*r) >= 0;
c49{k in 1..N}: v[k]/r - L*(w[k])/(2*r) <= 46;
c50{k in 1..N}: v[k]/r - L*(w[k])/(2*r) >= 0;

#Accel
c51{k in 0..N}: accelr[k] <= 1;
c52{k in 0..N}: accelr[k] >= -1;
c53{k in 0..N}: accel[k] <= 1;
c54{k in 0..N}: accel[k] >= -1;

c63{k in 0..N}: ang_accelr[k] <= 1;
c64{k in 0..N}: ang_accelr[k] >= -1;
c65{k in 0..N}: ang_accel[k] <= 1;
c66{k in 0..N}: ang_accel[k] >= -1;

#Jerk
c55{k in 1..N}: (accelr[k]-accelr[k-1])/ds =
                                  (scale_f[k-1])*jerkr[k-1];
c57{k in 1..N}: (accel[k]-accel[k-1])/ds =
                                   (scale_f[k-1])*jerk[k-1];

c67{k in 0..N}: jerkr[k] <= 1;
c68{k in 0..N}: jerkr[k] >= -1;
c69{k in 0..N}: jerk[k] <= 1;
c70{k in 0..N}: jerk[k] >= -1;

##############################
data;
param L:=0.14;
param r:=0.05;

param N := 10282;

param:      x_t     z_t     theta_t   k_t:=
# Data from matlab file 'generate_racetrack_data.m'
# goes here
##############################

solve;
display vref,tref, v, theta, xpos, zpos, accelr,
                           accel, jerk, jerkr, time, w;
```