©Copyright 2019
Anshul Sungra

Different control strategies for Mobile Robots

Anshul Sungra

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2019

Committee:

Brian C. Fabien, Chair

Joseph L. Garbini

Sawyer B. Fuller

Program Authorized to Offer Degree:
Mechanical Engineering

University of Washington

Abstract

Different control strategies for Mobile Robots

Anshul Sungra

Chair of the Supervisory Committee:
Professor Brian C. Fabien
Department of Mechanical Engineering

In this research, the following (ego) vehicle was maintaining a safe relative distance by varying the velocity through different controllers to catch up with the lead vehicle. Two major sensors were used, a rotating Laser Distance Sensor (LDS) and an RGB camera sensor. The camera sensor operated as a secondary system and was used to improve the detection probability of the lead vehicle. A sensor fusion algorithm was used for localizing the ego vehicle which includes a detection clustering algorithm and a Kalman filter to estimate the relative distance between the two vehicles. The sensors were calibrated for the test conditions to obtain detections with in feasible limits. Different controllers such as Proportional, Proportional-Integral, and Model Predictive Control were implemented and validated both in simulation and experiment on the turtlebot3-burger robot.

TABLE OF CONTENTS

Page

List of Figures . . . iii

List of Tables . . . v

Glossary . . . vi

Chapter 1: Introduction . . . 1

Chapter 2: Description of Platforms, Problem and Methodology . . . 5
- 2.1 Robot Operating System (ROS) . . . 5
- 2.2 MATLAB . . . 9
- 2.3 Aggregate Channel Feature (ACF) Detector . . . 12
- 2.4 Sensors . . . 13
- 2.5 Turtlebots . . . 19
- 2.6 Sensor Fusion . . . 20
- 2.7 Proportional Control . . . 26
- 2.8 Proportional-Integral Control . . . 27
- 2.9 Model Predictive Control . . . 29
- 2.10 Adaptive Cruise Control . . . 31

Chapter 3: Application . . . 33
- 3.1 Simulations . . . 33
- 3.2 Experimental Setup . . . 35

Chapter 4: Results and Discussion . . . 44
- 4.1 Sensor Calibration . . . 44
- 4.2 Simulation Results . . . 45
- 4.3 Experimental Results . . . 49

Chapter 5: Conclusion and Future work 56

Bibliography 58

Appendix A: Code, Simulation models and Project repositories 62

LIST OF FIGURES

Figure Number Page

1.1 Architecture Design for operating mobile robot to maintain safe distance . . 4

2.1 Robot Operating System (ROS) concept for communication between the nodes. 8
2.2 PC-Robot communication in ROS-MATLAB Interface 11
2.3 Communication in ROS-MATLAB Interface 11
2.4 LDS sensor visualization when it detects objects in the environment 15
2.5 Distribution of LDS measurement and span of detection 16
2.6 Relation between width of bounding box and relative distance 18
2.7 Clustering of LDS detection points for sensor fusion 21
2.8 Representation of controller in the system. 27
2.9 Representation of Model Predictive Control. 29
2.10 Maintain safe Relative Distance between the robots 30

3.1 System Architecture design in Simulation. 34
3.2 LDS estimated relative distance with actual distance 38
3.3 Vision estimated relative distance with actual distance 39
3.4 Actual Track 41
3.5 Track setup 42

4.1 Comparison between LDS and camera relative distance. 44
4.2 Proportional Controller to maintain safe distance when lead vehicle has constant velocity. 46
4.3 Proportional Controller to maintain safe distance when lead vehicle has ramp velocity. 46
4.4 Proportional-Integral Controller to maintain safe distance when lead vehicle has constant velocity. 47
4.5 Proportional-Integral Controller to maintain safe distance when lead vehicle has ramp velocity. 48

4.6 Model Predictive Controller to maintain safe distance when lead vehicle has constant velocity. 49
4.7 Model Predictive Controller to maintain safe distance when lead vehicle has ramp velocity. 50
4.8 Experimental implementation Proportional Controller to maintain safe distance when lead vehicle has constant velocity 50
4.9 Experimental implementation Proportional Controller to maintain safe distance when lead vehicle has step velocity 51
4.10 Experimental implementation Proportional-Integral Controller to maintain safe distance when lead vehicle has constant velocity 52
4.11 Experimental implementation Proportional-Integral Controller to maintain safe distance when lead vehicle has ramp velocity 53
4.12 Experimental implementation Proportional-Integral Controller to maintain safe distance when lead vehicle has constant velocity through camera sensor only 53
4.13 Experimental implementation Model Predictive Controller to maintain safe distance when lead vehicle has constant velocity 54
4.14 Experimental implementation Model Predictive Controller to maintain safe distance when lead vehicle has step velocity 55

A.1 Simulink Process for Proportional Control 85
A.2 Simulink Process for Proportional-Integral Control 85
A.3 Simulink Process for Model Predictive Control 86

LIST OF TABLES

Table Number — Page

2.1 Distance Performance Specification for LDS 14
2.2 Hardware Specifications for Raspberry Pi Camera V2 18
2.3 Hardware Specifications for Turtlebots 20
2.4 Effects of increasing the parameters independently 28

GLOSSARY

ACC: Adaptive Cruise Control which normally found in most of the modern cars to maintain the set velocity by following the lead vehicle.

ACF DETECTOR: a detection algorithm that detects the objects in the images through Aggregated Channel Feature.

CAV: Connected Automated Vehicles, where the vehicles are connected and sharing information with each other through a different medium.

CLUSTERING: a group of points with similar properties on the specific space or domain.

KALMAN FILTER: a filter which widely used to smooth out the noise in the system and estimate the parameters.

LIDAR: is a surveying method that gets the relative distance by illuminating the laser and measuring the reflected photon through the sensor.

MATLAB: a platform for computation, analyzing and many other features which widely used in science and engineering applications.

MPC: called Model Predictive Control where the next control action is predicted from the previous step by getting the optimal solution.

OPENCR: an open-source embedded board which normally used for robotics applications.

RADAR: an object detection system that uses radio waves to determine the velocity, distance, the orientation of the detected object.

RASPBERRY PI: a small computational board used in many different engineering applications.

ROS: Robot Operating System is a platform usually used in robots for the development of algorithms and hardware-software interface.

SIMULINK: is a platform to create and analyze the system models for their behavior.

TURTLEBOT: is a small mobile robot that is handy to develop and test different algorithms.

V2V: Vehicle to Vehicle Communication where the vehicles communicate various information through the wireless system in the environment.

ACKNOWLEDGMENTS

Since joining UW EcoCAR team, I have learnt and applied various algorithms related to Advanced Driving System. This lab. also provides all sorts of resources to learn and help an individual to develop personally and professionally. There were many failures along the way while working on this project however, all those problems were resolved with perseverance to achieve the goal of this project. There are so many people in the team helped me to accomplish this research project.

I want to thank my thesis committee members. Dr. Fuller was with this project since the beginning of this research. Also, thanks to Dr. Garbini to accept my request during the last quarter of the presentation. I also want to thank Dr. Ashis Banerjee who was guiding me about the robotics project and the future research in this field. He was the one who suggested some of the different approaches for solving the problem. Moreover, he was also providing his suggestions about the scope of work in this domain. I also want to thank Aman Kalia to help me in the experimental setup and reviewing the control concepts which I was applying to the system.

I want to thank my faculty research advisor, Dr. Brian Fabien to guide me in this project. Even during his busy schedule, he was eagerly helping me to solve the problem. Whenever I was stuck to formulate the problem, he was there to review and check the results of the solutions. Due to his advice, this project has resulted in high-quality outcomes.

Finally, I want to thank my parents who have encouraged and supported me during my life to achieve my goals.

DEDICATION

to my mother, Shaila and father, Virendra

Chapter 1
INTRODUCTION

This project describes one of the strategies that can be used to make self-driving vehicles. Most of the self-driving technology focuses on self-localization and navigation. However, it will be beneficial to driver safety, fuel economy, and localization of the vehicle if vehicles can adapt to the needs of the traffic in the environment. For example, vehicles maintaining a safe distance between them. Most of the autonomous vehicles are equipped with different sensors such as Inertial Measurement Unit (IMU), Global Positioning System (GPS), Cameras, Light Detection and Ranging (LIDAR) and Radio Detection and Ranging (RADAR). The adaptions of the vehicles in the environment is an Adaptive Cruise Control. Therefore, the implementation of this system in the real vehicles provides a large extent of possibilities to improve the localization system and also passenger safety. Some companies are focusing on this technology to implement in the vehicles more efficiently. So, this research project has emerged to apply and check the feasibility of the ACC system in the real world for the EcoCAR team.

There are many other technologies available, however, the ACC algorithms should be more efficient so that it can decide any control action on time. This project used small mobile robots i.e., turtlebot3-burger to implement and test some distance maintenance algorithms. By testing and validating control strategies on mobile robots, we will have the confidence to implement in the actual cars. Moreover, its future application is vast including, save driving time and fuel consumption. These are the main factors that might create an efficient transportation system and have optimal energy consumption.

Wang et. al., [36] provided an overview of the future development of the Cooperative

Adaptive Cruise Control (CACC), where vehicles communicate with each other. This method also enhances the efficiency of the vehicle and their driving conditions. This paper mentioned that the Connected and Automated Vehicles (CAV) system can achieve high passenger safety, mobility and the sustainable energy consumption of fuel in transportation. CACC technology is widely used in platooning where many vehicles come together and follow each vehicle in front, thus forming a vehicle train. Moreover, this paper also stated different control methodologies and their pros and cons during implementation. This paper also discussed the reliable system architecture in a realistic environment and the cost of implementing CACC on a large scale.

Mehra et. al., [17] implemented Adaptive Cruise Control on the scaled model car with a model predictive control strategy. They demonstrated this methodology by going beyond Proportional-Integral-Derivative (PID) controllers. They used safety constraints through Control Barrier Functions (CBF) and achieved control objectives through Control Lyapunov Functions (CLF). They also used the QP based controller for implementation of Model Predictive Control (MPC). Moreover, this paper validated the experimental results with the simulation. In this paper, online optimization of the control actions was achieved to satisfy the constraints and other parameters. This controller also handled the multiple objective functions in one way.

Taku Takahama and Daisuke Akasaka [18] implemented the Model Predictive Control with a low computational cost for ACC during the traffic jams. This paper emphasizes that the vehicle characteristics changes with the driving conditions. The algorithm also ensured a high response and less discomfort during the traffic jams. They experimentally verified this algorithm with various traffic situations. Moreover, the operating regions were also shown in the phase plane. They also implemented a smooth braking system regarding passenger safety and comfort.

Rawlings [20] had described the use of Model Predictive Control in dynamic systems. Moreover, he provided the framework to address critical issues and its analysis. In this book, also discussed some trade-offs while using MPC and also mentioned the application

of MPC in nonlinear systems. This article also talked about the feasibility of the MPC. Furthermore, the tutorial also discussed the robustness of the algorithms.

In the master thesis by Peng Xu who designed the learning-based Adaptive Cruise Lane Control (ACLC) system. This ACLC was tested in different simulation environments. He used a deep reinforcement learning algorithm to train the system to keep the safe distance with the same velocity. Moreover, he used the Robot Operating System (ROS) and Gazebo platform for simulation and tested different scenarios such as multiple lanes, crowded environments, etc. This thesis had a deep Q learning model by the open AI gym which used for learning and training. Also, he used the CAN simulator kit which provided open-source software for the Robotic Operating System (ROS) and Gazebo. Some of the path planning algorithms were implemented to keep the car aligned while running the lane.

Shouyang Wei et. al. [37] design and experiment with the Cooperative Adaptive Cruise Control (CACC) based on Supervised Reinforcement Learning (SRL). They used the SRL framework in longitudinal vehicle dynamics control. They trained a neural network by real driving data to achieve human-like CACC with the actor-critic reinforcement learning approach. This approach used real driving and gain scheduler to update the network. The SRL control policies are then compared with the simulation and experimental results. This process validated the adaptability of the system which was closely performing like human driving behavior.

As the researchers are trying different techniques to solve the problem of learning and adapting the system to take control actions, this project also provides the essence of solving the problem with simple solutions. This work is different from Mehra et. al., [17] because it was using different sensors and hardware to test. Moreover, the experimental setup is different in this project when compared to their work. They had implemented the control algorithm on board instead of computation on master PC (Personal Computer). Figure 1.1 shows a brief architecture of the robot system to have an understanding of the process. This project used small robots i.e., turtle bots to maintain the safe distance between the lead and ego robots. Moreover, the ego robot which was following the lead vehicle focused on

longitudinal autonomy. The two most important sensors were a rotating laser sensor and a camera sensor to estimate the relative distance between the robots. After estimating the distance, sensor fusion was carried out by combining detection from laser and camera sensors. This sensor fusion algorithm also clusters the detection of the Laser sensor point cloud in real-time. Kalman filter was used to eliminate noise and estimate the velocity from distance measurement. This velocity was used in the application of different control algorithms such as Proportional, Proportional-Integral and Model Predictive Control.

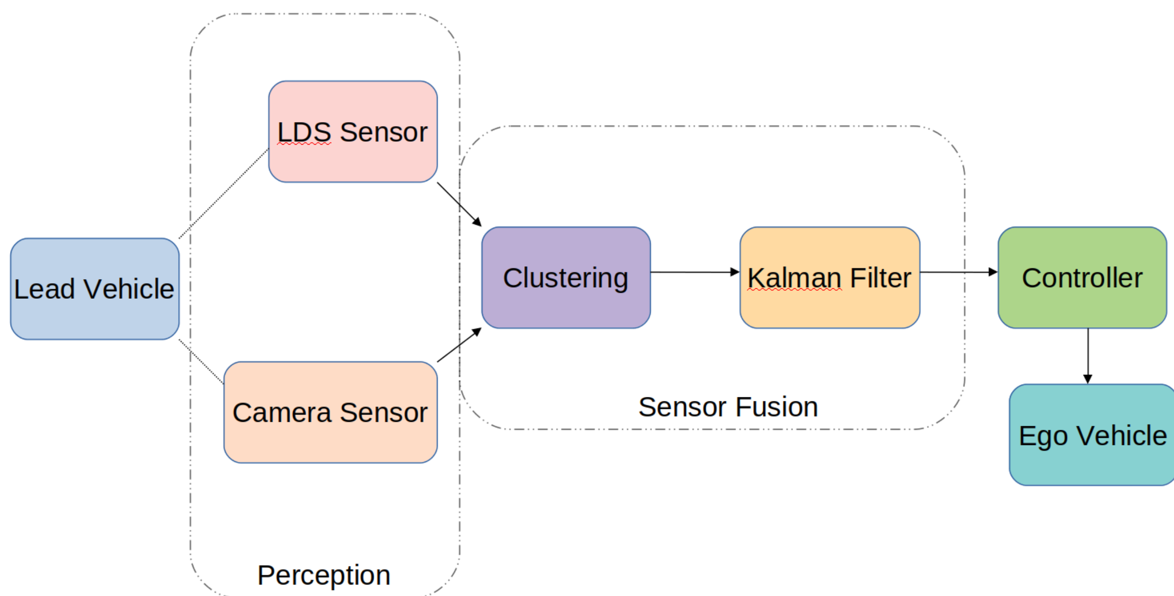

Figure 1.1: Architecture Design for operating mobile robot to maintain safe distance

All this development was done on the ROS-MATLAB platform. Moreover, for experimental setup, a physical track was built to make the robots run on the longitudinal way instead of deviating to curved routes which will deteriorate the detection of sensors mounted on ego vehicle. All these implementations were also done in the simulations in Simulink platform to check the validity of the experiments. Both simulation and experimental results were compared, analyzed and validated.

Chapter 2

DESCRIPTION OF PLATFORMS, PROBLEM AND METHODOLOGY

2.1 Robot Operating System (ROS)

Robot Operating System (ROS) is an open-source, meta operating system for robots. It provides services such as hardware abstraction, device control, implementation of functionalities, message-passing between the processes, and package management. Moreover, it provides libraries for building, writing, running and obtaining code across multiple computers. There are similar kinds of robot frameworks such as Microsoft Robotics Studio, MOOS, CARMEN, etc [32].

ROS supports the reuse of code in robotics research and development. It is a distributed framework that enables executables to be designed for processes. These processes can be easily shared and distributed by grouping them as packages and stacks. ROS also provides contributions and distribution of code repositories. This design makes independent decision making about development and implementation at the community level. In the end, this distributed design can be brought together with ROS infrastructure tools.

Right now ROS only runs on Unix based platforms. The most preferable operating system is Ubuntu and also Mac OS X systems can be used. Other operating systems can be used but those are experimental projects and not fully explored for ROS development. The core ROS system releases its ROS distribution along with useful tools and libraries. More information about its distribution is mentioned here [32]. These distributions are a different version of packages. Moreover, these distributions provide developers a stable code-based environment to release their packages. For this project, ROS Kinetic Kame is used which was released on May 23, 2016, where this distribution is compatible with Ubuntu 16.04 LTS

version. ROS is an open-source platform, where many robotics researchers and developers contribute towards its libraries and packages for others to use and build for their projects. This makes a very active ROS community in the world.

2.1.1 Robot Operating System (ROS) concept

ROS runtime graph is a network of processes that communicate with each other through ROS communication infrastructure. It implements synchronous RPC-style communication over services, streaming of data over topics and storage of data on Parameter Server. For explaining in detail, ROS has three levels of concepts: the Filesystem level, the Computation Graph level and the Community Level.

ROS Filesystem Level

The Filesystem covers different levels of files in a package on the disk such as:

- **Packages**: Packages are the main organized file system in ROS which contains nodes/run-time processes, ROS dependent libraries, and other files that are organized together. Packages are build and released in ROS to execute the different processes in ROS.

- **Metapackages**: These are represented as a group of other packages. They are most commonly used as a compatible place holder for converted rosbuild stacks.

- **Package Manifests** : This manifest file is in the format of .xml. This file contains meta-data of the package, its name, version, descriptions, license information, dependencies and other meta information like exported packages. It serves as a template for a real dissertation.

- **Repositories** : It is a collection of packages that share the same version control system. Packages that are at the same version can be release together using a catkin release. These repositories can be converted to rosbuild stacks.

- **Message types** : This file contains the type of messages stored in the file with defined data structures for messages sent in the ROS. This file in the format of $package_name/msg/message.msg$.

- **Service type** : This file contains the definitions for request and response data structures for services in ROS. This file is in the form *package/srv/service.srv*.

ROS Computation Graph Level

The Computation Graph is peer to peer network which connect to different processes and they communicate with each other. The basic concepts are Nodes, Master, Parameter Server, messages, services, topics, bags. All provide data to graphs while communicating with each other. Figure 2.1 shows a general overview of the ROS concept.

- **Nodes** : Nodes are the different types of processes computationally perform to control the robots to perform the task. These nodes might be the one which takes the information of laser scan, other nodes control the actuator of the robot and another node might do localization and path planning processes. ROS node is written with the help of ROS libraries such as roscpp, rospy, etc.

- **Master** : The ROS Master looks up the computational graph and registration. Without a master, the nodes may not find each other and communicate with each other through messages and services.

- **Parameter Server** : The Parameter server is part of Master, store data in a central location.

- **Messages** : Nodes communicate with each other bypassing the messages. A message contains the data structures of different data types such as int32, float32, boolean, etc. This much similar to the C structs which are stored in the form of a file.

- **Topics** : Topics are used to transport messages through publish/subscribe semantics. The topic is used to identify the name of the message. A node subscribed to an appropriate topic to do the process and publish the messages to the topic to perform the task. There might be multiple subscribers and publishers to different topics in a node.

- **Services** : Sometimes there is difficulty in transporting the request/reply interaction in the distributed systems. Requests and rely are done by services, which is defined by the pair of message structures. A node offers service under the name and a client uses the service by sending a request message and waiting for a reply.

- **Bags** : Bags are the format of saving the ROS message data. It is an important mechanism to record the data of the sensors and can be used for developing and testing the algorithms.

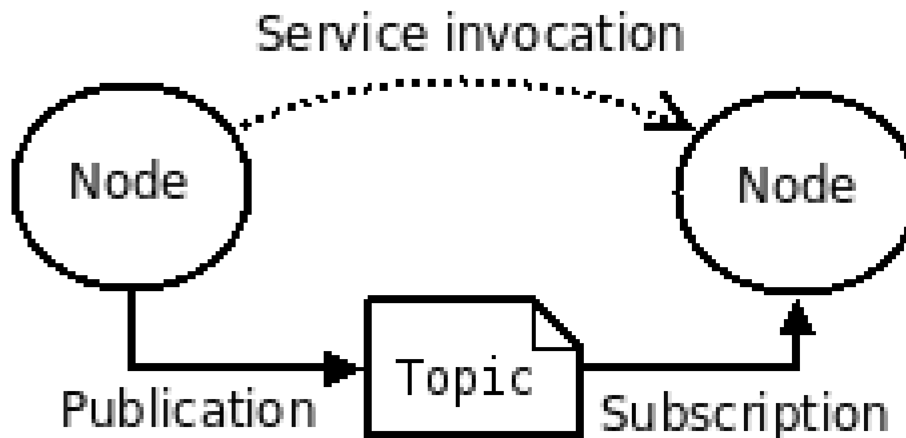

Figure 2.1: Robot Operating System (ROS) concept for communication between the nodes [33].

ROS Community Level

ROS community level concepts enable the ROS community to contribute and exchange software and knowledge. These resources include:

- **Distributions** : ROS distributions are collections of versioned stacks. It makes easier to install software which is consistent with the set of software.

- **Repositories** : ROS has a network of code repositories where different institutions can develop and release their robot software components.

- **ROS Wiki** : ROS community has the main forum for documentation of information about ROS. Anyone can contribute to that documentation content and have fully explained tutorials. For more information about ROS, a more detailed version is available here[33].

2.2 MATLAB

MATLAB (Matrix Laboratory) is a numerical computing environment and computer software language developed by MathWorks. MathWorks is also providing Simulink software which supports data and simulation. Both of the software provides different tools and a range of products to solve the problem in engineering to science. Many products have the support and full documentation of functions, solvers, and toolboxes which is helpful to develop an algorithm in a small amount of time. MATLAB was created in the 1970s by Cleve Moler, who was a chairman of the Computer Science department at the University of Mexico. It was a free tool for academics. He founded Mathworks along with Steve and Little Bangert who rewrote the MATLAB in C programming language. The two lead products of Mathworks are MATLAB, which provides the environment for programmers to analyze data, develop algorithms. Moreover, Simulink is a graphical and simulation environment which mostly used to develop the dynamical models of the systems.

MATLAB allows matrix manipulation, plotting functions, and data, implementation of algorithms, creation of user interfaces and have good interfaces between `C, C++, Python, Java, C#, Fortran`. MATLAB users come from various backgrounds of engineering, science, and economics. MATLAB was first adopted by researchers in control engineering. Soon after spread into other domains. It is mostly used in teaching linear algebra, numerical analysis and also in image processing. MATLAB supports the necessary data structures and algorithms such as functions, structures, classes, and Object-Oriented Programming. Moreover, it has good graphics and a graphical user interface to understand and analyze the scientific data and do research. MATLAB interaction between C and other languages with some executables and call the functions written in C or Fortran.

MATLAB-ROS interface

MATLAB provides the Robotics System Toolbox which allows ROS functionality. This toolbox communicates with the ROS network, access to the sensor data for visualization to develop robotics applications and simulators. Moreover, it also enables physical robot simulators such as Gazebo. As information about the ROS is already covered in section 2.1, most of the functionality for algorithm development can be done in the MATLAB. It initializes the ROS network. There are numerous commands in MATLAB to enable the function of ROS one such is `rosinit`, which creates a ROS master in MATLAB by starting the global node that is connected to the master. In Figure 2.2, shows the connection between different computers with the robots in the ROS network. Computers connect with the IP address of the ROS master which is also called the global node in the MATLAB. After specifying the IP address of the ROS nodes, the robot and computer will use this address to send and receive data to the global nodes in the MATLAB.

Therefore, after connecting to the ROS network, then we can see and exchange data in the system and also send signals to the robot. These types of network connections can be verified through different commands in the MATLAB. These are similar to the ROS such as publisher: for publishing messages of the rostopic. Moreover, the system can subscribe to

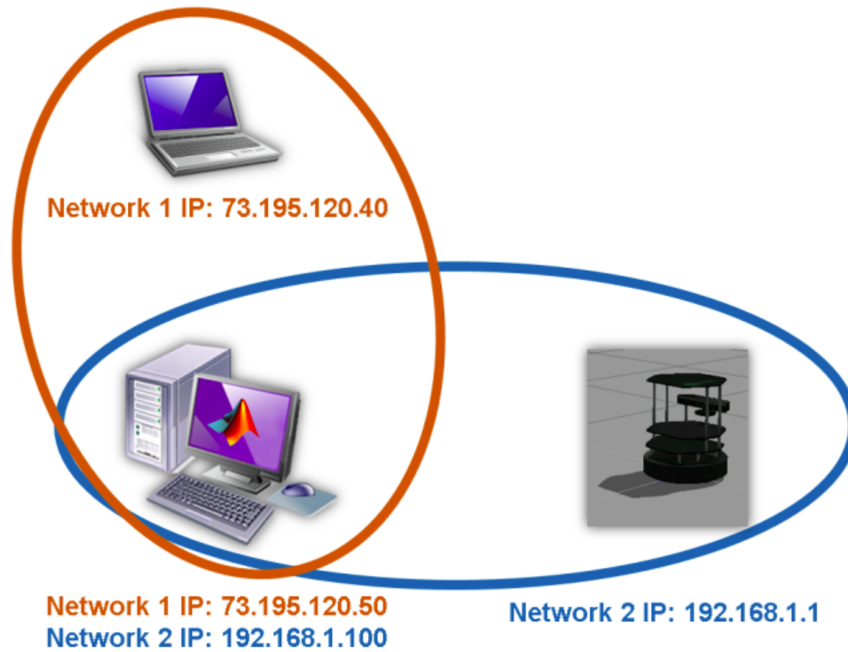

Figure 2.2: ROS-MATLAB concept for communication between the PC [11].

the different rostopics of the robot and see different sensor data and messages. This infers that these messages and multiple nodes are communicating the different systems efficiently.

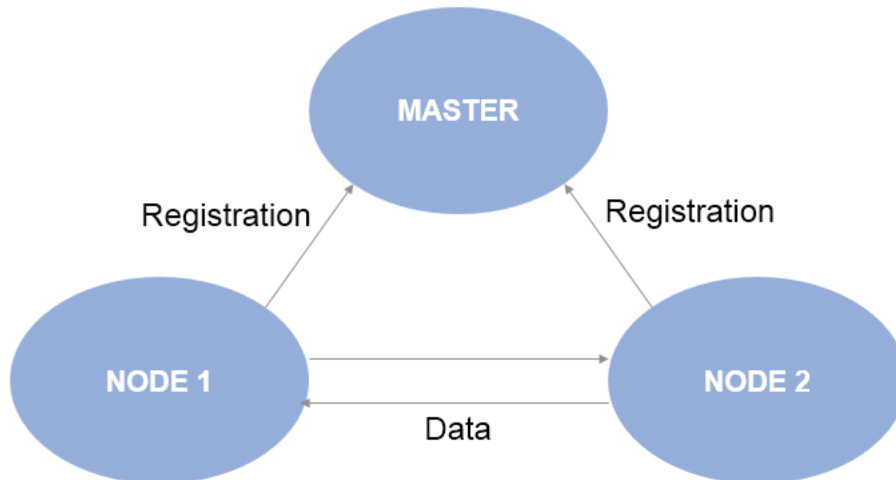

Figure 2.3: ROS-MATLAB concept for communication between Master and Nodes [11].

In Figure 2.3, shows the ROS network with one ROS master with two different nodes that are registered with the ROS master. Each node communicates to the ROS master by finding the advertised address of the other node in the ROS network. After recognizing the address of each node, then they exchange data without a ROS master. More information about this platform can be found here [11][12].

2.3 Aggregate Channel Feature (ACF) Detector

This detector is an evolution of the Viola & Jones detector with a decrease in false-positive rates. It is a fast and effective sliding window detector developed by Piotr Dollr et. al. and tested on different pedestrian datasets. The main reason to use this detector other than different deep learning detector algorithm because it is computationally inexpensive. Moreover, this algorithm is accurate. In this detector, features from the images are sampled and create pyramids without sacrificing the performance. Moreover, this algorithm has a wide range of applications for detecting different objects. More detailed information about this detector can be found in this paper [3].

To train this detector algorithm a dataset was created. This dataset contains images of turtlebots in the environment which was also called the positive image for training and classification from the environment features. Moreover, negative sample images were the environment or surrounding images without the turtlebot. Therefore, there was a total of around 2000 images where 1000 were positive images and others are negative ones. This dataset then labeled with turtlebot, to identify it in the image to every 1000 images before training. This will provide the information to the detector that what object in the image it is detecting. This labeling was done with the help of the image labeler app [13] available in the MATLAB. This application is suitable to label multiple images in one instant by loading images in one go. The definition for the region of interest (ROI) was defined manually for each image in the dataset to get labels accurately and good input for training. This labeled data then exported to the MAT file and then used for training.

As mentioned above about ACFdetector, which was more convenient in this type of

application was used to train with this dataset. For training this data pre-built ACF detector model was used, This model and function can be found in the MATLAB [16]. This detector model returned a trained aggregate channel features (ACF) object detector. This function collects the positive instances of the data and automatically collects the negative part in the training. After training, the detector was then tested on different sample images to see the performance of detection. There are many different types of training models available in the MATLAB which can be used according to the application of the project. The detect function returns the location of turtlebot in the image by providing a set of bounding boxes. This also provides the confidence score of the detection in the image. This bounding box information then later used to estimate the relative distance between the lead and ego turtlebots from the camera point of view.

2.4 Sensors

There are mainly three sensors used for this project i.e., 360° Laser Distance Sensor LDS-01, Inertial Measurement Unit (IMU) and, Raspberry Pi Camera Module v2. These sensors were sufficient for the application to apply the control algorithm. These sensors made the system more robust and make control algorithms more useful. These sensors are explained in more detail:

2.4.1 360° Laser Distance Sensor LDS-01

This Laser sensor is capable of sensing objects around 360 degrees of the robot. It mainly uses for SLAM(Simultaneous Localization and Mapping) and Navigation. This sensor is also supported by the ROS platform which is very important in developing algorithms for the project. This sensor was connected to the USB to PC and had firmware which was downloaded and installed on OpenCR in Arduino IDE. Moreover, individual drivers are also provided by the manufacturer to use in different applications. For turtlebots, the main use of this sensor was to get the distance measurement of the object in the environment.

LDS perform well to detect the objects in the environment. Moreover, it also has some

Table 2.1: Distance Performance Specification for LDS

Item	**Specifications**
Distance Range	120 mm – 3500 mm
Distance Accuracy (120 mm – 499 mm)	±15 mm
Distance Accuracy (500 mm – 3500 mm)	±5.0%
Distance Precision (120 mm – 499 mm)	±10 mm
Distance Precision (500 mm – 3500 mm)	±3.5 %
Scan Rate	300 ± 10 rpm
Angular Range	360°
Angular Resolution	1°

noise in the system which has to eliminate. For more detail about the specifications of the LDS sensor can be found in the document [23] [6]. Some of the Measurement specifications for this sensor are mentioned in Table 2.1

This sensor provides the point clouds for visualization whenever it detects any objects in the surroundings. These point clouds can also give knowledge about the size of the object. Moreover, from laser detection, the type of object can also be known because there are different reflectivity of the laser photons to different materials. Some materials absorb photons and some reflect it in high volume. We can see this visualization in figure 2.4. So, these detections can be downsampled to the requirements such as there is no need for object detection in the left and right side of the turtlebots because there this project was focused on the longitudinal autonomy of the robot instead of lateral one. Therefore, the span of the detection was downsampled from 360 points to 60 to 30 according to the need of the application for the system. For this project, the span was only 30° front of the robot and the rest of the detections were ignored. This also saved some computational time involved while interpreting the data for the sensors and use those data for further process. Moreover,

these points then used in the clustering algorithm to get the distance of the object more accurate and precise. These points will also help in developing the sensor fusion for the system. Moreover, the sensor system was calibrated while using its detection data.

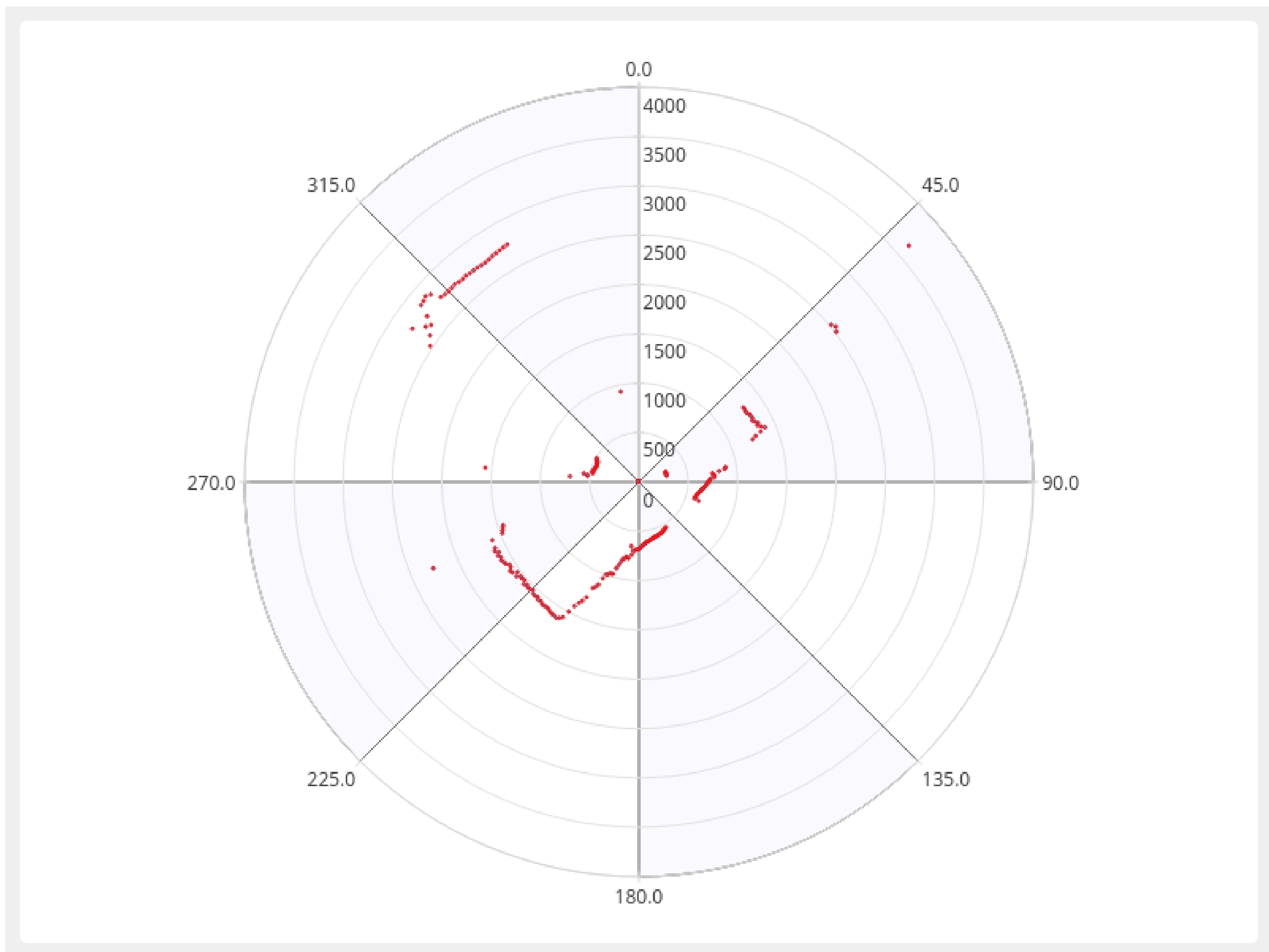

Figure 2.4: 360° Laser Distance Sensor visualization [23].

After downsampling the points of the sensor detection, these points are then mapped into the planar coordinate system where x coordinate was the front direction of the robot and y coordinate as the left side of the robot. Let us consider R_i as a relative distance detected by these points. Moreover, in this case $346 \leq i \leq 360$ and $1 \leq i \leq 15$ where $R_{360} = R_0$. Therefore, we can have a simple relation to converting into a planar coordinate.

$$x = R_i \cos \theta_i \tag{2.1}$$

$$y = R_i \sin \theta_i \tag{2.2}$$

where, $346^\circ \leq \theta_i \leq 15^\circ$. Then, these coordinates are transferred to $m \times 2$ matrix size where $m = 30$ represents the number of detection and 2 columns represent the coordinates x and y. In Figure 2.5, the distribution of angle is shown and the span of the LDS detection was down-sampled to do further computation.

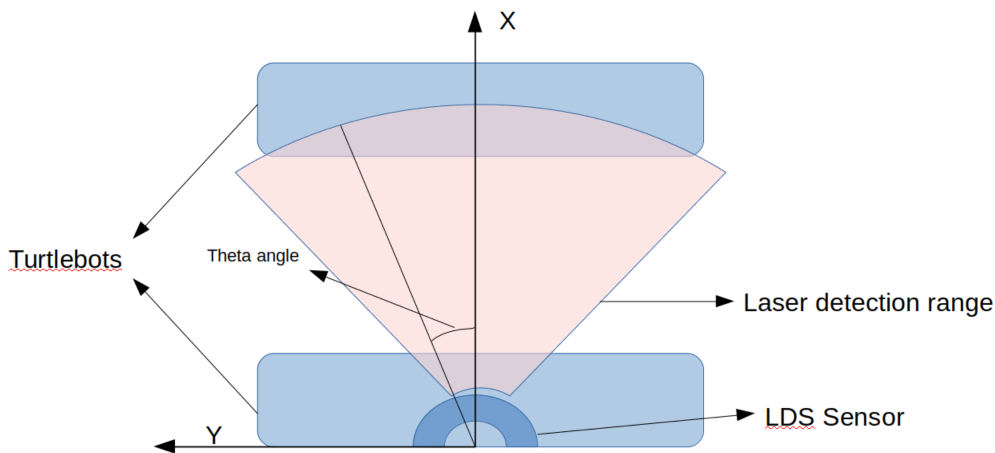

Figure 2.5: Distribution of LDS measurement and span of detection

2.4.2 Inertial Measurement Unit (IMU)

The Inertial Measurement Unit (IMU) was installed on the OpenCR1.0 embedded board which was developed for ROS embedded system to provide complete open-source hardware and software. This board is compatible with Arduino IDE for firmware development. For more information, the manual for this board is given here [24]. This board has a Gyroscope, Accelerometer, and Magnetometer to measure the orientation, angular velocity, and acceleration of the turtlebot. This is useful to develop the ROS messages from these sensors. The

IMU data published in the ROS topics subscribe to it for use. Moreover, turtlebot estimates the movement in the environment through the odometry model [19]. This model was already programmed in the turtlebot to get the information about how much robot moves i.e distance, velocity, and acceleration in the coordinate system. This information helped to get the velocity of the robot. Therefore, to use in the algorithms this ROS topic was subscribed to get the movement information about the robot.

2.4.3 Raspberry Pi Camera Module V2

The Raspberry Pi Camera was a good sensor according to the requirements of the project. Moreover, the Raspberry Pi board compatible with it and supports many features such as time-lapse, slow-motion, etc. There are libraries available with camera modules to create effects. The installation of the camera was smooth and its interface was working fine without any hurdle. Some calibration files need to be set up for the ROS environment to transmit messages to master Personal Computer (PC). Already existed calibration files were used for this camera. The ROS software packages libraries are already done by ROBOTIS and Ubiquity Robotics [27][35]. Some of the specifications of Raspberry camera module V2 are mentioned in Table 2.2.

After installing and checking the video transmission from robot to master PC, algorithm development was started. As mentioned that the ACF detection algorithm was used to detect the lead robot, which gave the bounding boxes for turtlebot detection. The width of the bounding boxes was used to generate a relation in relative distance estimation between the robots. An experimental data was generated with different pixel images to check the behavior of the relative distance with the number of pixels width covered by the bounding box. Therefore, the relationship between the relative distance and width of the bounding box is shown in Figure 2.6. This was curve fitted with the experimental data. In Figure 2.6 The experimental data were taken by placing the lead robot at a fixed relative distance about 10cm intervals till 2.5m. At each interval of 10cm, the pixel width of the bounding box was recorded. This gave an approximate relation between width and relative distance.

Table 2.2: Hardware Specifications for Raspberry Pi Camera V2

Items	**Specifications**
Sensor	3280 × 2464 pixels
Pixel Size	1.12 μm × 1.12 μm
S/N ratio	36 dB
Fixed Focus	1 m to infinity
Focal Length	3.04 mm
Horizontal Field of View	62.2°
Horizontal Field of View	48.8°
Focal Ratio (F-stop)	2.0

The same procedure was used in all different pixel size images. From the Figure, we can see that the relationship is following the power equation close to the hyperbolic curve in different image sizes.

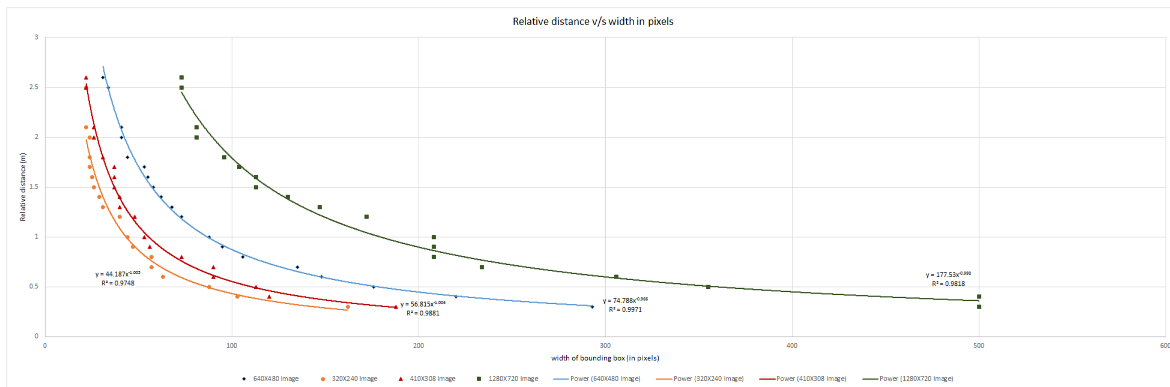

Figure 2.6: Relation between width of bounding box and relative distance

For the experimental use, 410 × 308 pixel image was applied because of the size of the bounding box was not varying frequently while taking the readings. Therefore, the relation

is:

$$\text{Relative distance} = 56.815 \times \text{width(in pixels)}^{-1.006} \tag{2.3}$$

with residual error for curve fitting was $R^2 = 0.9881$. After deducing the x coordinates, these data were transformed into y coordinates. So dimension of the bounding box in the image was $[x_{rimage}, y_{rimage}, \text{width}, \text{height}]$ in pixels. Therefore, the midpoint of the bounding box in x coordinate in the image plane was calculated by:

$$x_{midimg} = x_{rimage} + \text{width}/2 \tag{2.4}$$

The image length is divided into half, where the left part considered as the negative and right part as a positive plane for world coordinate which was seen in the image. Therefore, x coordinate of midpoint from the center of the image was given by:

$$x_{frommid} = x_{midimg} - 205 \tag{2.5}$$

where 205 was derived from half of the image length which was a total 410 pixel. After getting the value from the center, this pixel coordinate was then transformed into a world coordinate by multiplying with the ratio between world x coordinate and pixel length of the image. Here, in this situation x_{world} = Relative distance.

$$y_{world} = \frac{x_{frommid}}{410} \times x_{world} \tag{2.6}$$

2.5 Turtlebots

Turtlebots are a good platform to test the algorithms before putting to the vehicle and other main hardware. Many already built ROS packages are available easily for visualization in the simulation environment. All customized hardware was available for it and therefore, the primary focus was software development. Many universities collaborated to research these vehicles to test their algorithms. Furthermore, robotics communities for these robots are actively contributing to this platform which convinced to buy them. There are some specifications for these robots mentioned in Table 2.3. For more information about the

Table 2.3: Hardware Specifications for Turtlebots

Items	**Specifications**
Maximum translational velocity	0.22 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)
Size (L x W x H)	138 mm $\times$ 178 mm $\times$ 192 mm
Threshold of climbing	10 mm or lower
Expected operating time	2 hr 30 min
SBC (Single Board Computers)	Raspberry Pi 3 Model B and B+
Actuator	Dynamixel XL430-W250

turtlebot burger can be found in this link [30]. Most of the software collaborations are open source and readily available for the current research community to work on it. Many other computers and sensors are compatible with these robots and platform. Some of them are Nvidia Jetson TX2, intel depth cameras, Intel Joule 570x, etc.

As turtlebot had two wheels and ball caster for its mobility, therefore it follows the differential drive. So, while working on lateral autonomy this differential drive has to be taken into account. But this project was only longitudinal autonomy, therefore there was no main effect due to the differential drive. This robot used a kinematic model of the system and motor dynamics are already modeled inside the system. Therefore, velocity commands were sent for its localization.

2.6 Sensor Fusion

There were multiple sensors used for controlling the mobile robots, therefore data from these sensors were fused. The coordinates which were estimated from different sensors were added to the data structure. These different sensor data in the form of x and y coordinates were added as individual objects. Moreover, the type of detections such as turtlebots, pedestrian,

traffic light, etc, has to be assigned with their class index for classification. As in this project, only one object was detected i.e., turtlebot, therefore the class index was assigned as number 1. Then these data structure was used for clustering algorithm to get the precise distance measurement and then eliminate noise with velocity estimation through Kalman filter.

2.6.1 Clustering Algorithm

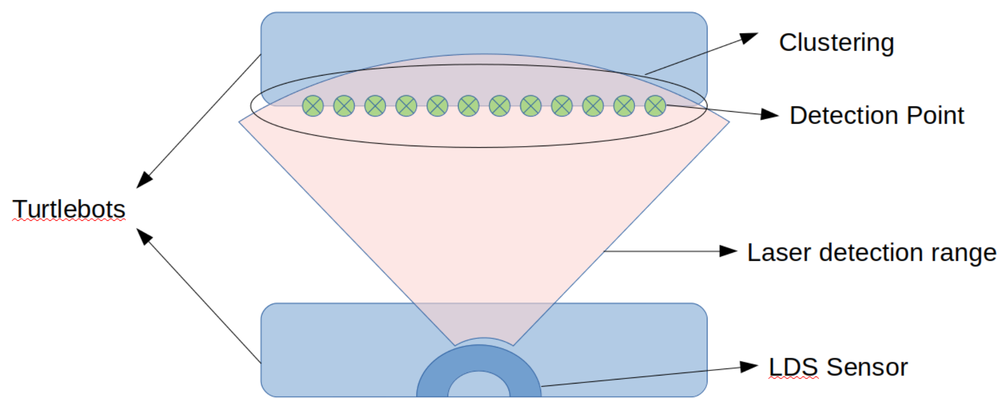

Figure 2.7: Clustering of LDS detection points for sensor fusion

In Figure 2.7 the LDS sensor provided different detection points for each degree angle, therefore for one object, there were multiple points of detection. So to get one detection point from multiple points, these points were clustered together and compare with the turtlebot length. So, this one detection point was achieved by transferring all measurement data which was in x and y coordinates into an array and calculated the euclidean distance between each nearby point of detection. This distance was then saved into a matrix that had dimensions equal to the number of points for detection. This means that if there were m points of detection then the matrix dimensions were $m \times m$. Then the distance measured stored in this matrix was compared with the turtlebot length. If distance was less than turtlebot length then that measurement information was retained to take the mean of the points of detection.

After completing through all points of measurements, this algorithm output the one single average value of x and y coordinates of lead turtlebot. Then this single LDS information was taken as mean(average) with the camera measurements. This process ensured that there will be more precision for objects detected at a particular coordinate. The pseudo-code for the clustering algorithm is mentioned here Algorithm 1. For more information about the sensor fusion can be found here [9].

```
Result: x and y coordinates of centroid of clusters
N = number of points for detection;
i = 1;
j = i + 1;
Check = 1 : N;
detectionClusters = N cells;
while i ≤ N do
    while j ≤ N do
        if see nearby detection for same sensors then
            calculate the euclidean distance of nearby detection;
        else
            distances ← inf;
        end
    end
end
i = 0;
if no detections then
    return ← no measurement data;
else
    while all elements of check do
        considering check's first element;
        forcluster ← distances which are less than vehiclelength;
        cluster the detections;
        while till all detection cluster do
            mean of all measurements;
        end
        remove checked detectionClusters;
    end
end
```

Algorithm 1: Detection clustering algorithm for same object

2.6.2 Kalman filter

Both sensors, LDS and camera had a noise for measurement. Moreover, their noise parameters were also provided in the manual and specifications. However, there should be experimental verification to do sensor calibration for testing their measurement noise. Therefore, this experimental verification was performed and determined the measurement noises for simulation and experimental applications. So, to smooth out and get rid of the measurement noises, the Kalman filter was the best choice to apply. Moreover, the Kalman filter also estimated the velocity through distance measurement from the sensors. This was important because velocity commands were sent to the follower vehicle by estimating lead vehicle velocity which was used in control algorithms. This filter solved both problems at one time.

Moreover, this system was linear because of longitudinal autonomy instead of lateral. Therefore, the state transition equation with added noise was:

$$x_t = A_t x_{t-1} + B_t u_t \tag{2.7}$$

and error covariance was:

$$P_t = A_t P_{t-1} A_t^T + Q_t \tag{2.8}$$

where x_t and x_{t-1} were the state vectors, and u_t was the control vector at time t. A_t was a square matrix of size $n \times n$ where n was the dimension of state vectors. B_t was of size $n \times m$ where m was the size of the control vector u_t. Moreover the measurement equation was also considered linear with added Gaussian noise and also used in update step:

$$z_t = C_t x_t + \delta_t \tag{2.9}$$

where C_t was a matrix of size $k \times n$ where k was the dimension of the measurement vector z_t. The vector δ_t was measurement noise where distribution was a multivariate Gaussian with zero mean and covariance Q_t. Then the innovation covariance was:

$$S_t = C_t P_t C_t^T + R_t \tag{2.10}$$

After that, optimal Kalman gain was found out:

$$K_t = P_t C_t^T S_t^{-1} \tag{2.11}$$

Then state estimation was updated:

$$x_{t+1} = x_t + K_t(z_t - C_t x_t) \tag{2.12}$$

After this, estimate covariance and then measurement was updated for next state estimation:

$$P_{t+1} = (I - K_t C_t) P_t \tag{2.13}$$

$$z_{t+1} = C_{t+1} x_{t+1} + \delta_{t+1} \tag{2.14}$$

Then loop over until the process completed. For more information about the Kalman filter algorithm and it's implementation for random uncontrolled forces while the vehicle on rails with initial position $[0, 0, 0, 0]^T$ is provided here [39][34]. For implementation example of the Kalman filter algorithm for the one-dimensional system was provided here [2] and governing equations for the two-dimensional system was provided here [4]. For implementing on the turtlebot, the state vector was $X = [x, y, v_x, v_y]^T$. Therefore the state transition equation was:

$$\begin{bmatrix} x_t \\ y_t \\ v_{xt} \\ v_{yt} \end{bmatrix} = A_t \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ v_{x(t-1)} \\ v_{y(t-1)} \end{bmatrix} + \epsilon_t \tag{2.15}$$

and for measurement equation was:

$$z_t = C_t \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ v_{x(t-1)} \\ v_{y(t-1)} \end{bmatrix} + \delta_t \tag{2.16}$$

Therefore, the matrices were:

$$A_t = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; C_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}; Q_0 = \begin{bmatrix} 0.0001 & 0 & 0 & 0 \\ 0 & 0.0001 & 0 & 0 \\ 0 & 0 & 0.0001 & 0 \\ 0 & 0 & 0 & 0.0001 \end{bmatrix};$$

$$P_0 = \begin{bmatrix} 0.0001 & 0 & 0 & 0 \\ 0 & 0.0001 & 0 & 0 \\ 0 & 0 & 0.0001 & 0 \\ 0 & 0 & 0 & 0.0001 \end{bmatrix}; R_0 = \begin{bmatrix} 0.0041 \end{bmatrix}$$

where dt is time difference the robot was moving to the next step. Moreover, ϵ_t was an error in the distance estimation for the system with zero mean in Gaussian distribution. Moreover, the covariance matrices R_t, Q_t, P_t were changing with the change in relative distance between lead and ego robot because this noise variation was provided in the specification document of the LDS sensor [6].

2.7 Proportional Control

One of the basic forms of a controller for a closed-loop system is a Proportional Controller. This controller was used to see the behavior of the system and its utilization on a larger scale. This controller acted like a correction controller which applied to the difference between the desired value of the system and the measured value from the sensors. The disadvantage of the proportional controller was that it does not eliminate the residual error in the system but it stabilizes it with the desired time period. The system will run with a particular error. Figure 2.8 shows the representation of controller in feedback loop. More information about this controller can be found here [42]. Mathematically, a proportional control algorithm is represented as:

$$P_{out} = K_p e(t), \tag{2.17}$$

where Kp is the proportional gain; $e(t)$ is the error value i.e. difference between desired and measured value in the system; P_{out} is the controller output to the system.

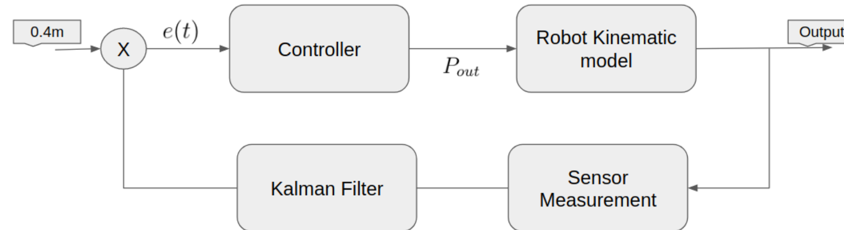

Figure 2.8: Representation of controller in the system.

In the experiment and simulation of the system, the desired value was 0.4 m where the following robot maintained that safe distance while measuring the relative distance between lead and following vehicle. This error between them was then applied with proportional gain to get the velocity output for the system to maintain that distance throughout the time-period. Moreover, the sensors already had an induced noise that was filtered with the Kalman filter. As the measurement values were smooth but there was no guarantee that the error in the system will be zero because there was also system process noise which might occur due to hardware of the robot or motor controller etc. As we saw that during the steady-state, there always be an error in the system whenever we use a proportional controller but it stabilizes the system well. In the experiments, the minimum error was set to the 0.01. However, if there was an increase of error more than 0.01 then proportional controller applied to the system send the velocity command to robot's motors. This command gradually increase the speed of the following robot to maintain a distance of about 0.4 m.

2.8 Proportional-Integral Control

After determining the parameter about the proportional gain for the system to stabilize it, then the Integral parameter was added to make a Proportional Integral Controller. Moreover, the robot output command was velocity which was working with the kinematic model as a

Table 2.4: Effects of increasing the parameters independently

Parameter	**Rise Time**	**Settling Time**	**Overshoot**	**stability**	**Steady error**
K_P	Decrease	Small change	Increase	Degrade	Decrease
K_I	Decrease	Increase	Increase	Degrade	Eliminate

second-order system. Therefore, Derivative control was not infused to the system. There are some advantages and disadvantages of this control such as this control can converge system steady-state error to zero but there was a windup error in the system which accumulated because of an integral term in the output of the system. This was eliminated by assigning system output for each iteration so that it will not accumulate for the next time-step. This also gives rise to an overshoot in the system and continues to increase until it was not unwound. More information is provided here [41]. Therefore, the controller output is given by

$$P_{out} = K_p e(t) + K_I \int e(t)\, dt, \tag{2.18}$$

where $e(t)$ is the error between the desired value and measured value in the system, K_p is proportional gain and, K_I is integral gain. In the experimental and simulation implementation, the desired value of 0.4 m relative distance was maintained during the time-period and reset the system output at every loop to avoid the windup error. The output from the controller was in the form of velocity to move the follower robot by maintaining a safe distance.

The tuning of the K_P and K_I parameter can get from Table 2.4 by trial and error. However, the steady state error can only be eliminated in step input to the system. A fast PI loop overshoots slightly to reach the desired value. However, in the practical application, the error may not be zero because of the noise in the system.

2.9 Model Predictive Control

Model Predictive Control (MPC) is a modern and advanced control strategy for a system that is used to control the process with constraints. It requires a plant model in the form of a dynamical/kinematic model to solve the process. It optimizes the current time-step while accounting the future time-steps. This process can be achieved in an infinite time horizon or finite time-step. After optimization it takes the control action in the current time-step, then it re-optimizes again.

MPC normally predicts the dependent variables of the system which are caused by independent variables. MPC uses current measurements, model of the system and process variables constraints to predict the future changes in the dependent variables in the system. The time period where the MPC predicts the dependent variable known as prediction horizon and shifts at every time -step when the control action is taken place which is called receding horizon control. For more general information about application of MPC can be found here [40] [20]. Figure 2.9 shown the similar state of the art implementation of the MPC which normally applied in this project.

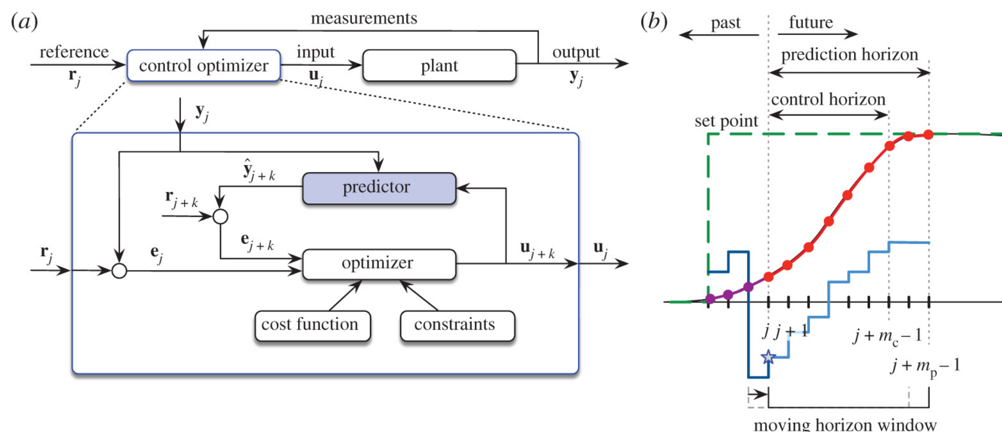

Figure 2.9: (a) control loop and (b) receding horizon representation in MPC [7].

In Figure 2.10, show the relative distance between the robots that the controller needs to maintain. The measurement was provided by the sensors of the robot and the plant

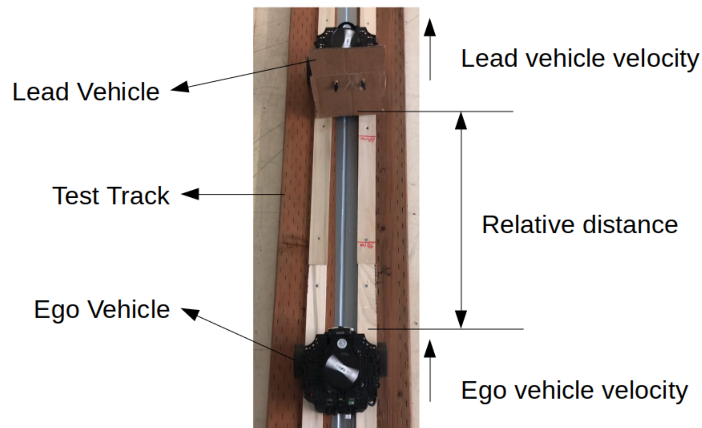

Figure 2.10: Maintain safe Relative Distance between the robots

model during the experiments was already there in the system but for simulations, it was designed as a rigid body kinematic model and control optimizer block represents the model predictive algorithm for the follower robot. From this state of the art, the predictor block in the system predicts the one time-step to take the control action for the following robot which was velocity command. From the optimizer, the cost function with constraints was defined to maintain the safe distance between the robots while executing the motion. The derivation of the Model Predictive control are:

Find V_{ego}:

$$\begin{aligned} J = \min_{d_{rel}} \quad & ||d_{rel} - d_{safe}||_2^2 \\ \text{subject to} \quad & d_{rel} \geq d_{safe}; \\ & 0 \leq V_{ego} \leq 0.2. \end{aligned} \tag{2.19}$$

Here, J is the cost function of the ego vehicle which was following the lead vehicle. d_{rel} is the relative distance between the robots, d_{safe} is the safe distance defined for the application, V_{ego} is the velocity of the ego vehicle. The cost function is minimizing the distance between the two robots to maintain a safe distance. The constraints mentioned in this were external constraints where the velocity of the robot should not exceed the 0.2m/s. this was the

hardware constraint on the following vehicle. The relative distance in the next time-step is given by

$$d_{rel}(t+1) = d_{rel}(t) + (V_{lead}(t+1) - V_{ego}(t+1))\Delta t \tag{2.20}$$

where Δt is the time difference between time executing the previous and current control action V_{lead} is the velocity of the lead vehicle and $(V_{lead}(t+1) - V_{ego}(t+1))$ is the relative velocity of the lead vehicle at time $t+1$. This relative velocity was estimated by the Kalman filter through the distance measurement through the sensors. Assuming the data is receiving from the sensors and considering that the velocity of the lead vehicle at the next time-step is the same as the previous one i.e. $V_{lead}(t+1) = V_{lead}(t)$ until there is any change in the Kalman filter estimation. Moreover, $d_{rel}(t)$ is known parameter from the sensors. Now replacing the d_{rel} with the $d_{rel}(t+1)$ to get the new form of cost function. Therefore, MPC was solving the ego velocity $V_{ego}(t+1)$ which was the control action for the following vehicle.

2.10 Adaptive Cruise Control

Adaptive Cruise Control (ACC) is the control system that adjusts the velocity of the vehicle by maintaining the safe distance from the lead vehicle. This type of controller normally uses the PID controller but the more complex dynamical system supports Model Predictive Control very well. As in section 2.9, the MPC controller can be model very easy to solve the optimization problem in each time-step to predict future control law. The general overview of the MPC in cruise control is

$$J = \min ||V - V_{set}||_2^2 \tag{2.21}$$

subject to,

$$D_{rel} - D_{safe} \geq 0,$$

$$-2 \leq a \leq 2.$$

where V_{set} is the velocity that controller needs to achieve while following the constraints on the relative distance that should be more than safe distance and acceleration a of the vehicle

in m/s^2. This acceleration determines by passenger comfortability to reach the objective of the cost function. More information about this system can be found here [38][10].

Chapter 3
APPLICATION

This project required both hardware and software development. Different hardware components were bought and assembled them to make a useful product to achieve the objectives of the project. Therefore, some of the small parts which can be 3D-print were manufactured for testing and evaluation to meet the goals of the project.

Moreover, software and hardware integration problems also solved to achieve goals in time. Robot Operating System (ROS) and MATLAB software integration were critical for this project because all algorithm development was done in MATLAB. ROS was used as a software and hardware communication interface. Finally, a wooden track was built to make turtlebot run in a longitudinal direction. Moreover, the simulation was also done to test and refine the control algorithms so that the robot will work efficiently. This saved a lot of time on system testing on the actual hardware. This also allowed to debug any problem in the system behavior of algorithm which was incorrect for the turtlebots.

3.1 Simulations

As simulations were primarily used to test and analyze the behavior of the control algorithms, therefore instead of using only Simulink blocks most of the MATLAB code was used in it. In the simulation, the actual velocity trajectory was used which was generated by moving the lead robot in the environment. Moreover, the sensor data was simulated by calculating the noise of both of the sensors. This was calculated while doing the sensor calibration. This sensor calibration method is explained in section 3.2.2 in detail. The system was designed as the closed feedback loop where sensor measurement was used as the feedback to the controller to actuate the kinematic model of the robot. Algorithms such as Kalman filter, controller,

etc. which were similar to the practical implementation of them. This also provided insights into the algorithm working in simulation and practical environment. As the system was dealing with longitudinal autonomy, the plant model was used as a kinematic model of a rigid body in the environment. Although turtlebot had a differential drive model for motion if there was any application of lateral autonomy in the system. Moreover, the simulation was carried out in one dimension instead of two dimensions which were implemented during the experiment. Therefore, the state space equation of the system was:

$$A = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix}; B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}; C = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

Moreover, this was a closed-loop system, a unit delay was added to the system for sensor

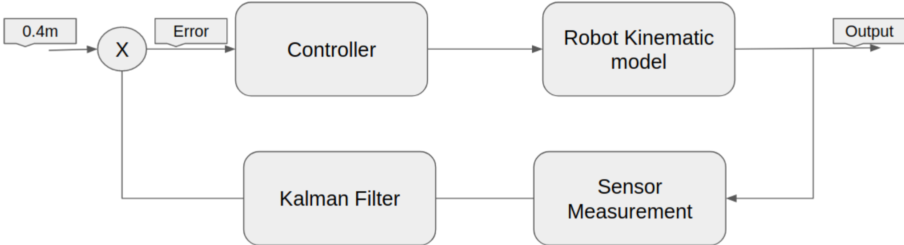

Figure 3.1: System Architecture design in Simulation.

feedback which allowed the system to respond after one time-step. The Kalman filter was used for smoothing and estimating the velocity of the lead robot from the distance measurement. For more information about the Simulink model of the system, it is provided in Appendix A and Figure 3.1 provides a general framework of the system. The time-step to run this system was determined through the practical implementation of the system which came out to be around $dt = 0.4$ sec. All the system implementation was in discrete instead of continuous. This time-step is high because of communication delays in the system while implementing in the hardware. This communication delay was there because the control algorithms were running in the remote PC i.e. Master PC and sending signals through Wifi

with the help of the ROS interface.

3.2 Experimental Setup

This experimental setup ranges from different domains of work from hardware and software implementation. These set up marked as the accomplishment for the project which might be used for future implementation and development for a vehicle to vehicle communication.

3.2.1 Turtlebot Setup

The hardware components of the turtlebots were bought to assemble them. These parts were Raspberry Pi 3 Model B, 360 Laser Distance Sensor LDS-01, an embedded board, actuators. But the first PC, SBC and OpenCR setup was done before hardware setup to make sure all software is working fine and then the test run was conducted. Moreover, there were many compatible devices such as sensors, on-board computers, etc and ROS packages of these devices can be found easily.

PC Setup

As the PC setup requires a lot of software installation from the operating system to the platform to connect with the robots. This setup procedure was followed from this [31] tutorial. This set up was only for host PC to control the robots. First Ubuntu 16.04[1] and ROS Kinetic Kame [31] was installed. This version is compatible with each other and their dependencies packages were suitable for the turtlebot3 platform. Robotis e-manual has created all dependencies package installation files including packages to control the turtlebot which come with ROS packages. After installing these packages then these are compiled and build with `catkin_make` command. Moreover, after this, a network configuration has to be set up which will communicate with the turtlebots. These can be done by changing the `ROS_MASTER_ URI` with the IP address of the host PC. Then, after configuring the network address, the bash file was sourced to save the changes on the remote PC.

Furthermore, the turtlebot was controlled through MATLAB. Therefore, a significant lesson learned while working with ROS and MATLAB connection. For setting up a connection between MATLAB and ROS, this [11] tutorial was followed. First, the ROS Master was created in MATLAB to communicate. However, before this, we need to set up the ROS environment in the MATLAB command line. Similar to the Linux machine, a network IP was provided which can transmit ROS messages to the robots to perform a certain task. This ROS IP environment can be set up by `setenv('ROS_IP','172.25.116.175')` command and then run the `rosinit` node to initialize the ROS communication between remote PC and the robot. For testing the signal transmission, this [15] tutorial was followed. After successfully achieving the communication channel, the ROS topic was evaluated to test the subscribing and publishing of the different topics of the robot. This was achieved by this [12] tutorial. First checked the `rostopic list` which we received from the robot and then subscribed to the topic to receive the signal of that topic in the remote PC and then publish the other ROS topic such as `/cmd_vel` to give control signals such as velocity to the robot. After this, the development of different algorithms was started to achieve the objective of the project.

Single Board Computer, OpenCR, Hardware Setup

SBC, Raspberry Pi 3 was used to run the commands to bring-up the sensors and actuators of the robots. Both Linux Operating System (Ubuntu Mate) was installed on the board. There was another option for Raspberry Pi to work with Raspbian which might have different functionality. Ubuntu Mate was using GNOME 2 with core applications. Moreover, there was the familiarity of Ubuntu and ROS packages were more compatible with Ubuntu OS. Furthermore, the ROS packages were also installed on the board. All required process is mentioned in this [29] tutorial.

OpenCR [26] was developed for ROS embedded systems which was compatible for Turtlebot3 packages. Also, the development environment was widely open from Arduino IDE. Therefore, Arduino software was installed in the Linux OS to install the OpenCR firmware

package which was provided by ROBOTIS. Moreover, after installing and uploading the firmware to OpenCR, check all the basic operations while connecting to a battery. For more information about the OpenCR Setup is mentioned here [25].

After setting up all the required software, all the components of the Turtlebots had been assembled. The Turtlebot hardware assembly was provided with Computer-Aided Design to perfectly fit all the parts at the correct position. Moreover, there is an open-source part development for these robots. This opens the opportunity to modify the components of the hardware and manufacture with operational needs. All Bill of Material and it's assembly manual is provided here [21]. Core components of the Turtlebot3 were Chassis, Motors, Wheels, Sensors, Battery, SBC, OpenCR. The Chassis which is called Waffle Plate plays an important role to bear all component weight. This plate is manufactured with injection molding to lower the manufacturing cost. Furthermore, the structure of the Turtlebot is customizable in different ways.

After all those setups, a raspberry pi camera v2 had been enabled by configuring the camera interface. After rebooting the system, the camera hardware was checked by clicking images. Then raspberry pi camera ROS packages were installed in Turtlebot to transmit signals to remote PC and test the functioning and camera data on remote PC. Moreover, default calibration files were stored in the right place to transmit the data. This procedure of setup is mentioned here [28]. after all this, algorithm development started in MATLAB from concept to application state.

3.2.2 Sensor Calibration Setup

After mounting all the components, Laser Distance Sensor (LDS) measured the distance of the objects around it. There was some Gaussian noise involved in this sensor to measure the distance. The specification of the LDS basic performance, measurement performance, mechanism layout, optical path, data information, pin description, and commands is mentioned here [5]. Moreover, there was an individual ROS package for this particular hardware. If someone wants to work with this sensor for their needs, they can build their packages out of

it. All the detailed documentation about the installation is provided here [22].

In the practical scenario, LDS measurements were tested on the test track with the measured and actual distance of the object. In figure 3.2, the plot is showing the measured distance by the sensor with the actual distance. After these observations, these data provided information about the nature of the change in measured distance with an actual one. Therefore, many new things can figure out to solve the problem of noise. As suggested in the documentation that there was a variable Gaussian noise according to distance measurement then a Kalman filter was designed to filter out this noise to make system estimate distance as accurate as possible.

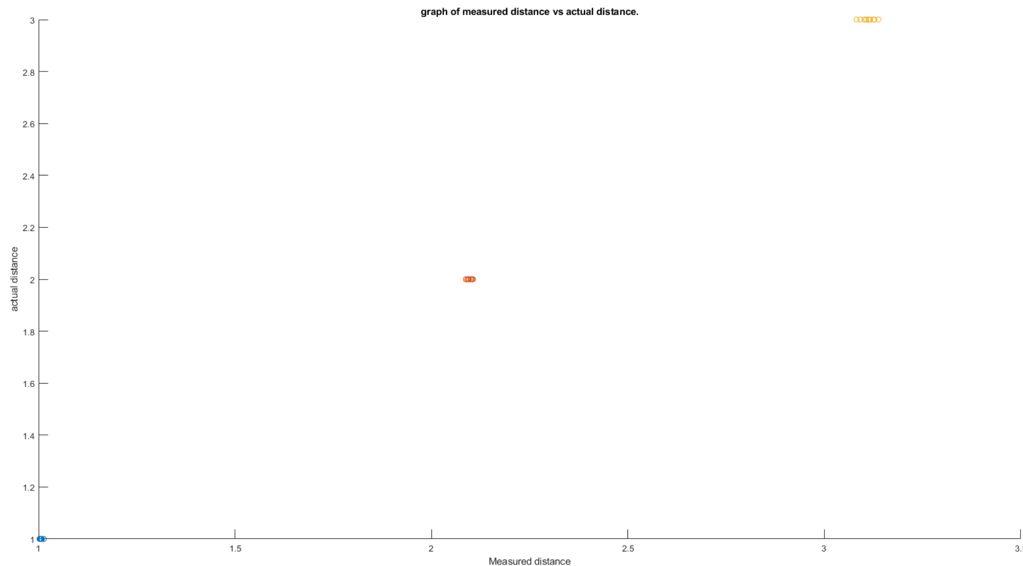

Figure 3.2: LDS estimated relative distance with actual distance

Now the camera was used in this system for increasing the probability of object position relative to Turtlebot. The algorithm was designed as mentioned in section 2.3 and section 2.4.3 about detection and distance estimation through the camera. From a monocular camera, the bounding box was created while detecting the object in front of the Turtlebot. This

rectangular bounding box in the image was used to estimate the distance in the real world. So, the experimental data was generated by measuring the distance v/s the width of the bounding box in the image. After generating the data, it was plotted to see the relation between them. This curve fitting method was good for some distance and providing a satisfactory result. This behavior is shown in Figure 2.6. To see the performance of the distance estimation, the lead robot was placed at a different known distance to measure the relative distance. In Figure, 3.3 shows the different readings of the sensors with their respective actual distance.

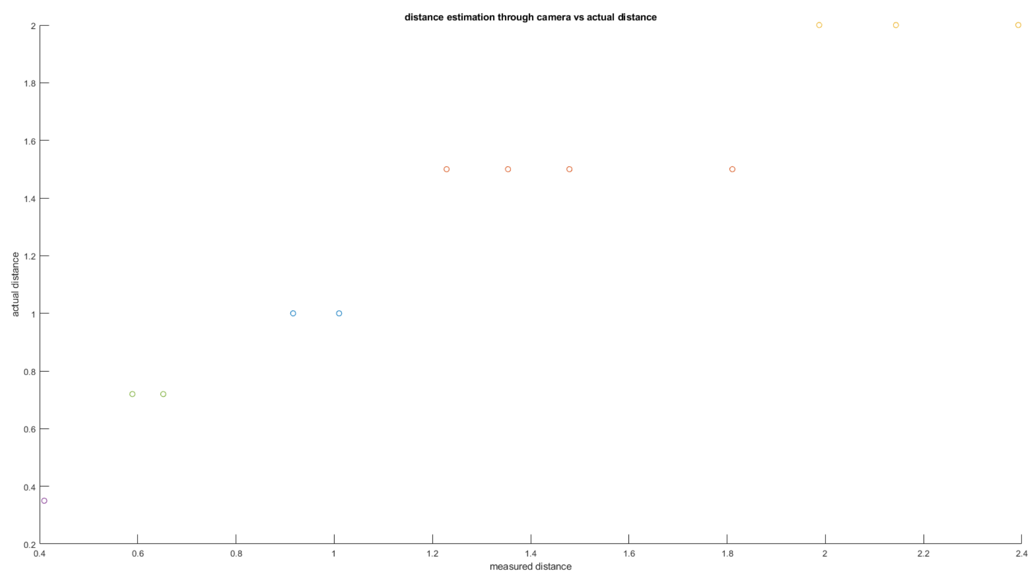

Figure 3.3: Vision estimated relative distance with actual distance

This curve fitting method was successful however, the only difficulty was that the bounding boxes detection is changing very frequently to estimate the distance. This means that at each frame of images the size of the bounding boxes will change and estimation of distances will also vary with frames. The only solution was to decrease the frame rate of the video to run the algorithm. However, there was no guarantee that the estimation will be correct

because the object distance in front was changing with time. This was the drawback of this algorithm for distance estimation through the camera.

3.2.3 ACC, KF, Track

Track

To align the Turtlebots to follow the same lane, a test track had been build. As lane keeping algorithm was not implemented in the Turtlebot, therefore it is better to restrict robots through the physical structure. This track was built by the wooden planks. To stabilize the movement of Turtlebots, care had been taken to align the strips in the best way possible. The wooden blocks are easily available online and just need to assemble the parts. The total length of track was about 6 m with 3 m planks which joined to each other with clips and 0.28 m wide. The dimension of the track is shown in Figure 3.5. The actual track is shown in Figure 3.4.

Kalman filter

To filter the noise of the system and estimate the velocity of the lead vehicle by observing the change in position through sensors. Therefore, to apply the Kalman filter which is defined in Chapter 2, A = State transition, C = Sensor observation, R = Sensor noise variance, Q = Process noise, P0 = Initial state estimate variance and sem = initial state matrix was defined. Then the filter was designed in advance to see the data. By applying the measurement step then, calculate the value of K = Kalman Gain. Then prediction Step is applied to update the state variance matrix. After getting the Kalman gain with definite iterations, the State matrix was updated to get the relative distance and estimated velocity of lead and ego Turtlebots. More information about the implementation of Kalman filter is given in this [2][4].

Figure 3.4: Complete $6m$ Turtlebot test track.

Model Predictive Control Implementation

After removing the noise and estimating the velocity through Kalman filter, Model Predictive Based Adaptive Cruise Control was applied. This algorithm used *mpcqpsolver* [14] which solve the quadratic programming problem using the KWIK algorithm [8]. Creating the Cost function which was mentioned in section 2.9 to solve the problem. Then inequality constraints A, b matrix was defined with no equality constraints. After this, the lower-

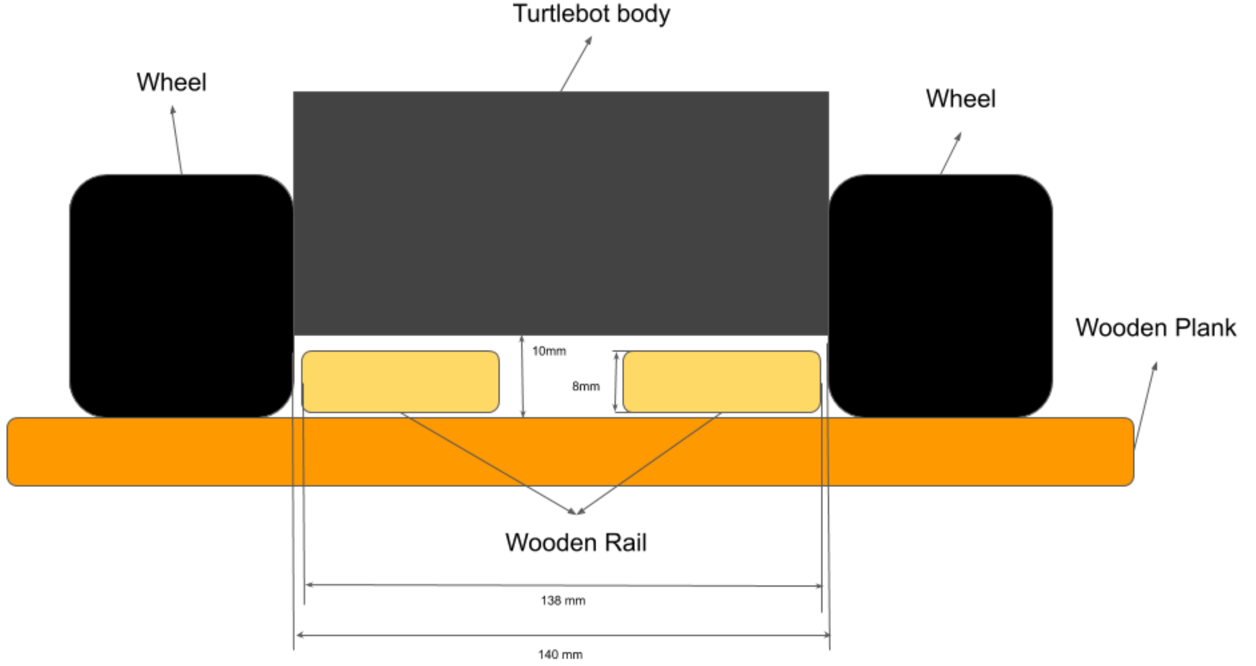

Figure 3.5: Cross-section concept view of the Turtlebot test track.

triangular Cholesky of Hessian H matrix was found, where H was a positive definite matrix. Then, the solver was cold started which means all inequality constraints as inactive. After this, the solver will output the optimal solution of the objective function with the feasibility of it. If the solution is feasible, then, velocity command was sent to ego turtlebot. The formulation of the model from section 2.9 to implementation to this solver is

$$J = \min \frac{1}{2} x^T H x + f^T x; \tag{3.1}$$

subject to,

$$Ax \geq b; A_{eq} x = b_{eq}$$

Therefore,

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 0.0001 \end{bmatrix}; f = \begin{bmatrix} -2d_{safe} \\ 0 \end{bmatrix}; A = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix}; b = \begin{bmatrix} 0 \\ -0.2 \end{bmatrix}$$

$$A_{eq} = \begin{bmatrix} 1 & \Delta t \end{bmatrix}; b_{eq} = d_{rel} + V_{lead}\Delta t; x = \begin{bmatrix} d_{rel} \\ V_{ego} \end{bmatrix}$$

Element at $(2, 2)$ in matrix H should be zero, but for practical implementation, that element was given value close to zero to get the solution from the solver.

Chapter 4
RESULTS AND DISCUSSION

4.1 Sensor Calibration

The first calibration of the sensor system was done to see the behavior and noise in them. The results at a different relative distance were plotted with the actual distance between lead and ego vehicle.

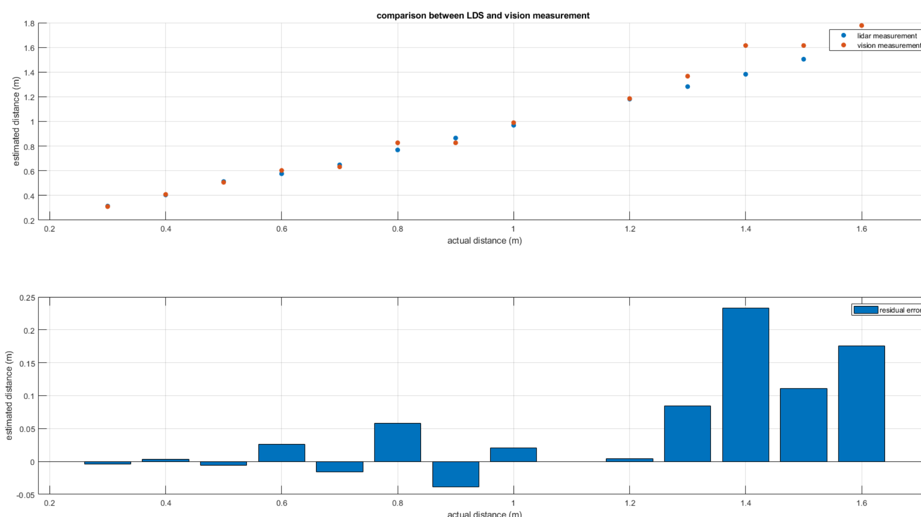

Figure 4.1: Comparison between LDS and camera relative distance.

Figure 4.1 shows relative distance measurement between ego and lead robots from LDS and camera separately v/s actual distance between them. As there is some deviation in the measurement of the sensors as compared to the actual distance between the bots. LDS measurements are more consistent as compared to vision detection measurement estimation. The residual error of measurements is very low when the relative distance is less than 1.2 m

and high after 1.2 m. Therefore, for experimental evaluations, the relative distance between turtlebots needs to be less than 1.2 m to avoid a drastic change in measurements or noise in the system.

4.2 Simulation Results

Simulation results of Proportional, Proportional-Integral and Model Predictive Control and Kalman filter estimation for relative distance. The behavior of the velocity of the ego vehicle while maintaining the relative distance between bots was analyzed. So, initially, a feedback system was designed without sensor noise to check the behavior of the system during P and PI controller by tuning it to minimize the error. Then from the calibration results, the noise of the sensor was calculated and applied to the system. After seeing the behavior of the system was not minimizing the relative distance error, again there was the tuning of parameters. Moreover, the delay in the system was adjusted according to the experimental system where the main delay in the system was due to communication. After all this process, the Kalman filter was added to the system to provide a smoother and better velocity estimation from it. These results can be seen in the following plots.

In Figure 4.2, there are three subplots, the first plot represents the relative distance of the robots through the sensors without Kalman filter. The second plot represents the distance estimation after the Kalman filter was applied to the system. The third plot represents the velocity of the ego and the lead vehicle with time-steps. If the ego vehicle is more than it's safe distance then its velocity is increased to maintain safe distance which was 0.4 m in this case. Those fluctuations of ego vehicle velocity are due to the noise in the system. As the proportional controller does not converge error to zero, this can be seen that over a time where there is some amount of error. This system is stable at a particular value. The relative distance is also having some fluctuation because of the noise. Moreover, if we see this plot on a large scale the error is very minuscule. Because the representation of the plot is in a very small distance therefore, a significant error is there. But at larger distances, it will be small and tolerable to control the robot.

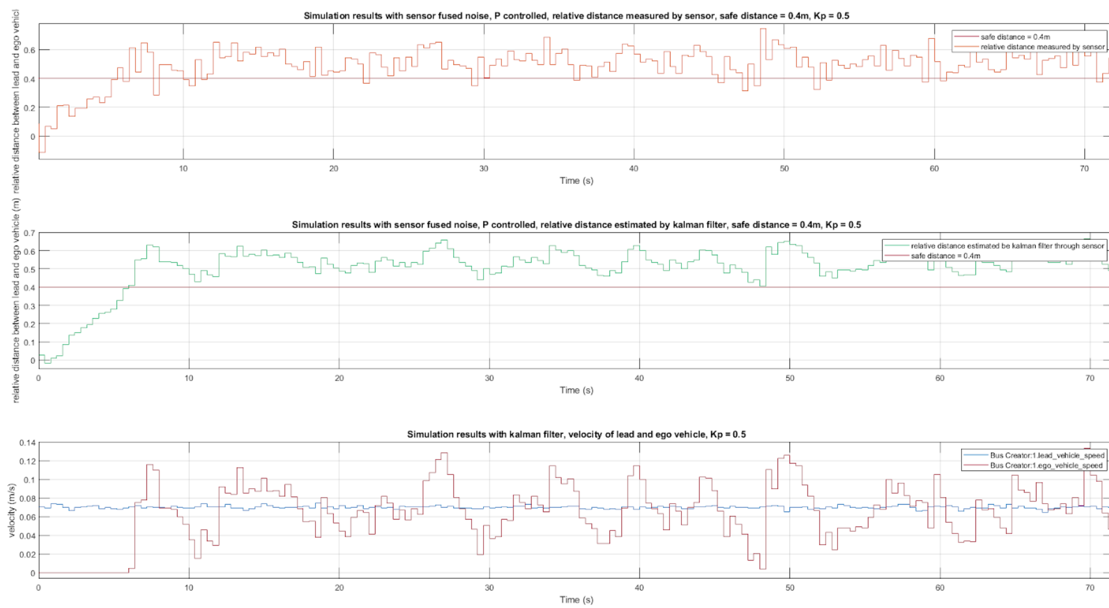

Figure 4.2: Proportional Controller to maintain safe distance when lead vehicle has constant velocity.

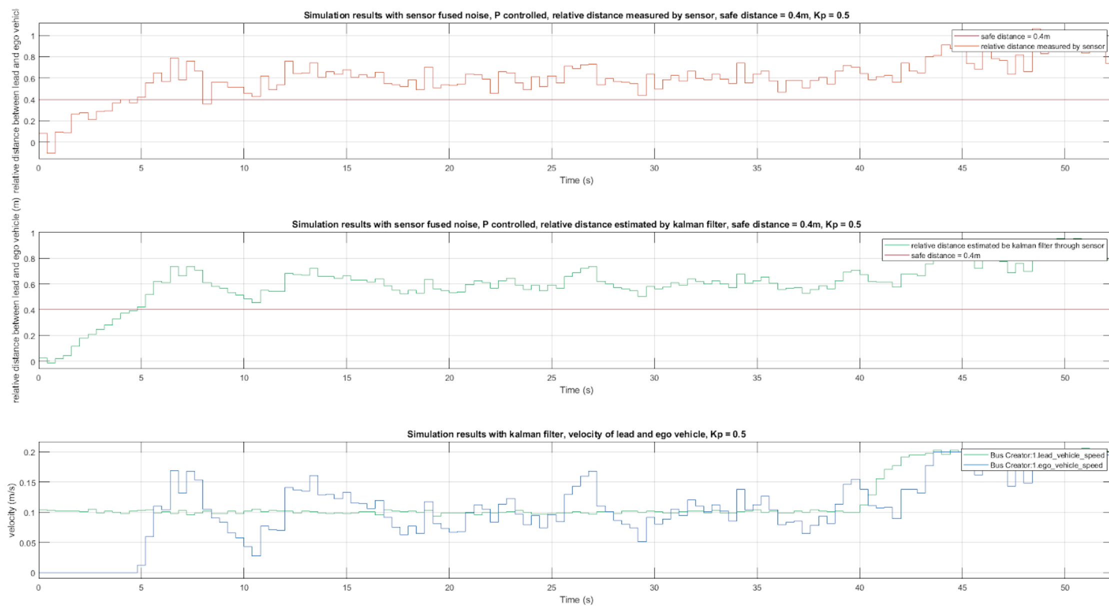

Figure 4.3: Proportional Controller to maintain safe distance when lead vehicle has ramp velocity.

Similarly, in Figure 4.3, the Proportional (P) Controller is trying to maintain the relative distance until the lead vehicle accelerates to achieve the maximum velocity of 0.2 m/s. Due to sudden change in the velocity of the lead vehicle ego vehicle also increased it's velocity to the maximum possible constraint. The error between the reference distance (0.4 m) and the relative distance between the robots increased because there is some time lag in the system to respond to the sensor feedback data. The relative distance increased until the ego vehicle reached its maximum velocity and the relative distance is constant in the next time-steps.

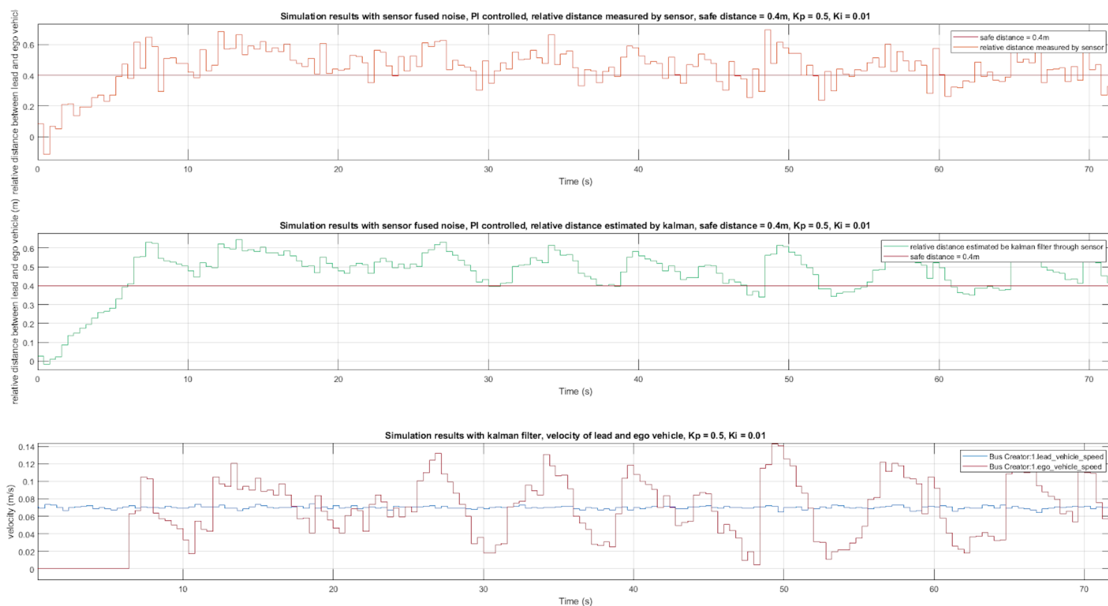

Figure 4.4: Proportional-Integral Controller to maintain safe distance when lead vehicle has constant velocity.

In Figure 4.4, the Proportional-Integral (PI) Controller ultimately converges the error to zero to maintain the relative distance. The velocity profile of the ego vehicle fluctuates to achieve that distance. Moreover, there is also a little bit of overshoot in the distance plot where relative distance is constant and then it converges afterward. Whenever the relative distance is equal to the safe distance between the ego and lead vehicle, the velocity is decreasing due to noise and time delay in the system.

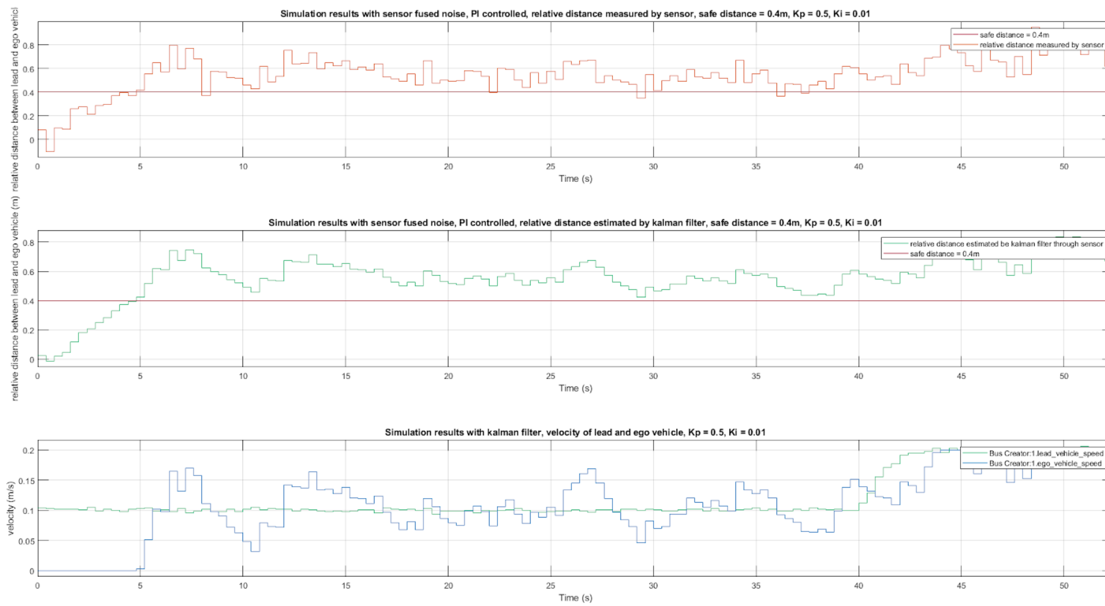

Figure 4.5: Proportional-Integral Controller to maintain safe distance when lead vehicle has ramp velocity.

Similarly in Figure 4.5, there is ramp velocity or sudden change in velocity of the lead vehicle where the ego vehicle is trying to catch up the safe distance but as the noise, time delay and constraint on maximum velocity, whatever relative distance is achieved by the ego vehicle until it attains its maximum velocity the relative distance becomes constant afterward. Moreover, the PI controller achieves error to zero. But in the P Controller, it is very difficult to achieve the error to converge. However, in the PI controller there is an overshoot but after some point of time that is not important as the system is achieving its objective.

In Figure 4.6, Model Predictive Controller is very aggressive to achieve the objective to maintain a safe distance. It is directly giving velocity output to its maximum value to attain the zero error in the system. Moreover, due to time delay, the system is also going to negative error. Because the previous velocity is already executing by the system. This again leads to minimizing the velocity of ego vehicle and the same sequence is followed. As the leading vehicle is running in constant velocity but ego vehicle is achieving its objective

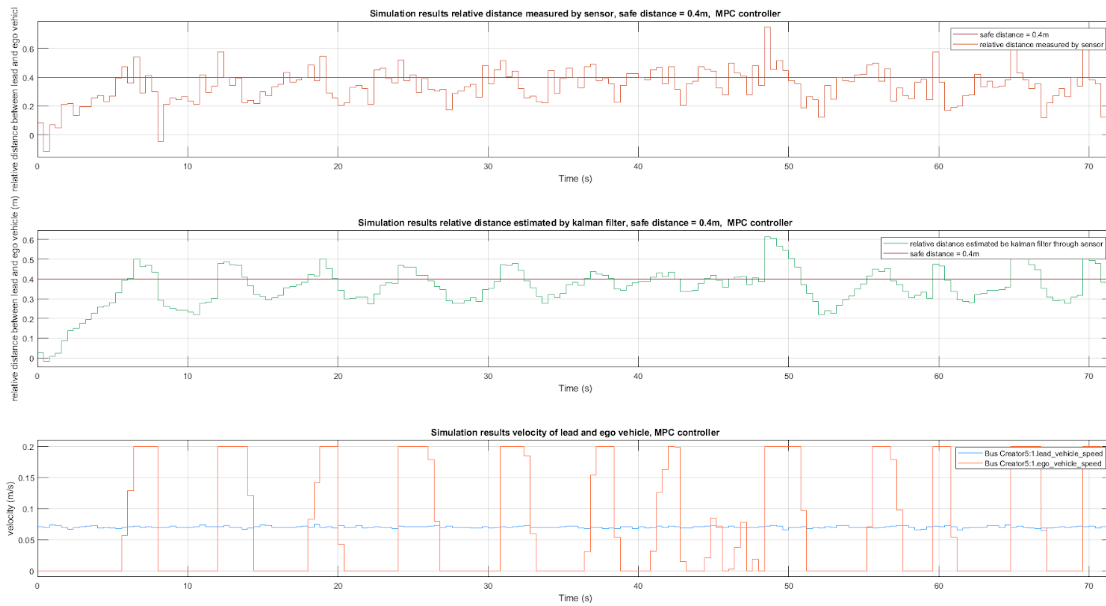

Figure 4.6: Model Predictive Controller to maintain safe distance when lead vehicle has constant velocity.

in minimum time. Moreover, the system is marginally stable because there is a bounded velocity output for the robots. When comparing with the other two controllers, the system is fast in achieving the objective. This makes Model Predictive Control take less time to achieve the goal of the system and more accuracy can be provided in the system.

In Figure 4.7, the system is behaving similarly when comparing to the constant lead vehicle velocity. When there is a sudden increase in the velocity of the lead vehicle, the system also achieves the highest velocity of the ego vehicle to minimize the error. But the error will be constant throughout the duration of time because there always be the constant velocity of both robots and the relative distance will be the same until the lead vehicle stop.

4.3 Experimental Results

In Figure 4.8, the relative distance between lead and ego vehicle, velocity estimated by the controller are recorded. Moreover, the blue color line represents in the first plot referred to as safe distance = 0.4 m and in the second plot as the constant velocity of the lead vehicle around

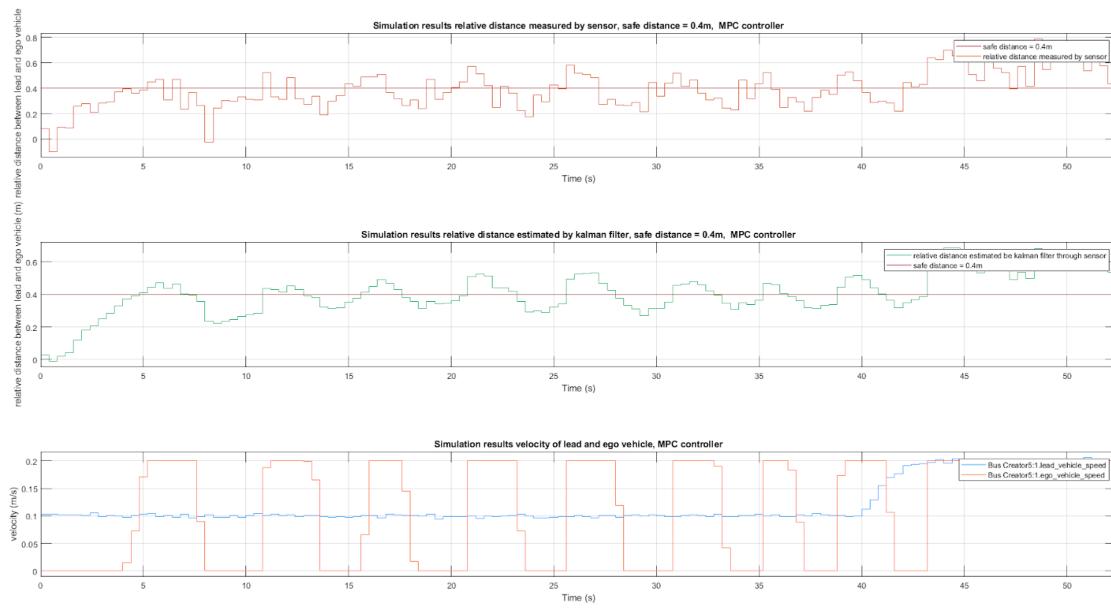

Figure 4.7: Model Predictive Controller to maintain safe distance when lead vehicle has ramp velocity.

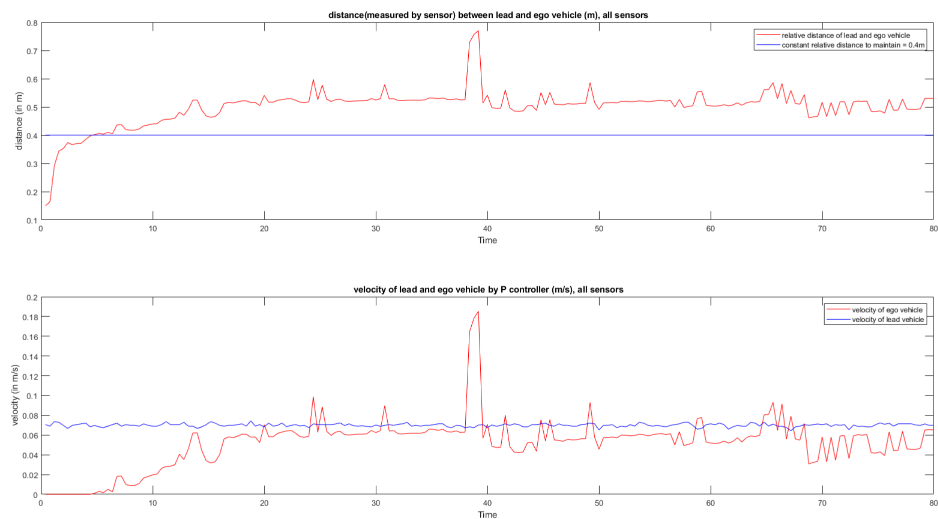

Figure 4.8: Experimental implementation Proportional Controller to maintain safe distance when lead vehicle has constant velocity.

0.07 m/s. The only fluctuation is because of camera detection which was changing with each frame by changing the dimensions of the bounding box for detection and therefore estimating the different relative distance between ego and the lead vehicle. Moreover, there is a peak in the middle of the plot because LDS had detected the other object in the environment close in the y-axis and farther away in the x-axis from the lead vehicle. While talking about the Proportional Controller, the behavior of the system is similar to the simulation results with less noise and more converged to the safe distance. Moreover, there is always an error in the system throughout time. This indicated the Proportional controller is working as expected to achieve its objective.

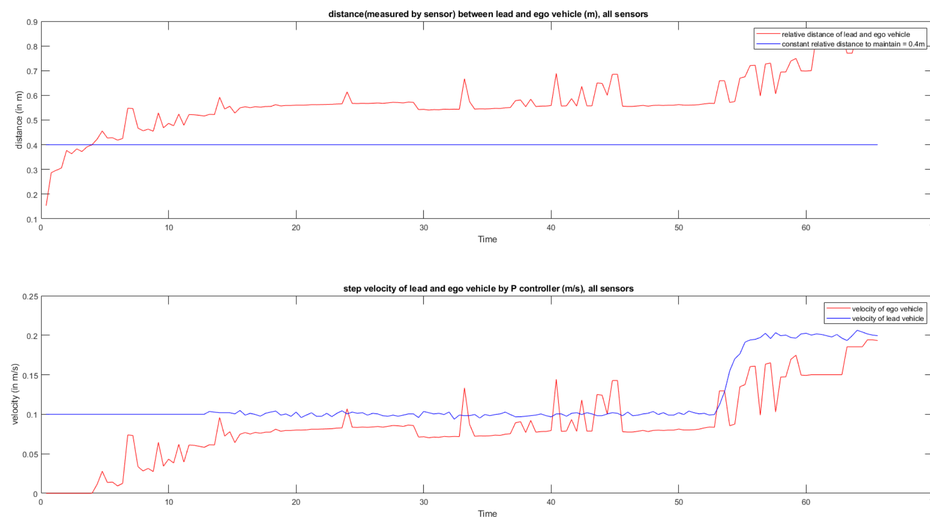

Figure 4.9: Experimental implementation Proportional Controller to maintain safe distance when lead vehicle has step velocity.

In Figure 4.9, the reference lines are the same with the variable velocity of the lead vehicle which was 0.1 m/s and 0.2 m/s. The ego vehicle had constraint of maximum velocity= 0.2 m/s. So, when lead vehicle velocity = 0.2 m/s, the ego vehicle was reaching to the constant maximum velocity of about 0.2 m/s. Therefore, the distance between the ego and lead vehicle is increasing because the ego vehicle is trying to catch up with the safe distance

due to the sudden increase in the velocity. This behavior is also similar to the simulation results.

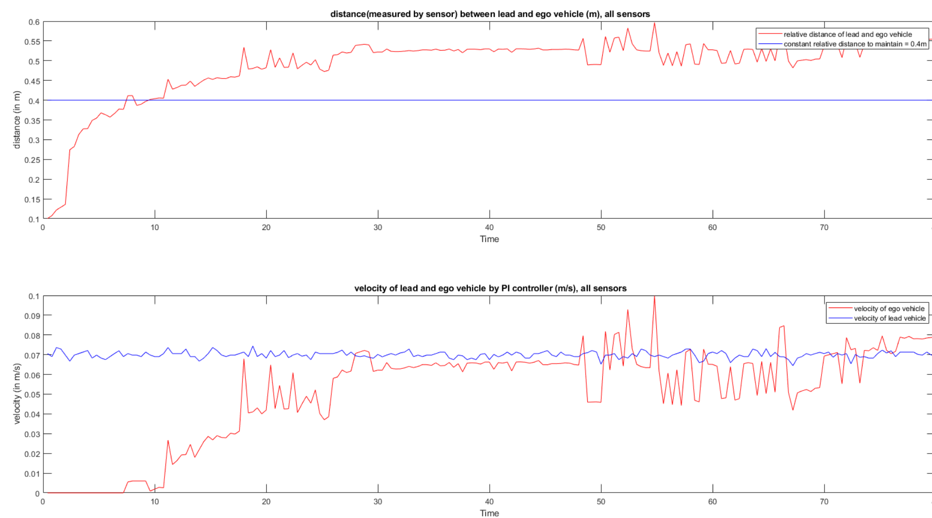

Figure 4.10: Experimental implementation Proportional-Integral Controller to maintain safe distance when lead vehicle has constant velocity.

In Figure 4.10, the output of the Proportional-Integral controller is taking time to converge the error to zero. After some point of time, there is a decrease in the relative distance of the robots but due to some windup error, this system is not converging early. Although the care had been taken to rule out any windup error in the system. Moreover, during the implementation of anti-wind up error the error was put till 1m which is way to large and therefore, there is significant amount of error while system was going to steady state. After time 45sec, there was some fluctuation in the detection of the lead turtlebots by the camera sensors which results in fluctuating velocity in the system.

In Figure 4.11, there was an increase in the lead vehicle velocity which also makes the error more than it should be because sensors take time to estimate the correct relative distance and velocity estimation from Kalman filter. But eventually, the PI controller sent the velocity command to the ego vehicle to match up with maximum velocity. This makes the relative

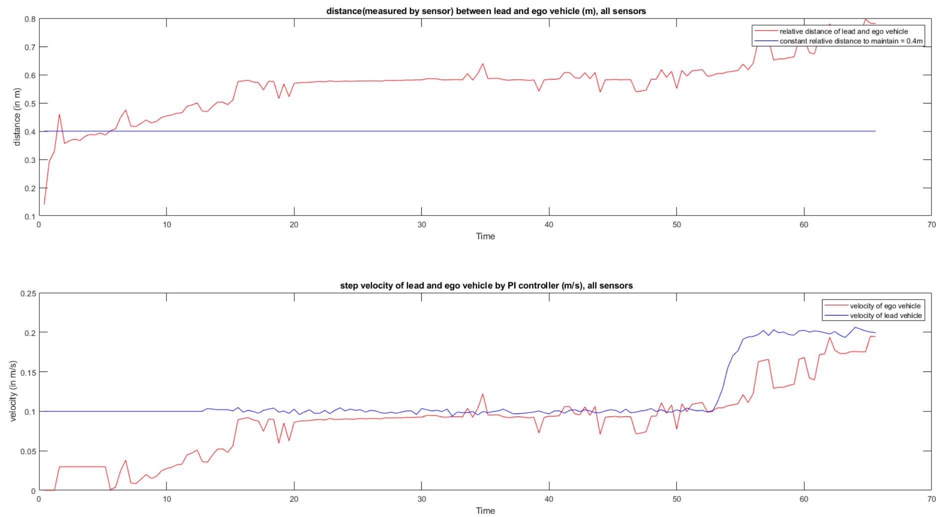

Figure 4.11: Experimental implementation Proportional-Integral Controller to maintain safe distance when lead vehicle has ramp velocity.

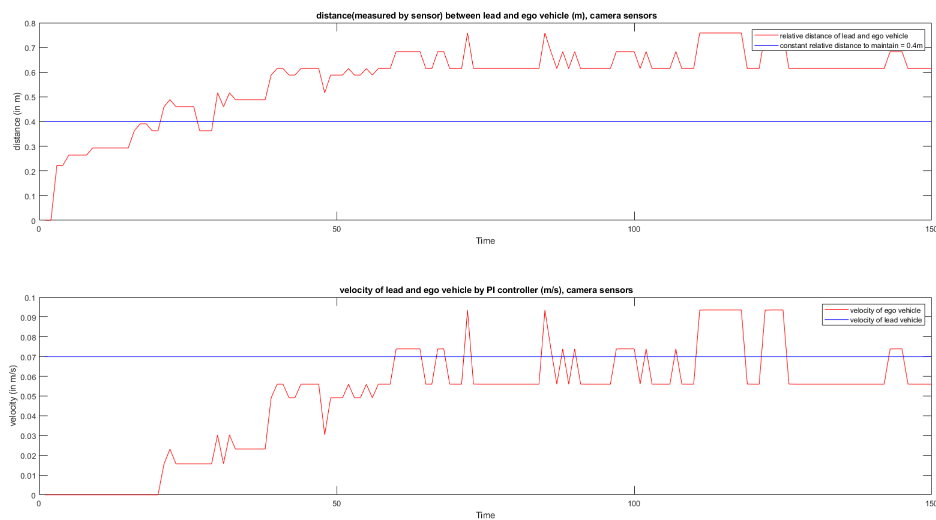

Figure 4.12: Experimental implementation Proportional-Integral Controller to maintain safe distance when lead vehicle has constant velocity through camera sensor only

distance between both the robots constant throughout time. In figure 4.12, showing the nature of the PI controller when the only camera is working. As there are different forms of estimating the relative distance by the camera, therefore while fusing this data with LDS, it is also incorporating that type of behavior into the system which leads to different velocity estimation by the Kalman filter and the output from the controller.

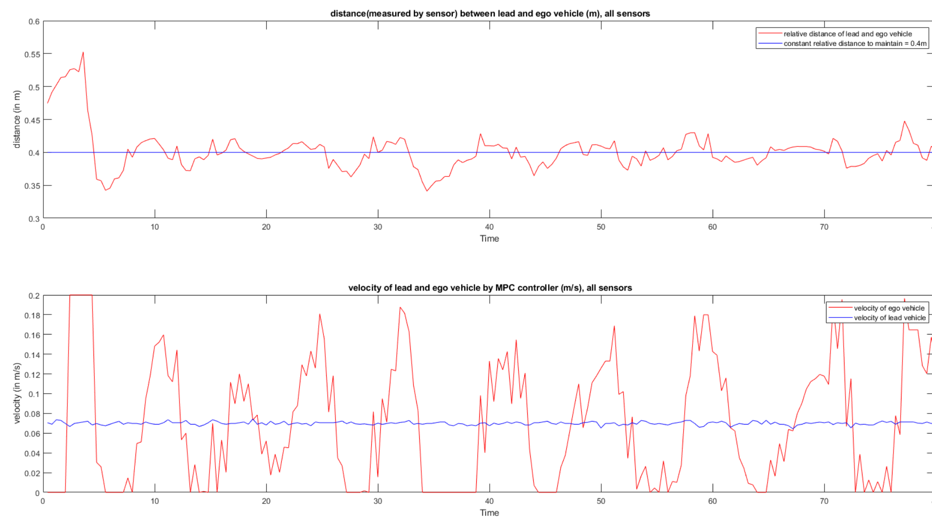

Figure 4.13: Experimental implementation Model Predictive Controller to maintain safe distance when lead vehicle has constant velocity.

In Figure 4.13, the Model Predictive Controller started when the lead robot was more than the safe distance provided to the system. Therefore, the velocity of the ego vehicle jumps to its maximum point to minimize the error. Due to communication delay in the system, the velocity and sensor feedback command execution is taking time. This also provided some negative error where the ego vehicle stops and waiting for the lead vehicle to proceed. Moreover, the MPC is trying to maintain that safe distance instead of P and PI controller which was taking time for them. Moreover, their error is very difficult to achieve zero. While comparing with the simulation results, this result is very similar to them. Only the velocity of the ego vehicle is not achieved to maximum possible value at every time frame.

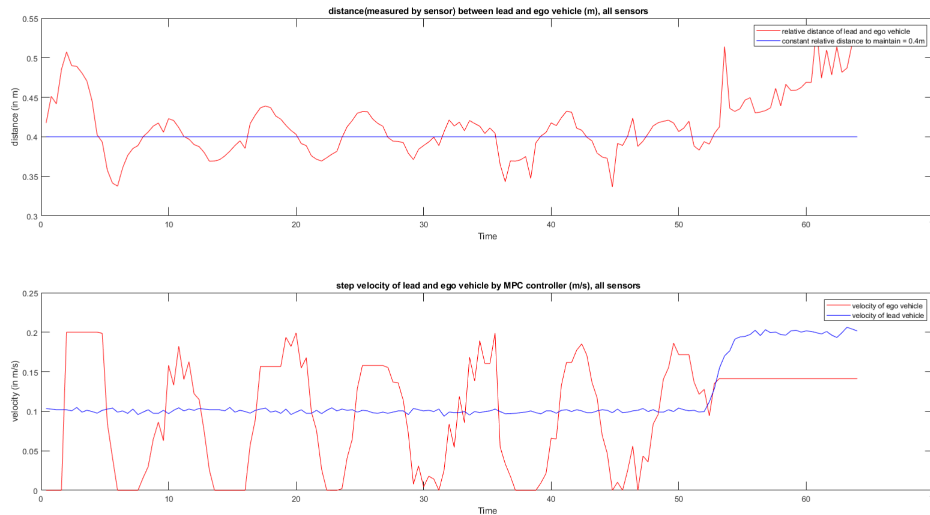

Figure 4.14: Experimental implementation Model Predictive Controller to maintain safe distance when lead vehicle has step velocity.

Similarly in Figure 4.14, where the lead vehicle velocity was increased to the 0.2 m/s and ego vehicle trying to catch up the velocity and minimize the error. As there was a constraint on the ego vehicle to achieve its maximum velocity, therefore it is following the lead vehicle with constant velocity and Kalman filter might not be correctly estimating the velocity of the lead vehicle. While comparing with the simulation result, it is similar. This also provided the nature of simulation and experimental results match with a lot of constraints on the system while performing experiments.

Now, compare the error in both the simulation and the experimental results. In simulation results for P control was around 0.1m to 0.2m and in experiments, error was around 0.1m. Moreover, for the PI controller, in simulations and experiments, the error was around 0 to 0.2m. Lastly, for MPC, in simulation, the error was around -0.1m to 0.1m and in experiments, the error was around -0.05m to 0.05m. The error in experiments is smaller then simulation because in experiments the there is a variable delay in the system up-to 0.4sec however in simulations there is a constant delay of 0.4sec.

Chapter 5

CONCLUSION AND FUTURE WORK

Robot Operating System (ROS) and MATLAB platforms are used to control the robots which have some delays in communication between robots and host PC. This project also tried to calibrate the sensors on the ego vehicle. This made the more robust estimation of the relative distance between the ego and the lead vehicle. Moreover, these sensors are independent of each other. So, if there is any fault in any of the sensors, the system will work and estimate the distance. As the distance measurement was observed in the system, the velocity of lead can be estimated through Kalman filter. This will help the control algorithm to have a good estimate of lead vehicle velocity and follow it. The results show the behavior of the ego vehicle to follow the lead vehicle to maintain the safe distance between the turtlebots. Furthermore, velocity estimation from Proportional, Proportional Integral and Model Predictive Control provides sufficient time for ego vehicle to adapt the lead vehicle.

In the future, more sensors can be introduced to the system to make a more robust estimation of the relative distance of the vehicle. These sensors might be a depth sensor camera or stereo vision to eliminate the sudden change of relative distance through the camera. Moreover, this update will provide a smooth implementation of control algorithms. Therefore, to work with this sensor Graphical Processing Unit(GPU) is required to process the image processing algorithms. So, one of the Nvidia board such as Jetson nano or TX2 can be used for future research. After this, vehicle to vehicle (V2V) communication can be set up to make a system that will accurately predict and follow the lead vehicle. This V2V communication will further help for the transmission of the lead vehicle information to ego vehicle. The Model Predictive Control can be improved by increasing the prediction

horizon so that it estimates more precise velocity. But increasing the prediction horizon will lead to high computational cost and would need a more powerful board to execute the control action very fast. Furthermore, these algorithms will be implemented in actual cars to commercialize this system and make an impact on society. Moreover, this research will also reduce the energy consumption of the vehicle by running them with optimal velocity on highways.

BIBLIOGRAPHY

[1] Canonical Web Team. Install ubuntu 16.04 desktop: Ubuntu tutorials. https://tutorials.ubuntu.com/tutorial/tutorial-install-ubuntu-desktop-1604#0. [Online; accessed 12-July-2019].

[2] Chris Atkeson. Kalman filter velocity estimation example. https://www.cs.cmu.edu/~cga/dynopt/kalman/kalman.html. [Online; accessed 12-July-2019].

[3] Piotr Dollar, Ron Appel, Serge Belongie, and Pietro Perona. Fast feature pyramids for object detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36(8):1532–1545, August 2014.

[4] Emmanuel Candes. Acm 116: The kalman filter. https://statweb.stanford.edu/~candes/acm116/Handouts/Kalman.pdf. [Online; accessed 12-July-2019].

[5] Hitachi-LG Data Storage. Lds1.5 specifications model : Hls-lfcd2. http://emanual.robotis.com/assets/docs/LDS_Basic_Specification.pdf. [Online; accessed 12-July-2019].

[6] HL Data Storage. Lds-01. http://emanual.robotis.com/assets/docs/LDS_Basic_Specification.pdf. [Online; accessed 12-July-2019].

[7] E Kaiser, J. Nathan Kutz, and Steven L. Brunton. Sparse identification of nonlinear dynamics for model predictive control in the low-data limit. In *Proceedings. Mathematical, physical, and engineering sciences*, 2018.

[8] Lihong Li, Michael L Littman, Thomas J Walsh, and Alexander L Strehl. Knows what it knows: a framework for self-aware learning. *Machine learning*, 82(3):399–443, 2011.

[9] Mathworks. Adaptive cruise control with sensor fusion. https://www.mathworks.com/help/driving/examples/adaptive-cruise-control-with-sensor-fusion.html. [Online; accessed 12-July-2019].

[10] Mathworks. Adaptive cruise control with sensor fusion. https://www.mathworks.com/help/driving/examples/adaptive-cruise-control-with-sensor-fusion.html. [Online; accessed 12-July-2019].

[11] Mathworks. Connect to a ros network - matlab and simulink. https://www.mathworks.com/help/robotics/examples/connect-to-a-ros-network.html. [Online; accessed 12-July-2019].

[12] Mathworks. Exchange data with ros publishers and subscribers. https://www.mathworks.com/help/robotics/examples/exchange-data-with-ros-publishers.html. [Online; accessed 12-July-2019].

[13] Mathworks. Get started with the image labeler. https://www.mathworks.com/help/vision/ug/get-started-with-the-image-labeler.html. [Online; accessed 12-July-2019].

[14] Mathworks. mpcqpsolver. https://www.mathworks.com/help/mpc/ref/mpcqpsolver.html. [Online; accessed 12-July-2019].

[15] Mathworks. rosinit. https://www.mathworks.com/help/robotics/ref/rosinit.html. [Online; accessed 12-July-2019].

[16] Mathworks. trainacfobjectdetector. https://www.mathworks.com/help/vision/ref/trainacfobjectdetector.html. [Online; accessed 12-July-2019].

[17] Aakar Mehra, Wen-Loong Ma, Forrest Berg, Paulo Tabuada, Jessy W. Grizzle, and Aaron D. Ames. Adaptive cruise control: Experimental validation of advanced controllers on scale-model cars. *2015 American Control Conference (ACC)*, pages 1411–1418, 2015.

[18] Gjl Gerrit Naus, Jeroen Ploeg, M.J.G. van de Molengraft, W. P. Maurice H. Heemels, and M Maarten Steinbuch. A model predictive control approach to design a parameterized adaptive cruise control. 2010.

[19] Open Source Robotics Foundation. navigation/tutorials/robotsetup/odom. http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom. [Online; accessed 12-July-2019].

[20] James B. Rawlings. Tutorial overview of model predictive control. *IEEE Control Systems Magazine*, 20(3):38 – 52, 2000.

[21] ROBOTIS e-Manual. Hardware setup. http://emanual.robotis.com/docs/en/platform/turtlebot3/hardware_setup/#part-list. [Online; accessed 12-July-2019].

[22] ROBOTIS e-Manual. Lds-01. http://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/. [Online; accessed 12-July-2019].

[23] ROBOTIS e-Manual. Lds1.5 specifications. http://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/. [Online; accessed 12-July-2019].

[24] ROBOTIS e-Manual. Opencr 1.0. http://emanual.robotis.com/docs/en/parts/controller/opencr10/. [Online; accessed 12-July-2019].

[25] ROBOTIS e-Manual. Opencr setup. http://emanual.robotis.com/docs/en/platform/turtlebot3/opencr_setup/#opencr-setup. [Online; accessed 12-July-2019].

[26] ROBOTIS e-Manual. Opencr1.0. http://emanual.robotis.com/docs/en/parts/controller/opencr10/. [Online; accessed 12-July-2019].

[27] ROBOTIS e-Manual. Raspberry pi camera. http://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_raspi_cam/. [Online; accessed 12-July-2019].

[28] ROBOTIS e-Manual. Raspberry pi camera. http://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_raspi_cam/. [Online; accessed 12-July-2019].

[29] ROBOTIS e-Manual. Sbc setup. http://emanual.robotis.com/docs/en/platform/turtlebot3/sbc_setup/#sbc-setup. [Online; accessed 12-July-2019].

[30] ROBOTIS e-Manual. Specifications. http://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/. [Online; accessed 12-July-2019].

[31] ROBOTIS e-Manual. Pc setup. http://emanual.robotis.com/docs/en/platform/turtlebot3/pc_setup/, 2019. [Online; accessed 12-July-2019].

[32] ROS.org. Distributions. http://wiki.ros.org/Distributions. [Online; accessed 12-July-2019].

[33] ROS.org. Ros/concepts. http://wiki.ros.org/ROS/Concepts. [Online; accessed 12-July-2019].

[34] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. The MIT Press, 2005.

[35] UbiquityRobotics. Ubiquityrobotics raspicam node. https://github.com/UbiquityRobotics/raspicam_node. [Online; accessed 12-July-2019].

[36] Ziran Wang, Guoyuan Wu, and Matthew J. Barth. A review on cooperative adaptive cruise control (CACC) systems: Architectures, controls, and applications. *CoRR*, abs/1809.02867, 2018.

[37] Shouyang Wei, Yuan Zou, Tao Zhang, Xudong Zhang, and Wenwei Wang. Design and experimental validation of a cooperative adaptive cruise control system based on supervised reinforcement learning. *Applied Sciences*, 8(7):1014, 2018.

[38] Wikipedia community. Adaptive cruise control. `https://en.wikipedia.org/wiki/Adaptive_cruise_control`. [Online; accessed 12-July-2019].

[39] Wikipedia community. Kalman filter. `https://en.wikipedia.org/wiki/Kalman_filter`. [Online; accessed 12-July-2019].

[40] Wikipedia community. Model predictive control. `https://en.wikipedia.org/wiki/Model_predictive_control`. [Online; accessed 12-July-2019].

[41] Wikipedia community. Pid controller. `https://en.wikipedia.org/wiki/PID_controller`. [Online; accessed 12-July-2019].

[42] Wikipedia community. Proportional control. `https://en.wikipedia.org/wiki/Proportional_control`. [Online; accessed 12-July-2019].

Appendix A

CODE, SIMULATION MODELS AND PROJECT REPOSITORIES

- Matlab code and function to run the turtlebots

1) The following is the main program of the system file to run the ego vehicle.

```
%% main file to execute different control algorithms
% make sure to initialize the ros master node by "rosinit" before
% executing it.


robot = rospublisher('/cmd_vel') ;
velmsg = rosmessage(robot);
laser = rossubscriber('/scan');
odom_info = rossubscriber('/odom');


if(ismember('/raspicam_node/image/compressed', rostopic('list')))
    imsub = rossubscriber('/raspicam_node/image/compressed');
    flag =1;
    load('use_for_turtlebot_detection.mat');
end


X = [-60:1: 59];
robvel = 0.0;  %Set robot speed command.


C = [1, 0, 0, 0;
```

```
        0, 1, 0, 0];
%R = [0.000225]; %old value
R = [0.0041];
Q = [0.0001, 0, 0, 0;
      0, 0.0001, 0, 0;
      0, 0, 0.0001, 0;
      0, 0, 0, 0.0001];
P0 = [0.0001, 0, 0, 0;
       0, 0.0001, 0, 0;
        0, 0, 0.0001, 0
        0, 0, 0, 0.0001];

sem = [0, 0, 0, 0]';

sem_vis = [0, 0]';
V_x_rel_save = [];
v_kalman_save = [];
V_x_rel_odom = [];
x_rel_save = [];
tStart = tic;
Measurement_vis =[0,0];
detection_meas =[];
lds_mean_measurement_save = [];
vis_measurement_mean_save = [];
 while(flag)
     msg_odom = receive(odom_info);
     V_x_rel_odom = [V_x_rel_odom; msg_odom.Twist.Twist.Linear.X];
```

```
    save('V_x_rel_odom_save_plot.mat', 'V_x_rel_odom');
    img = receive(imsub);
    img.Format = 'bgr8; jpeg compressed bgr8';
    I = readImage(img);

    bboxes = detect(acfDetector, readImage(img));
    if ~isempty(bboxes)
        vis_xrel = plot_pix_vs_dist(bboxes(3));
        y_vis = y_coord_estimation_vision_410X308
        ((bboxes(1)+(bboxes(3)/2)), vis_xrel);

        Measurement_vis = [vis_xrel, y_vis];
       %disp(['##1 -- vision distance : ',
        num2str(Measurement_vis)])
        vis_measurement_mean_save = [vis_measurement_mean_save;
        Measurement_vis];
    end
    I = insertObjectAnnotation(I,'rectangle',bboxes,
    'Turtlebot');
    imshow(I);
    scan = receive(laser,3);
%% Mimicing 30 degree angle for radar sensor
    velmsg = rosmessage(robot);
    scan.Ranges(16:345) = 0;
    % for only 30 degree range others are 0.

    new_lds(1:15) = flip(scan.Ranges(1:15));
```

```
new_lds(16:30) = flip(scan.Ranges(346:360));

lds_coord = lds_convert(new_lds');
% Calculating the coordinates
%of the detected objects.

        % concatenation for measurement of form [x, y]
        %% adding only non zeroes values, after coordinate
        %calculation
       Measurement_lds = [lds_coord(lds_coord(:, 1)~=0, :)];
       % compensating the 10 cm distance
       Measurement_lds = Measurement_lds -[0.1, 0];
       lds_measurement_mean = mean(Measurement_lds,1);
       lds_mean_measurement_save=[lds_mean_measurement_save;
       lds_measurement_mean];
       %save('turtlebot_lds_detect_data.mat',
       'Measurement_lds');
       % passing measurement values to clustering algorithm
       if exist('Measurement_lds','var') == 1
           num_point = size(Measurement_lds);
       else
           num_point=1;
       end
       if num_point(1,1) ~=1
         for det = 1: num_point(1, 1)
             sensors{det, 1} = struct('SensorIndex', 1,
             'Measurement',
```

```
        Measurement_lds(det, :)', 'ObjectClassID', 1);
    end
end
if ~isempty(Measurement_vis)
    sensors{num_point(1,1)+1, 1} = struct('SensorIndex',
    2, 'Measurement', Measurement_vis',
    'ObjectClassID', 1);
end
if exist('sensors','var') == 1
    detections = [sensors{:, 1}];
else
    detections= struct('SensorIndex', 1,
    'Measurement', [0,0]',
    'ObjectClassID', 1);
end

vehicleLength = 0.20;
detectionClusters = cluster_lds_new(detections,
vehicleLength);
num_detect = size(detectionClusters);
for i=1:num_detect
  detection_meas = [detection_meas,
  (detectionClusters{i, 1}.Measurement)];
end

mean_measurement = mean(detection_meas, 2);
%detection to sensor fusion.
```

```
    % creating noise parameters which depend on measurement:
if detectionClusters{1, 1}.Measurement(1, :) >= 0.5
Q = [(0.035 * detectionClusters{1, 1}.Measurement(1, :))^2,
    0, 0, 0;
    0, (0.035 * detectionClusters{1, 1}.Measurement(1, :))^2,
    0, 0;
    0, 0,
    (0.035 * detectionClusters{1, 1}.Measurement(1, :))^2, 0;
    0, 0, 0,
    (0.035 * detectionClusters{1, 1}.Measurement(1, :))^2];
end
if detectionClusters{1, 1}.Measurement(1, :) >= 0.5
    R = [(0.05*detectionClusters{1, 1}.Measurement(1, :))^2];
end
if detectionClusters{1, 1}.Measurement(1, :) >= 0.5
P0 = [(0.035 * detectionClusters{1, 1}.Measurement(1, :))^2,
        0, 0, 0;
    0, (0.035 * detectionClusters{1, 1}.Measurement(1, :))^2,
    0, 0;
    0 , 0,
    (0.035 * detectionClusters{1, 1}.Measurement(1, :))^2, 0;
    0 , 0, 0,
    (0.035 * detectionClusters{1, 1}.Measurement(1, :))^2];
end
    dt = toc(tStart);
    tStart = tic;
    A = [1, 0, dt, 0;
```

```
          0, 1, 0, dt;
          0, 0, 1, 0;
           0, 0, 0, 1];

%       B = [0;
%            0;
%            dt;
%            dt];
      % applying kalman filter after clustering:
      Pm = P0;
     [K] = vel_est_kalman_filt( A, C, Q, R, Pm);
     sep = sem + K*(mean_measurement - C*sem);
     pose = [sep(1:2)];
     velex = [sep(3:4)];
     % prediction step
     sem = A*sep; %+ B*vele; % pass this final value

     % measurements from laser sensor at each point of time
     %therefore,
     % it is in the form [x; y; vx; vy]
     % subscribe to odometery rostopic to get the current
     %velocity of the vehicle
     %there is a slight change in the velocity for
     %odometery model
     % might be because of covariance noise or distribution
     %between x
     % and y velocity of the odometery model.
```

```
% use function: "ACC_PI_test" for Proportional-Integral
%Control
% use function: "new_mpc_test" for Model Predictive
%Control
% use function: "ACC_P_test" for Proportional Control

    acc_vel = ACC_PI_test(velex(1), velex(2),
    mean_measurement(1),
    mean_measurement(2), msg_odom.Twist.Twist.Linear.X,
    msg_odom.Twist.Twist.Linear.Y);
  % maximum velocity handle by the turtlebot burger motor
    if (acc_vel < 0.2 && acc_vel >= 0)
      velmsg.Linear.X = acc_vel;
      send(robot,velmsg);
      robvel = 0;
    end

    v_kalman_save = [v_kalman_save; velex(1)];
    save('V_x_kalman_save_plot.mat', 'v_kalman_save');
    V_x_rel_save = [V_x_rel_save; robvel];
    %x_rel_save = [x_rel_save, mean_measurement];
    x_rel_save = [x_rel_save, mean_measurement];
    save('V_x_relsave_plot.mat', 'V_x_rel_save');
    save('x_relsave_plot.mat', 'x_rel_save');
    %disp(['## -- Robot Velocity :',num2str(robvel)])
    detection_meas =[];
```

```
end
```

2) The following is the function to convert bounding box width into relative distance.

```
function x_rel_dist = plot_pix_vs_dist(pixel_width) %#codegen

%dist_x = [0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1;
 %1.1; 1.2; 1.3; 1.4; 1.5; 1.6; 1.7; 1.8; 1.9; 2; 2.1; 2.2];


%width_pixels = [280; 181; 169; 140; 127; 100; 92;
 %84; 77; 72; 70; 64; 59; 54; 49; 48; 46; 45; 44; 42];


%plot (width_pixels, dist_x )
%ylabel('RELATIVE DISTANCE OF TURTLEBOT')
%xlabel('width of turtlebot in pixels')

%% According to the plot we can get the interpolating polynomial
% which can satisfy the value of the y= relative distance
% and x= width in pixels
%% Taking exponential function to fit the curve:
% x = [.18; .25; .335; .41; .50; .60; .70; .83;
% .90; 1.00; 1.10; 1.20; 1.30];
% x_rel_dist = 0.3809258 + 4.681089*e^(-0.02398154*pixel_width)
% use this or bottom one!!

 if ~isempty(pixel_width)
```

```
%x_rel_dist = 0.04417456 + (1983352 - 0.04417456)/(1 +
%(pixel_width/0.0001949206)^1.120355); older equation
%x_rel_dist = -0.009777765 + (16.48311 + 0.009777765)/(1 +
%(pixel_width/6.138883)^1.024073); new one


%x_rel_dist = 74.788*pixel_width^(-0.966); % 640X480 Image

 x_rel_dist = 56.815*pixel_width^(-1.006); % 410X308 Image

%end
%hold on
%plot(x, y)
%xlabel('relative distance')
%ylabel('width in pixels')
end
```

3) The following is the function to calculate y coordinate from image.

```
   % calculating the y coordinate from the image
% line([320, 320], [0, 480]) %vertical line on image
%for reference.
% pix_fromx : x pixel centroid point from image (0,0)
%i.e. x + (width/2)
function y_vis = y_coord_estimation_vision_410X308(pix_fromx,
        x_rel_dist_vis) %#codegen
   %bboxes2 = [475, 330, 130, 110]; % this is the reference
   %bounding box for calculation. for 640X480 image
```

```
    bboxes2 = [290, 210, 90, 75]; %for 410X308 image
    pixels = bboxes2(1)+bboxes2(3) - 205; % change 320 i.e.
    %1/2*(size of image in x direction) value according to
    %pixel size of the image
    f= (pixels*9)/4; % constant
    new_pixel_value = pix_fromx - 205; % change 320 i.e.
    %1/2*(size of image in x direction) value according
    %to pixel size of the image
    %pixel value change according to position.
    y_vis = (new_pixel_value * x_rel_dist_vis)/410;
end
```

4) The following is the function to convert LDS detection into x and y coordinates.

```
function x_y_coord = lds_convert(detect_rad) %#codegen
% This function converts the Lidar detections into [x, y]
% coordinates at different degrees of the detections.

% flipping the array values to make values in the form
%(60: 1 : 360 : 301) degree values. i.e. [60: 0: -59] degrees.
%theta = [flip(linspace(1, length(detect_rad(1:60)),
%length(detect_rad(1:60)))), -linspace(0,
%length(detect_rad(61:120))-1,
% length(detect_rad(61:120)))];

theta = [flip(linspace(1, length(detect_rad(1:15)),
length(detect_rad(1:15)))), -linspace(0,
length(detect_rad(16:30))-1,  length(detect_rad(16:30)))];
theta = deg2rad(theta);
```

```
x_y_coord = zeros(length(detect_rad), 2);
x= detect_rad.*cos(theta');
y= detect_rad.*sin(theta');
x_y_coord(:, 1) = x;
x_y_coord(:, 2) = y;
%the height of the Lidar placed on the turtlebot.
%x_y_coord(:, 3) = 0.18;
end
```

5) The following is the function to clustering the LDS detection.

```
function detectionClusters = cluster_lds_new(detections,
vehicleSize) %#codegen
N = numel(detections);
distances = zeros(N);
for i = 1:(N)
    for j = i+1:N
        if detections(1, i).SensorIndex ==
        detections(1, j).SensorIndex
        distances(i,j) = norm(detections(1, i).Measurement(1:2)
            - detections(1, j).Measurement(1:2));
        else
            distances(i,j) = inf;
        end
    end
end
leftToCheck = 1:N;
i = 0;
```

```
detectionClusters = cell(N,1);
% if array is empty!!
if (distances)==0
    detectionClusters{1} = detections(1,1);
    detectionClusters{1}.Measurement = [0; 0];
    detectionClusters(2:end) = [];
else
    while ~isempty(leftToCheck)
        % Remove the detections that are in the same cluster
        %as the one under consideration
        underConsideration = leftToCheck(1);
        clusterInds = ( distances(underConsideration,
        leftToCheck) < vehicleSize);
        detInds = leftToCheck(clusterInds);
        clusterDets = [detections(1, detInds)];
        for k = numel(clusterDets)

        clusterMeas = [clusterDets(1).Measurement];
        meas = mean(clusterMeas, 2);
        %disp(['##1 -- cluster meas :',(meas)])
        meas2D = [meas(1:2)];
        i = i + 1;
        detectionClusters{i} = detections(1, detInds(1));
        detectionClusters{i}.Measurement = meas2D;
        leftToCheck(clusterInds) = [];
    end
    detectionClusters(i+1:end) = [];
```

```
end
end
```

6) The following is the function for Kalman Filter computation.

```
% This code has referred from
% https://www.cs.cmu.edu/~cga/dynopt/kalman/kalman.html and
% https://statweb.stanford.edu/~candes/acm116/Handouts/Kalman.pdf
% for more info check it out.
% as we are having longitudinal autonomy, therefore taking simple
%state transition matrix!!

function [K] = vel_est_kalman_filt(A, C, Q, R, Pm) %#codegen
% designing the Kalman filter

for i = 1:1000
% measurement step
    S = C * Pm * C' + R;
    K =  Pm*C' * inv(S);

    Pp = Pm - K*C*Pm;
 % prediction step
    Pm = A*Pp*A' + Q;
end
end
```

7) The following is the function for Proportional Control.

```
function accel_new = ACC_P_test(rel_velx, rel_vely, rel_posx,
    rel_posy, vel_egox, vel_egoy) %#codegen
```

```
% Proportional Control
% accele_new: spit out the velocity of the turtlebot
% brake: stop while no movement of the lead vehicle
%vel_ego = accele_new =0.
% rel_dist : rel_posx distance between the ego and lead vehicle.
% vel_egox:  current speed of ego robot. this info will be
% taken from the imu of the vehicle/turtlebot.
% x_des: safe distance to be considered on the actual vehicle
% for braking is 0.4m in this case

% Main constraint is Dis_rel - D_safe >= 0. which is
%derived through
% ego velocity constraint i.e. 0< vel_egox < 0.2 m/s
% kp, ki for PI
% dist_safe: safe distance to be considered on the actual
%vehicle for braking but for
% this turtlebot application it is 0

minDist = 0.4; % Distance between the vehicles in meters.
%Kp = 0.26;
Kp = 0.5;
%Ki = 0.001;
%Kd = 1.0;
deltaT = 0.01;
err = intmax;
```

```
err = rel_posx − minDist;
Out = (Kp ∗ err); % only P controller
if Out >= 0
    accel_new = Out;
else
    accel_new = single(0.0);


% Display in command window
%disp(['##1 −− PI Error :',num2str(err)])
%disp(['##2 −− Acceleration Command : ',num2str(accel_new)])
%disp(['##3 −− Integral : ',num2str(inte)])
%disp(['##4 −− Distance : ',num2str(rel_posx)])
%disp(['##4 −− Out : ',num2str(Out)])
end
```

8) The following is the function for Proportional Integral Control.

```
function accel_new = ACC_PI_test(rel_velx, rel_vely, rel_posx,
  rel_posy, vel_egox, vel_egoy) %#codegen

%  Proportional Integral PI Controller
% accele_new: spit out the velocity of the turtlebot
% brake: stop while no movement of the lead vehicle
%vel_ego = accele_new =0.
% rel_dist : rel_posx distance between the ego and lead vehicle.
% vel_egox:  current speed of ego robot. this info will be
% taken from the imu of the vehicle/turtlebot.
% x_des: safe distance to be considered on the actual
```

```
%vehicle for braking
% is 0.4m in this case

% Main constraint is Dis_rel - D_safe >= 0. which
% is derived through
% ego velocity constraint i.e. 0< vel_egox < 0.2 m/s
% kp, ki for PI
% dist_safe: safe distance to be considered on the actual
% vehicle for braking but for
% this turtlebot application it is 0



persistent err_old
if isempty(err_old)
    err_old = 0;
end

minDist = 0.4;  % Distance between the vehicles in meters.
%Kp = 0.26;
%Ki = 0.001;
Kp = 0.5;
Ki = 0.01;

deltaT = 0.01;
err = intmax;
err = rel_posx - minDist;
```

```
int_err = err_old + ((err - err_old) * 0.4); % dt = 0.4
Out = (Kp * err) + (Ki * int_err); % PI controller
    % anti-windup error
if Out > 1
     accel_new = 0.2;
     Out1 = 1;
     Ki = 0;
elseif Out < -0.05
     accel_new =0.0;
     Out1 = 0;
     Ki = 0;
else
     accel_new = double(Out);

     Out1 = double(Out);
    %Ki = 0.001;
     Ki = 0.01;
end
err_old = double(err);
Out = Out1;

% Display in command window
%disp(['##1 -- PI Error :',num2str(err)])
%disp(['##2 -- Acceleration Command : ',num2str(accel_new)])
%disp(['##3 -- Integral : ',num2str(inte)])
%disp(['##4 -- Distance : ',num2str(rel_posx)])
%disp(['##4 -- Out : ',num2str(Out)])
```

```
end
```

9) The following is the function for Model Predictive Control.

```
function [accel_new] = new_mpc_test(rel_velx, rel_vely, rel_posx,
rel_posy, vel_egox, vel_egoy) %#codegen

%  MPC, solving QP problem: min (0.5*x'*H*x + f' * x) with
% linear inequality constraint A*x >= b
% speed: current speed
% accele_new: spit out the velocity of the turtlebot
% brake: stop while no movement of the lead vehicle
% vel_ego = accele_new =0.
% rel_dist : rel_posx distance between the ego and lead vehicle.
% vel_egox:  current speed of ego robot. this info will be
% taken from the imu of the vehicle/turtlebot.
% x_des: safe distance to be considered on the
%actual vehicle for braking
% is 0.4m in this case

% using reletive distance minimization.
% Cost function: minimize J = |x_rel - x_des|^2
% Main constraint is Dis_rel - D_safe >= 0. which
%is derived through
% ego velocity constraint i.e. 0< vel_egox < 0.2 m/s

delta_t = 0.4;
x_des = 0.4; % minimum distance is 40 cm.
v_lead = rel_velx + vel_egox;
```

```
H = [2,  0;
     0,  0.0001];
f= [-2*x_des;  0];
A = [0,  1;
     0,  -1];
b = [0;  -0.2];
iA0 = false(size(b));
Aeq = [1,  delta_t];
beq = double(rel_posx + v_lead*delta_t);

[L,p] = chol(H,'lower');
Linv = inv(L);
%Linv = reshape(Linv,1,1);

opt = mpcqpsolverOptions;

%cold start the solver, inequality constraints
[x,stat,iA] = mpcqpsolver(Linv,f,A,b,Aeq,beq,iA0,opt);
if rel_velx < 0
    accel_new =0;
end
if stat > 0
    accel_new =  x(2);

else
    accel_new = vel_egox;
end
```

```
% disp(['##1 -- Status : ',num2str(stat)])
% disp(['##2 -- relative position : ',num2str(rel_posx)])
% disp(['##2 -- ego velocity : ',num2str(vel_egox)])
% disp(['##2 -- relative velocity : ',num2str(rel_velx)])
% disp(['##2 -- Acceleration Command : ',num2str(accel_new)])
end
```

- Running Simulation models:

Procedure for using the Simulation models.

1) Run the `read_rosbag_file.m` file in the matlab environment. This file reads the rosbag file which is `const_vel_track_record.bag` for constant lead robot velocity or `step_vel_track_record.bag` for variable lead robot velocity. These files are logged data of the Lead vehicle velocity. This data will be used in the simulink process to check the behaviour of the system.

2) After this run these commands in MATLAB terminal: `len = length(lead_bot_vel_vari);` for variable lead robot velocity and then `timebpt = 0:0.4:((len-1)*0.4);`.

3) Also, make sure the simulation time should be less than the length of the `timebpt`.

- Simulation models:

1) The following is the code for reading the rosbag file in matlab.

```
%clear all, close all, clc

%% constant velocity = 0.07 m/s is recorded on track through odom
%%rostopic actual time of record is 99.1749 and time step
%is T = 0.0333
```

```
% as the experiment has time step T = 0.4, therefore saving
% the data after about each 12 data record.

lead_vel_vari = rosbag('const_vel_track_record.bag');
lead_vel_vari_select = select(lead_vel_vari,'Topic','/odom');
msgStructs = readMessages(lead_vel_vari_select,'DataFormat','struct');
lead_bot_vel_vari_const =[];
for i = 1:12:length(msgStructs)
    lead_bot_vel_vari_const = [lead_bot_vel_vari_const,
    msgStructs{i, 1}.Twist.Twist.Linear.X];
end

%% constant velocity = 0.07 m/s is recorded on track through odom
%% rostopic actual time of record is 56.8305 and time step is
%T =0.0333
% as the experiment has time step T = 0.4, therefore saving
% the data of after about each 12 data record.

lead_vel_vari = rosbag('step_vel_track_record.bag');
lead_vel_vari_select = select(lead_vel_vari,'Topic','/odom');
msgStructs = readMessages(lead_vel_vari_select,'DataFormat',
'struct');
lead_bot_vel_vari =[];
for i = 1:12:length(msgStructs)
    lead_bot_vel_vari = [lead_bot_vel_vari,
    msgStructs{i, 1}.Twist.Twist.Linear.X];
end
```

2) The following is the function for 1-D Kalman Filter.

```
function [sep, sem] = new_kalman_sim_test(sem, sensor_data,
                                A, C, Q, R, P0)
        if sensor_data(1) >= 0.5
          Q = [(0.035 * sensor_data(1))^2, 0;
                    0,  (0.035 * sensor_data(1))^2];
        end
        if sensor_data(1) >= 0.5
                R = [(0.05*sensor_data(1))^2];
        end
        if sensor_data(1) >= 0.5
                P0 = [(0.035 * sensor_data(1))^2, 0;
                          0 , (0.035 * sensor_data(1))^2];
        end
  Pm = P0;
  [K] = sim_vel_est_kalman_filt_1D( A, C, Q, R, Pm);
  sep = sem + K*(sensor_data(1) - C*sem);
  pose = [sep(1)];
  velex = [sep(2)];
        % prediction step
  sem = A*sep; %+ B*vele; % pass this final value
```

3) The Figure A.1 is the simulink process for Proportional Control.

4) The Figure A.2 is the simulink process for Proportional-Integral Control.

5) The Figure A.3 is the simulink process for Model Predictive Control.

- This project is open-source, and can be found here .

`https://github.com/AnsSUN/Turtlebot_project_maintain_distance`

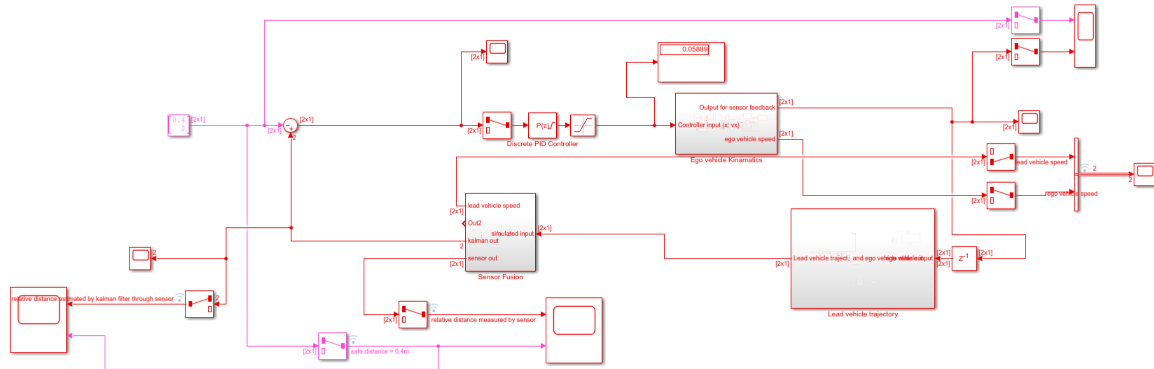

Figure A.1: Simulink Process for Proportional Control

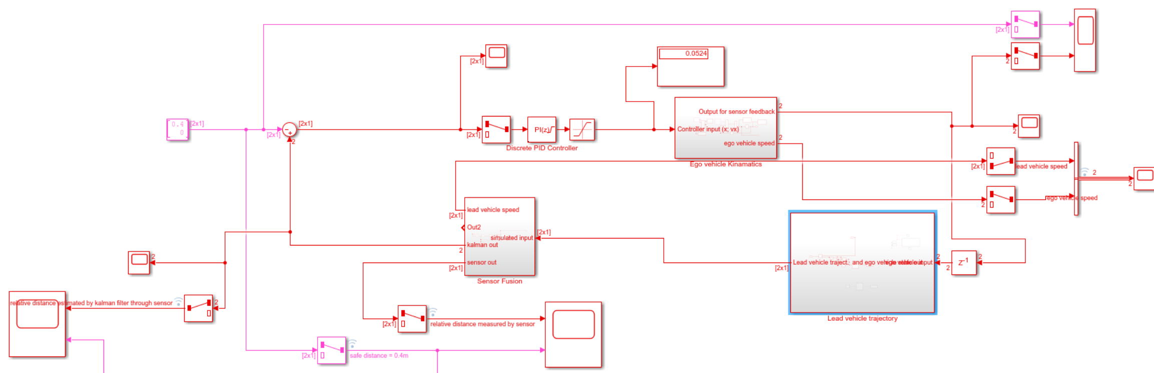

Figure A.2: Simulink Process for Proportional-Integral Control

- Demo project video

  A demonstration video can be obtained here.

  `https://youtu.be/KBwvVyJJjb0`

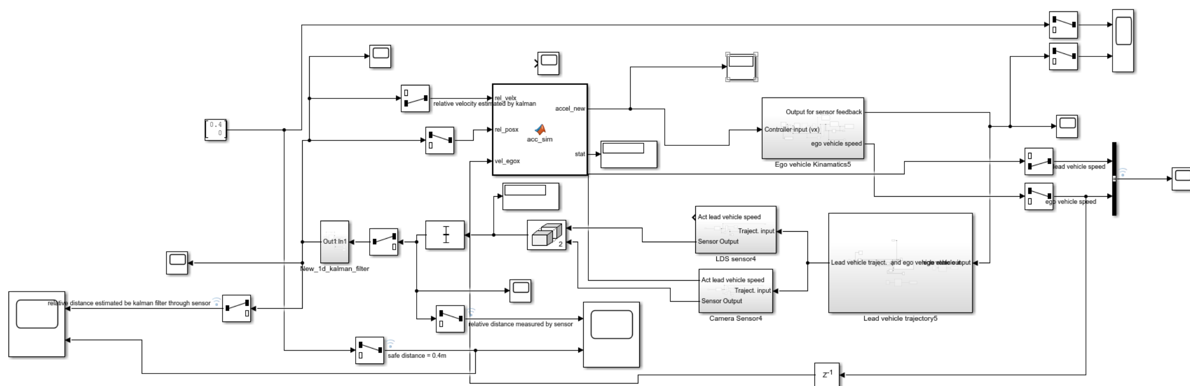

Figure A.3: Simulink Process for Model Predictive Control

VITA

Anshul Sungra is a candidate for MS in Mechanical Engineering at the University of Washington, Seattle. Before joining the University of Washington, he worked at Airbus on the behalf of L&T Technology Services as a Mechanical Design Engineer. He completed his Bachelor of Technology in Mechanical Engineering from National Institute of Technology, Bhopal.

Please look up his page to contact him: `https://anssun.github.io/`.