TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

B-03-BME-2017/2022

Optimal Trajectory Generation for Autonomous Navigation of Wheeled Robots

by:

Ankit Kharel (074BME604)

Anusha Acharya (074BME607)

Nitesh Subedi (074BME624)

Prajwal Koirala (074BME627)

A PROJECT REPORT

SUBMITTED TO THE DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF BACHELOR IN MECHANICAL ENGINEERING

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING

LALITPUR, NEPAL

FEBRUARY, 2022

COPYRIGHT

TRIBHUWAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING

The undersigned certify that they have read, and recommended to the Institute of Engineering for acceptance, a project report entitled " Optimal Trajectory Generation for Autonomous Navigation of Wheeled Robots" submitted by Ankit Kharel, Anusha Acharya, Nitesh Subedi, and Prajwal Koirala in partial fulfillment of the requirements for the degree of Bachelor of Mechanical Engineering.

Supervisor, Dr. Mahesh Chandra Lunitel

Professor

Department of Mechanical and Aerospace Engineering

Supervisor, Dr. Sanjeev Maharjan

Assistant Professor

Department of Mechanical and Aerospace Engineering

External Examiner, Dr. Krishna Prasad Shrestha

Assistant Professor

Kathmandu University

Committee Chairperson, Surya Parsad Adhikari, PhD

Associate Professor and Head

Department of Mechanical and Aerospace Engineering

2022/03/06

Date

ABSTRACT

Wheeled robots are propelled using motorized wheels for navigation. The trajectory generation problem refers to the determination of a set of control inputs to achieve the desired motion of the robot. The project aims to develop a feedback system that plans the trajectory of the wheeled robot taking account of the current state and environment. The trajectory generator needs to lead the robot toward the goal location while avoiding the obstacles and fulfilling other dynamic and static constraints. To design the trajectory generator, an optimal control problem is formulated for the wheeled robot. The optimal control problem is transcribed into a non-linear programming problem and solved in terms of state-control pair at each time step. This method of controlling the robot for a finite-length time horizon using the real-time information updates is also called Model Predictive Control (MPC). To accomplish the project objectives, the proposed trajectory generation system is first developed in the numeric computing environment MATLAB. It is then reimplemented in Python to test against Turtlebot3 Waffle in Gazebo environment using Robot Operating System (ROS). Finally, it is used in an actual robot with the necessary hardware and software. Results obtained in the simulated environment for a number of cases are validated using the physical implementation of the robot.

Keywords: *Robots, Navigation, Trajectory Generation, Optimal Control, Non-linear Programming, Model Predictive Control, Simulation, ROS*

ACKNOWLEDGEMENT

First of all, we would like to express our deepest appreciation to the Department of Mechanical and Aerospace Engineering for providing us with the opportunity to undertake this project. We are highly indebted to the unwavering supervision and guidance of our supervisors Prof. Dr. Mahesh Chandra Luintel and Asst. Prof. Dr. Sanjeev Maharjan.

Fabrication of this project would not have been possible without the cooperation and financial support from NSDevil. We express our sincere gratitude to the senior manager Mr. Geoffrey Tegny and manager Er. Abhinab Acharya of the NSD-Robo project for facilitating this completion of this project.

We are extremely grateful to the Robotics Club, Pulchowk Campus for providing some necessary parts and equipments to fabricate this project. We acknowledge the assistance of Mr. Sabin Shrestha and all other members of the club for sharing the knowledge about electronic components and helping to design the circuit board.

We would like to acknowledge our seniors Er. Nirdesh Bhattarai and Er. Prashant Bhatta for their constant guidance and assistance in problem solving, experience sharing and encouragement during our hard times.

We wish to show our gratitude to Dr. Min Adhikari for the headstart discussion on solving the optimal trajectory problems and other invaluable suggestions. The crucial support and guidance of Er. Rishav Mani Sharma (Orion Space) helped to take this project to next level.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CG	Center of Gravity
CzIDAS	Czech In-depth Accident Study
DOF	Degree of Freedom
EKF	Extended Kalman Filter
GPIO	General Purpose Input/Output
IMU	Inertial Measurement Unit
IPOPT	Interior Point Optimizer
MPC	Model Predictive Control
NLP	Non Linear Programming Problem
PCB	Prediction and Cost function Based approach
PPR	Pulse per revolution
ROS	Robot operating system
SLAM	Simultaneous Localization and Mapping
UAV	Unmanned Aerial Vehicle
UKF	Unscented Kalman Filter
URDF	Unified Robot Description Format

LIST OF SYMBOLS

α	Angle of slip
ω	Angular velocity
μ	Coefficient of friction
κ	Slip ratio
v	Linear velocity
V_r	Linear velocity of right wheel
V_l	Linear velocity of left wheel
F_x	Longitudinal force on tire
F_y	Lateral force on tire
F_z	Vertical load on tire
R	Instantaneous center of rotation
r	Radius of wheel
L	Wheel base
I	Mass moment of inertia
H_k	Measurement matrix
Q_k	Co-variance matrix of process noise
R_k	Co-variance matrix of measurement noise
w_{k-1}	Process noise
v_k	Measurement noise
z_k	Sensor measurement
y_k	Measurement residual
S_k	Co-variance matrix of measurement residual
K_k	Kalman gain
P	State Co-variance matrix

CHAPTER ONE : INTRODUCTION

1.1 Background

Robotics is one of the rapidly developing fields in today's world. For the past few decades, robots have proven to be good human friends by replacing the repetitive and tedious tasks. With regard to the locomotive mechanism, mobile robots can be broadly classified depending upon whether they are based on land or air or water. Wheeled mobile robots navigate on the ground. They use wheels for locomotion which are driven by the motor. The various types of wheeled robots are single-wheeled robots, two-wheeled robots, three-wheeled robots, four-wheeled robots, and multi-wheeled robots.

Wheeled robots are widely used in smart factories and warehouses for safe material handling. They also find their applications in surveillance, transportation, medical care, reconnaissance and planetary exploration, where a robot has to navigate around the dynamic and chaotic environment, maneuvering through various types of obstacles. But, as technology advances, future generations of robots require autonomous handling of complex problems circumscribed by limited time and resources.

Autonomous Navigation is the ability for a vehicle to determine its location within the environment and figure out a path that will take it from its current location to some goal without human intervention. This ability finds its application in ground vehicles like self-driving cars, Unmanned Aerial Vehicle (UAV), spacecraft, submersible, and other mobile robots. Broadly, there are two different approaches to navigate. A heuristic approach utilizes practical rules or behaviors to produce a feasible solution that may not be optimal but is good enough to achieve some immediate goal. On the other hand, the optimal approach generates optimal solutions through optimization of some objective function subject to various constraints.

Autonomous driving is getting closer, but it's been a long difficult journey to arrive at this point. The dream of self-driving cars started way back along with the start of automobile. Even though there is a lot of work to do, until we see self-driving cars on our

streets, companies like Waymo, GM's Cruise and Tesla made impressive progress over the last years (Cusumano, 2020). Even their autonomous system is not fully featured. Still there are limitations but with the technical prerequisites becoming available with respect to previous scenarios, the vision of highly autonomous vehicles gets closer to realization. Considerable effort is being directed towards autonomously navigating vehicles in different urban environments. One of the most dangerous works we do in our daily life is driving. There are hundreds of vehicles running and most of them are driven by human, in the present context. As evidenced by the results of Czech In-depth Accident Study (CzIDAS) from the amount of analyzed traffic accidents, in 40% of analyzed accidents inattention has been contributing to the accident occurrence (Bucsuházy et al., 2020). This is one of the human errors, still there are many. Through automation, we can minimize or completely eliminate driving deaths by taking human error out of the picture.

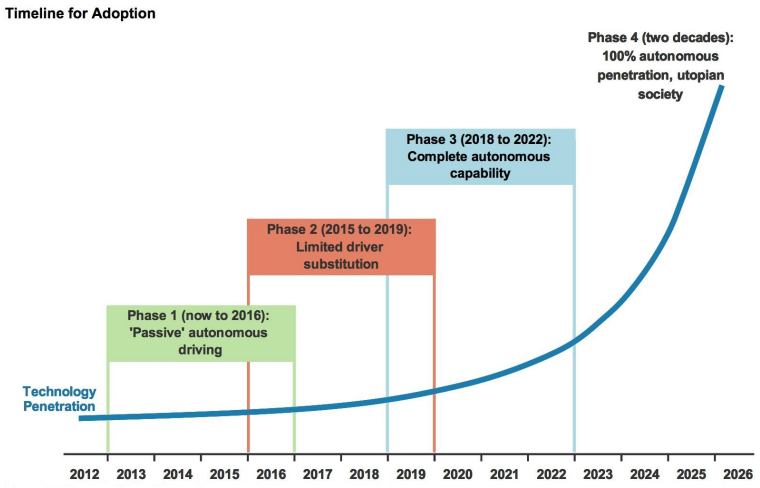


Figure 1.1: Autonomous Vehicle Adoption Timeline

(Source: Company data, Morgan Stanley research.)

1.1.1 Problem Statement

The question of how to generate an optimal trajectory for an autonomous wheeled robotic vehicle to follow is one of the challenges in this area. Path planning in general is a difficult task, especially when considering vehicle dynamics and moving obstacle referred as dynamic environment. Rapidly changing environments require a re-computation of the

path in real-time (Purwin, 2008). In the dynamic environment with autonomous vehicles, we have to ensure that each and every vehicle is performing accurately. Otherwise, a small error in one of the vehicles may lead to a disaster.

Generally, the main issues that are to be faced while developing a autonomous vehicle exists at three different levels-perception, planning and control as shown in figure 1.2 (You et al., 2019)

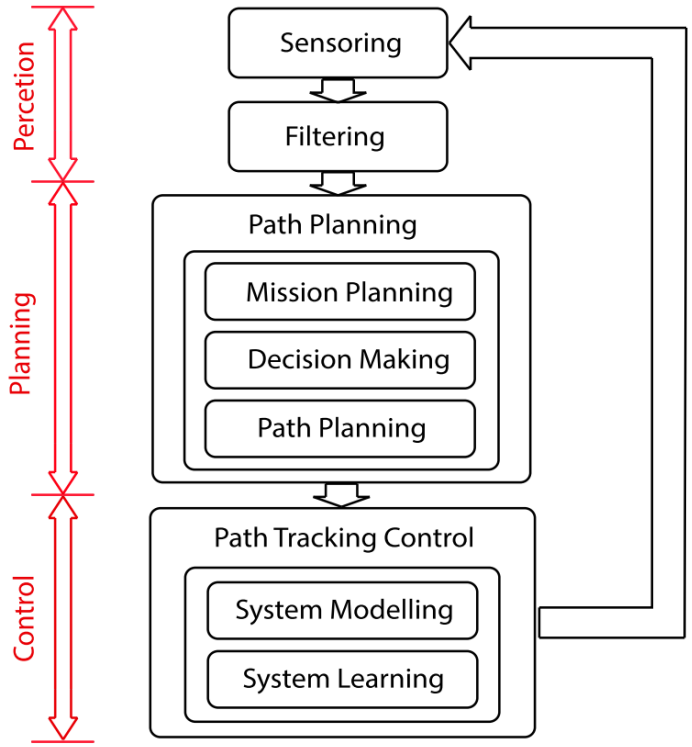


Figure 1.2: Autonomous control architecture at different levels

(You et al., 2019).

The perception level consists of sensing and filtering the data which may be the surrounding data with one's current location. This system consists of number of sensors which provides data and filter denoises it giving the reasonable estimate for the unmeasureable states (Wan and Van Der Merwe, 2000). The planning level consists of mainly three tasks, which include mission planning, where the vehicle solves a routing problem in order to complete a task, decision making, where the vehicle chooses an appropriate action for the next time step from an available action set, and path planning,

where the vehicle plans its future trajectory as a function of space or time (Shalev-Shwartz et al., 2016). Finally, the control level receives the signals from the planning level, maintains the stability of the vehicle, and tracks the desired path. All of these levels should operate simultaneously in a control system of a vehicle to operate autonomously. In our robot, we plan to perform all of the three levels. So we will take data from the sensor, filter it, solve our optimization problem to plan the trajectory with the required control actuation and finally implement the actuation to track the desired path.

1.2 Objectives

1.2.1 Main Objective

- To design and implement an optimal control system in a wheeled robot for autonomous navigation in an uncertain environment.

1.2.2 Specific Objectives

1. To develop kinetic and dynamic models of wheeled mobile robots
2. To formulate the optimal trajectory generation problem as an optimization problem with appropriate cost function and constraints
3. To compute numerical solution to the optimization problem and derive closed loop solution with real-time information feedback
4. To develop a prototype of a differential drive robot and compare its performance with the simulation result

CHAPTER TWO : LITERATURE REVIEW

2.1 Vehicle Modeling

2.1.1 Unicycle Model

It is among the simplest models of wheeled robot with a single upright rolling wheel (Lynch and Park, 2017).

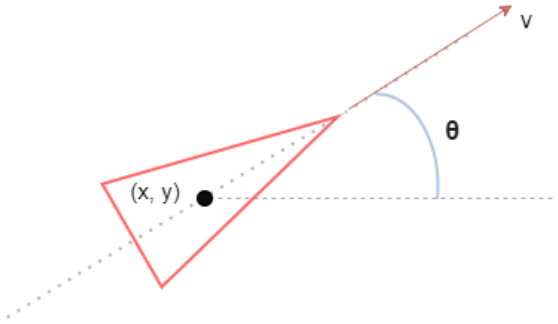


Figure 2.1: Unicycle Vehicle Model

If x and y represent position of the wheel in Cartesian coordinate's and θ represents heading direction of the robot, then

$$\dot{x} = v.cos(\theta) \tag{2.1}$$

$$\dot{y} = v.sin(\theta) \tag{2.2}$$

$$\dot{\theta} = \omega \tag{2.3}$$

2.1.2 Differential Drive Robot

A differential drive robot consists of two driven wheels whose speeds can be independently controlled. It is the most common type of model used to control mobile robots.

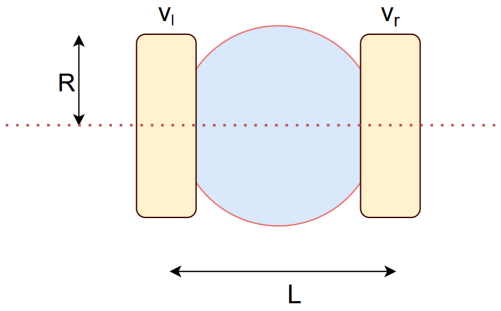
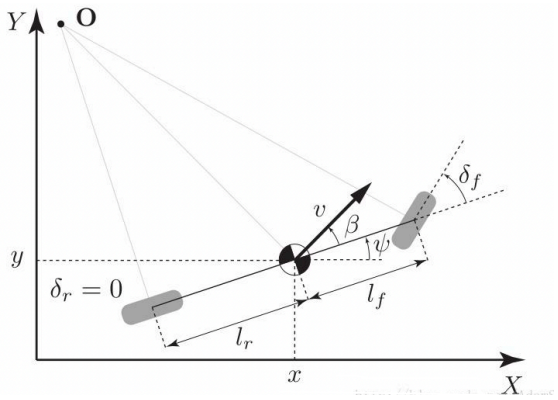


Figure 2.2: Differential Drive Robot

2.1.3 Kinematic Bicycle Model

In kinematic bicycle model, the two front wheels and also the two rear wheels are lumped together to a unique wheel located at the centre of the front and rear axle as shown in figure 2.4. It is assumed that only the front wheel of the vehicle is steered. The control inputs are acceleration at the rear wheel and front wheel steering angle (Polack et al., 2017).



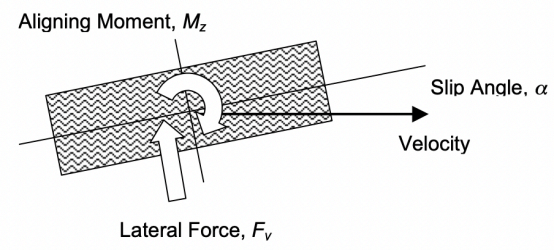
Source: (Polack et al., 2017)

Figure 2.3: Kinematic Bicycle Model

2.1.4 Dynamic Bicycle Model

Lateral Tire Model

In the absence of side force, a tire travels along a straight line. However, during turning maneuver the tire contact patch slips laterally which causes the direction of motion to be misaligned with the tire orientation. The angle between direction of motion and tire orientation is angle of slip (α).

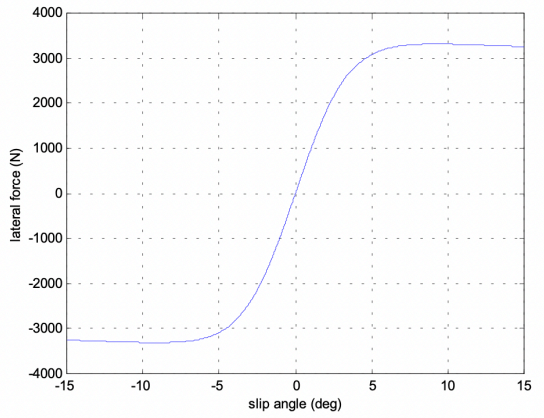


Source: (Polack et al., 2017)

Figure 2.4: Tire operating at a slip angle

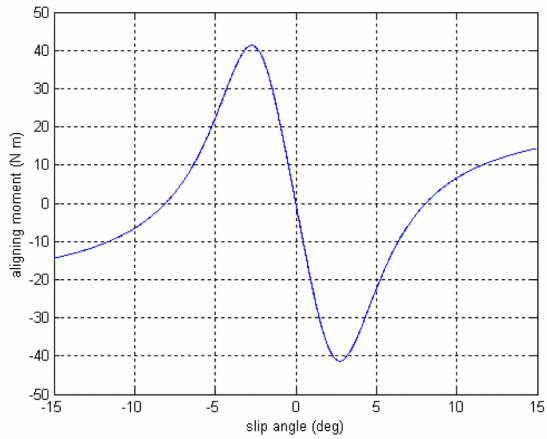
H.Pacejka of Delft University of Technology experimentally observed the lateral forces (figure 2.5) and aligning torque (figure 2.6) exerted on the tire with respect to slip angle and developed an empirical tire model called Pacejka tire model. The formula is called ”Magic Formula” and uses tire slip angle to calculate lateral force and aligning torque. The model parameters are dependent upon vertical tire load F_z (H.Pacejka, 2012).

H.Pacejka found that the lateral force on the tire varies largely linearly for small slip angle. So, for small slip angles the forces and moments on tire was approximated as linear function of slip angle.



Source: (G. Gim, 1990; G. Gim, 1991)

Figure 2.5: Lateral force VS slip angle



Source: (G. Gim, 1990; G. Gim, 1991)

Figure 2.6: Aligning torque VS Slip angle

UA-Gim tire model based on research by Dr.Dwangun Gim uses tire’s kinematic states

to calculate the forces and moments at the contact point. This makes the formulation of forces simplified by providing general equation for all road-tire interaction condition (G. Gim, 1990; G. Gim, 1991).

2.2 State estimation and Localization

State estimation problem deals with determining the best value of some unknown physical quantity from the set of measurements. It started way back when Gauss (1963) developed the method of least square to accurately estimate planetoid Ceres's orbital parameters from Piazzi's published measurements. One of the most important types of state estimation for wheeled robots is localization, which is the process of determining the position and orientation of the robot (Jaroszek and Trojnacki, 2015). Probabilistic methods (Cassandra et al., 1996) such as Kalman filters (Kalman, 1960), Marvok localization (Burgard et al., 1998) and particle filters (Dellaert et al., 1999) are widely applied for the localization of robot.

Meriem et al. (2016) did a study on the weighted least square method for state estimation and investigated how the efficiency of this method changes according to measurement number, type, weight, and noise level. Considering weight or the co-variance of measurement noise produced better performance.

Kalman filter analyzes the performance of state co-variance and provides an optimal estimation for linear systems. Nasir et al. (2017) were able to predict position and heading of two wheeled drive vehicle by applying kalman filter to sensor data obtained from Inertial Measurement Unit (IMU), odometers and ultrasonic sensors. Their error converged to 10mm, 10mm and 0.06 deg in x-position, y-position and heading.

To obtain better results in nonlinear system, extended version of kalman filter is used. Tong (2012) applied the Extended Kalman Filter (EKF) approach to estimate the lateral and longitudinal velocity of a three Degree of Freedom (DOF) full-vehicle model formulated in MATLAB /SIMULINK. Results of actual vehicle parameters obtained from CarSim software, well-matched estimated value.

To overcome disadvantages of EKF such as linearization errors and higher computa-

tional cost to calculate Jacobian matrices, Antonov et al. (2011) proposed utilization of Unscented Kalman Filter (UKF) which directly approximate probability density function. They have tested the performance of UKF on the nonlinear planer vehicle model in BMW 5 series, which showed a good compromise between accuracy and computational cost.

2.2.1 Odometry

The use of motion sensors to determine the robot's change in position relative to a known position is known as odometry. Shaft encoders are frequently mounted to the driving wheels of robots to count the number of pulses with wheel rotation. The distance travelled by the robot is then estimated by counting these pulses (*Robo-Rats Locomotion: Odometry*, 2001). However, problem due to drift is unaccounted by the estimate.

Wheel Encoder Odometry

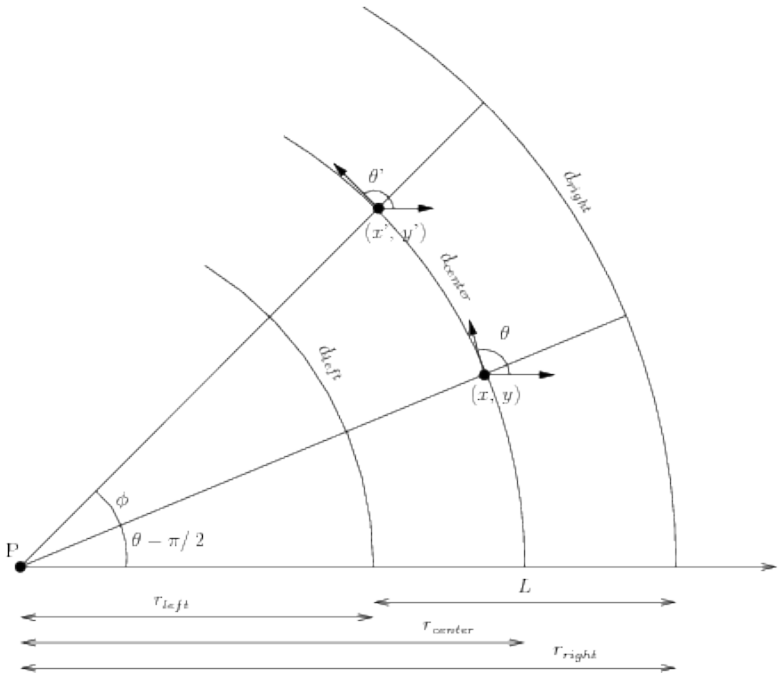


Figure 2.7: Wheel Encoder Odometry Calculations

$$D_c = d_{center} = \frac{d_{left} + d_{right}}{2} \tag{2.4}$$

$$x' = x + D_c\, cos(\theta) \tag{2.5}$$

$$y' = y + D_c\, sin(\theta) \tag{2.6}$$

$$\theta' = \theta + \frac{d_{left} - d_{right}}{L} \tag{2.7}$$

2.3 Trajectory Generation

Trajectory is represented as a sequence of states traversed by the vehicle, parameterised by time and, possibly, velocity. Trajectory planning (also known as trajectory generation) is concerned with the real-time planning of the actual vehicle's transition from one feasible state to the next, satisfying the vehicle's kinematic limits based on vehicle dynamics and constrained by the navigation comfort,while avoiding, at the same time, obstacles including other road users. Katrakazas et al. (2015) generated a number of trajectories during each planning cycle from the vehicle's current location, with a look-ahead distance, depending on the speed and line-of-sight of the vehicle's onboard sensors, and evaluating each trajectory with respect to some cost function to select the optimal one.

The state lattice (Pivtoraiko et al., 2009) is a method for inducing a discrete search graph on a continuous state space while respecting differential constraints on motion. The lattice planner formulation was not readily applicable to on-road driving scenarios due to the high density of vertices and edges that would have been necessary to represent paths conforming to lanes and this would also drastically increase the computational time (Katrakazas et al., 2015). Hardy and Campbell (2013) modelled the vehicles as rectangles; their trajectories were clustured for an easier identification. In this work, planning is seen as a non-linear constrained optimization problem (Katrakazas et al., 2015) . The optimization function includes both dynamic and static obstacles, distance to goal. The similar approach was found to be used by Ziegler et al. where hierarchical concurrent state machines are used with respect to static and dynamic obstacles, as well as yield and merge rules (Ziegler et al., 2014). The main limitation of this technique, however, is that other vehicles are presumed not to accelerate and to keep safe distances from the road boundaries.

Wei et al. in 2014, adopted a Prediction and Cost function Based approach (PCB). In this approach, a reference trajectory is used with static and dynamic obstacles as input

and multiple candidate trajectories are generated. After predicting the evolution of the traffic environment, the best strategy is chosen according to comfort, fuel consumption and progress towards goal (Wei et al., 2014). This method was compared with spatio-temporal lattice planner and 90% reduction of computational cost was found.

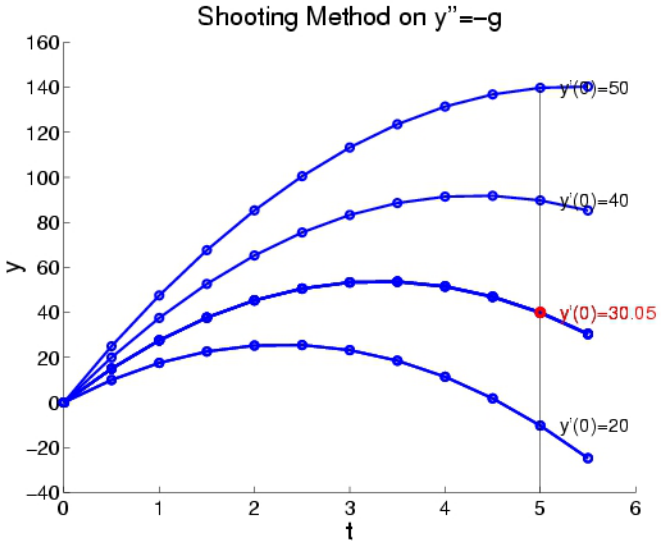
In terms of handling obstacles, existing approaches primarily rely on predicting the trajectories of other traffic participants, either by taking their trajectories into account, or by making assumptions of constant velocities or constant accelerations (Kushleyev and Likhachev, 2009). This leads to a huge computational power requirement, since the obstacles' trajectories need to be calculated and checked at each moment.

A different approach for trajectory planning, is Model Predictive Control (MPC), which combines aspects of control engineering within the planning module. Within MPC, a dynamic model for the vehicle is used and, through it, inputs from the controller are sampled about the future evolution of the vehicle's motion. From the dynamic model and the controller inputs, the optimisation problem of finding the best trajectory for the vehicle is solved. MPC was used within a driving corridor by Madås et al. (2013) using a linear bicycle model with linear tyre characteristics which also considers lateral and yaw dynamics.

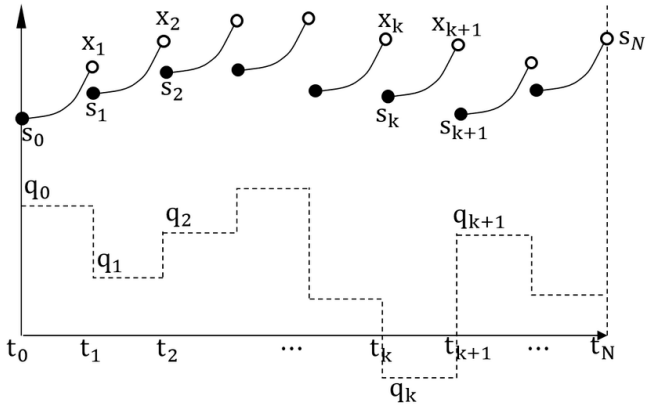
Model predictive control (MPC) scheme based on tailored non-quadratic stage cost is used to fulfill this control task. The weights in each terms of the functions are constant numbers and do not change with time or position of the vehicle. Since the system is finite time controllable, it is stabilized by a receding horizon control scheme with purely quadratic stage costs if an infinite optimization horizon is employed. However, due to the so called short-slightness of model predictive control, these stability properties are not preserved if the control problem is only optimized on a truncated and, thus, finite prediction horizon — even if an arbitrarily large terminal weight is added. Hence, it is necessary to either incorporate structurally different terminal costs or use non-quadratic stage costs to appropriately penalize the deviation from the desired set point (Müller and Worthmann, 2017; Arora, 2004). Worthmann et al. (2015) conducted numerical

solutions to explain the necessity of having non-quadratic running costs.

Shooting methods are widely used in the optimal control problems (Worthmann et al., 2015; Subchan, 2011). The reason behind this is the efficiency, robustess and convergence of the method in solving the boundary value problem. In addition to that, breaking the simulation running using shooting method into N number of sections and running the simulation individually makes he shooting method more powerful than other methods (Kelly, 2017). This method is called direct multiple shooting. Koch and Weinmüller



(a) Direct Single Shooting Method



(b) Direct Multiple Shooting Method

Source: (Koch and Weinmüller, 2003)

Figure 2.8: Shooting Methods

(2003) conducted the research on the convergence of the direct single shooting and direct multiple shooting for the perturbed Newton iteration. Multiple shooting was found to be much faster than single shooting with additional memory usage. Also, the number

of grid points used in multiple shooting makes the difference in the computational time. The higher or lower the grid points from a certain optimum value, the computational time increases (Koch and Weinmüller, 2003). The right value varies with the properties of the system. Adhikari and Ruiter (2020) have used direct multiple shooting method for the optimal trajectory generation problem. Subchan (2011) have used this method for Missile Trajectory Optimization with The Terminal Bunt Manoeuvre.

Andersson (2013) presented an open-source software framework for numerical optimization called CasADi. CasADi is a general-purpose tool that can be used to model and solve optimization problems with a large degree of flexibility (Andersson, 2013). It works on the symbolic frameworks which makes the user to learn and use CasADi efficiently. Worthmann et al. (2015) used CasADi in Model predictive control of non-holonomic mobile robots. Arbo, Grøtli, and Gravdahl (2019) used CasADi in closed-Loop Inverse Kinematics controller. It provides a framework for converting the optimal control problem into Non Linear Programming and solve the non linear problem to get the optimal solution. Moreover, CasADi provides a stack called Opti-stack which is a collection of CasADi helper classes that provides a close correspondence between mathematical NLP notation. This further helps in solving the problem efficiently.

2.3.1 Nonlinear Optimization Software

For simplicity, let us represent a nonlinear optimization problem as Equation 2.8.

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & c(x) = 0 \\ & x \geq 0 \end{aligned} \tag{2.8}$$

In general Equation 2.8 cannot be solved directly or explicitly so, an iterative method is used that takes an initial guess, and solves sequence of approximate sub problems to give sequence of approximate solution and refine the local model (2.8) until the improved solution is optimal (Leyffer and Mahajan, 2010). Depending upon how local model is constructed, Leyffer and Mahajan (2010) have distinguished it into three broad categories:

1. Sequential Linear Models
2. Sequential Quadratic Models
3. Interior-point Models

The solution of nonlinear optimization problem has four main components; a local model which approximates the optimization problem, a global convergence strategy, a global convergence mechanism, and convergence test. Global convergence strategy is essential to promote convergence from remote starting point. That means, it checks whether our current estimated solution is better than the previous estimated solution. Various globalization strategy like Augmented Lagrangian Method, Filter and Funnel Method, and globalization mechanism like line-search methods and trust-region methods are elaborated in detail by Leyffer and Mahajan (2010).

Depending largely on the local model, some Interior-point solvers and Sequential Linear/Quadratic Solvers are:

Interior-Point Solvers

Some of the commonly used interior-point solvers are; CVXOPT, IPOPT, KNITRO, and LOQO. CVXOPT is a free software package that uses an interior-point barrier method to solve convex optimization problem. It is highly efficient for small and medium scales problems but cannot be generalized to solve non-convex problems (Diamond and Boyd, 2016). KNITRO is a most advance commercial solver for nonlinear optimization that implements a trust-region interior-point penalty-barrier method and LOQO another commerical solver, implements infeasible primal-dual interior point method (Leyffer and Mahajan, 2016). IPOPT a open source software implements interior point line search filter method. Its mathematics in detail is described by Andreas Wachter (2005). Because of it's ability to solve large scale problems with upto millions variables and constraints it is widely used to solve optimal control problems. Assuming the Jacobian matrix of constraint function as sparse it can efficiently solve large problems and is not limited for small scale and dense problems (Andreas Wachter, 2005).

Sequential Linear/Quadratic Solvers

Some of the available sequential linear and quadratic solvers are CONOPT, FilterSQP, LINDO, NLPQLP, NPSOL, and SNOPT (Leyffer and Mahajan, 2010). CONOPT and SNOPT are solvers for large-scale nonlinear optimization however they are best suited with problems having moderate degree of freedom (Gill et al., 2005).

2.4 Simulation and Visualization

For simulating and understanding the performance of the robot in different situations Robot operating system (ROS), a free and open-source platform specially designed for robot programming can be used. It is not a real operating system but has operating system-like features (Joseph, 2018). The framework also provides different capabilities of robot programming like supporting high-level programming languages, off-the-shelf algorithms, integration with third-party libraries, interprocess communication, and so on. It has many built-in GUI tools and command lines that help debugging and visualization which are extremely useful while working with robots. For instance, visualization and simulation with cameras, inertial measurement units can be obtained by using the Rviz tool and Gazebo simulator (Joseph, 2018).

Quigley et al. (2009) presents a brief overview of ROS framework. ROS makes use of the Unified Robot Description Format (URDF) to model the robot. URDF is basically the collection of files (written in XML format) that has all the information of the robot's physical description such as its 3D model, actuators, controllers, sensors, etc. (Joseph, 2018). It also includes information about physical shape and size of the robot, description of the parts and their dynamic properties like mass, moment of inertia, collision and interaction with external environment etc. It gives knowledge of what a robot actually looks like to the computer without being the need to develop a real one.

To perform a specific action like finding a path by the robot, ROS services are used. ROS services make use of the ROS server: the node that is going to provide the service, and ROS clients: the node that consumes the service. The service is essentially one-time communication where the clients send the request and the server responds. A service can essentially be called anytime. Unlike Services, with ROS Topics, the publisher continuously sends information to the subscriber.

For simulation of robot in simulated environment gazebo simulator can be used. Gazebo simulates multiple robots in a 3D environment, with extensive dynamic interaction between objects. Using gazebo simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It also provides necessary plugins for controlling robots using different control mechanisms. It can also be used to simulate the sensing devices such as LIDAR, camera etc.

2.4.1 Hector SLAM

Simultaneous Localization and Mapping (SLAM) is a robotics problem that attempts to locate itself while also mapping the surroundings. The SLAM challenge arises when a robot lacks a global positioning sensor and must instead rely on ego-motion sensors like odometry, IMU, etc (Thrun, 2007). Hector SLAM is an open-source method that uses a LIDAR to create a 2D grid map of the surrounding area (Kohlbrecher et al., 2011). This system uses scan matching to determine the robot's location rather than the odometry of the wheel, which is a frequent way in other SLAM methods. Due to its rapid update rate, the LIDAR is able to complete the scan matching duty to find the robot quickly and precisely. The Hector algorithm employs the Gaussian-Newton minimization approach, which is considered an update to the Newton method and has the benefit of not requiring the computation of second derivatives (Eliwa et al., 2017).

The system subscribes to the $sensor\_msgs/LaserScan$ message and publishes $nav\_msgs/OccupancyGrid$, tf transformation, and $geometry\_msgs$ $/PoseWithCovarianceStamped$. Based on Hector SLAM data, the information may be used for sensor fusion in the system. The technology creates a detailed 2D map of the surroundings that may be utilized by a mobile robot to navigate (Filipenko and Afanasyev, 2018).

CHAPTER THREE : METHODOLOGY

Project can be divided into distinct and independent activities that needs to be carried out and integrated to produce the final result. Major steps include modelling, algorithm development, cost formulation, simulation and so on.

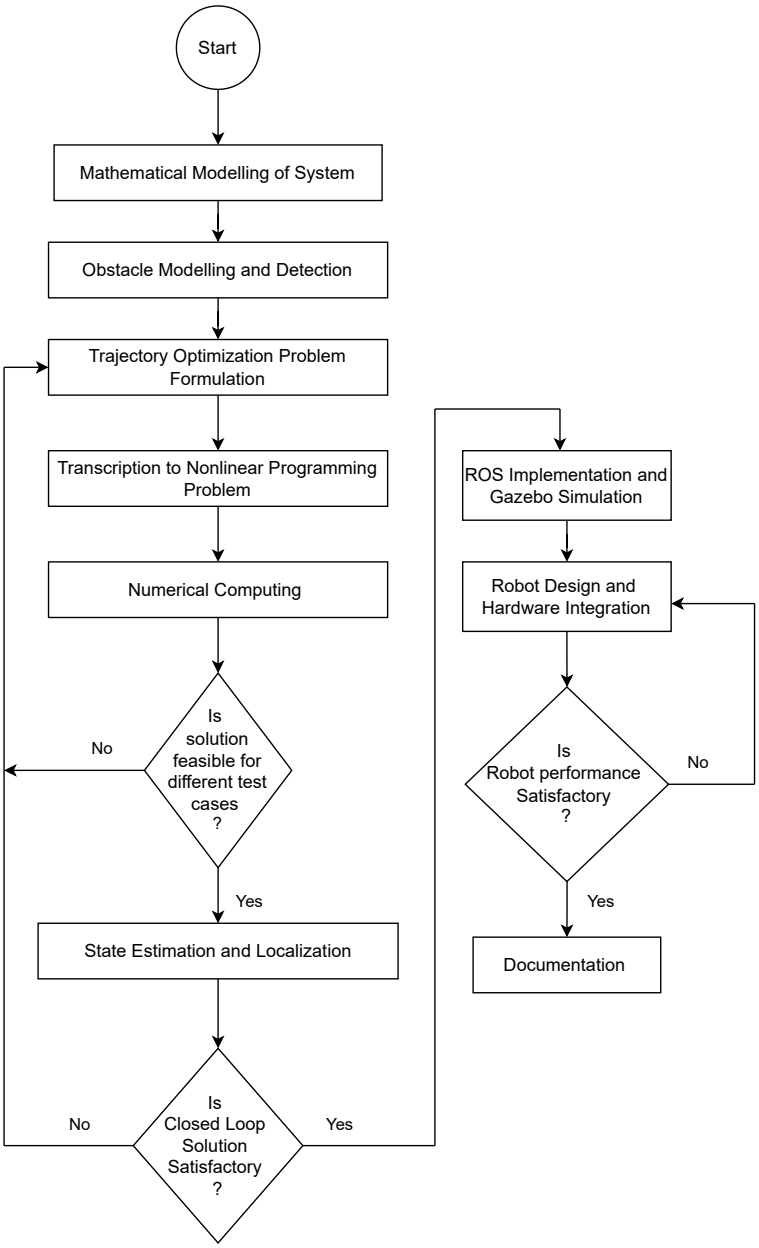


Figure 3.1: Research Methodology

3.1 System Modelling

Modelling of system is a method to represent the dynamic and kinematic aspects of all the system components in a mathematical form, generally differential equations that govern the input-output relation of the system. This includes developing the relations that best represents the response of the system for the given input, mathematically representing the system behavior, natural physical constraints and human imposed constraints. Mathematical analysis of the vehicle requires the development of some type of vehicle model, modelling of the environment in which the vehicle is supposed to operate, modelling of stationary as well as moving obstacles. These models along with their imposed constraints are solved using some form of numerical method to obtain the optimal solution while at the same time satisfying all the imposed constraints.

Modelling of the vehicle behavior can be done in different form depending upon the vehicle type and analytical requirement. The most basic method is to develop the kinematic model of the vehicle which gives kinematic response of the vehicle for given set of inputs. A set of force equations can be used to represent the response of the system to external actuating force. Initially, the kinematic models of the vehicle are developed without considering the actuating force. The response of such models for control inputs are observed and analyzed (Rubio and Albert, 2019).

3.1.1 Unicycle and Differential drive robot

Unicycle is one of the simplest models of wheeled robot with a single upright wheel that rolls on a flat surface. For unicycle model in two-dimensional plane with known position and orientation, the vehicle can be controlled with control input of linear velocity of the wheel and angular turning velocity. These control inputs are obtained from the controller and are optimal for the given conditions. The control inputs are constrained to some maximum and minimum values.

If x and y represent position of the wheel in Cartesian coordinate's and θ represents heading direction of the robot, then

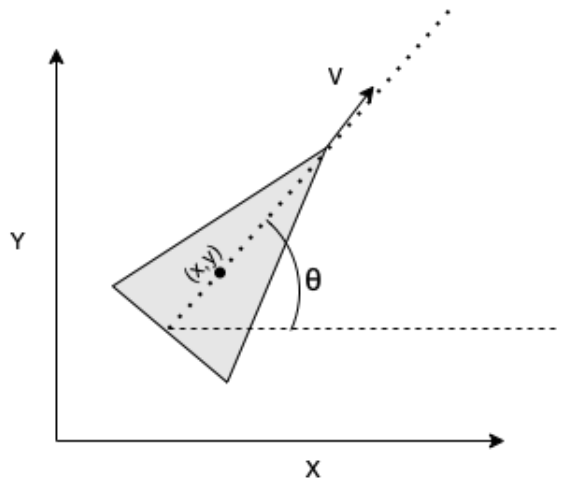


Figure 3.2: Unicycle Model

$$\dot{x} = v.cos(\theta) \tag{3.1}$$

$$\dot{y} = v.sin(\theta) \tag{3.2}$$

$$\dot{\theta} = \omega \tag{3.3}$$

Here v (forward velocity) and ω (angular turning velocity) are optimal control inputs from the controller.

The state-space representation of the above model can used to better observe the development of states throughout the motion of the vehicle. For a vehicle with states $X = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$ and $\dot{X} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$, a control input of U = $\begin{bmatrix} v \\ \omega \end{bmatrix}$ applied for time dt,

space state relation is written as,

$$\dot{X} = f(X,U) = \begin{bmatrix} cos(\theta) & 0 \\ sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \times U.$$

The new state of the vehicle is updated as,

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \\ \theta_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \\ \theta_n \end{bmatrix} + \Delta\,\mathrm{t}\cdot \begin{bmatrix} cos(\theta_n) & 0 \\ sin(\theta_n) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_n \\ \omega_n \end{bmatrix}$$

$$or, X_{n+1} = X_n + \Delta t \cdot A * U$$

The unicycle model can be used to derive the model of differential-drive robot. A differential drive robot consists of two wheels of radius r with two independent actuators which can rotate two wheels in the same axis independently. The speed difference between the two driven wheels provides maneuverability to the robot. A third low friction caster wheel/ball helps to keep the robot in horizontal position (Noga, 2006).

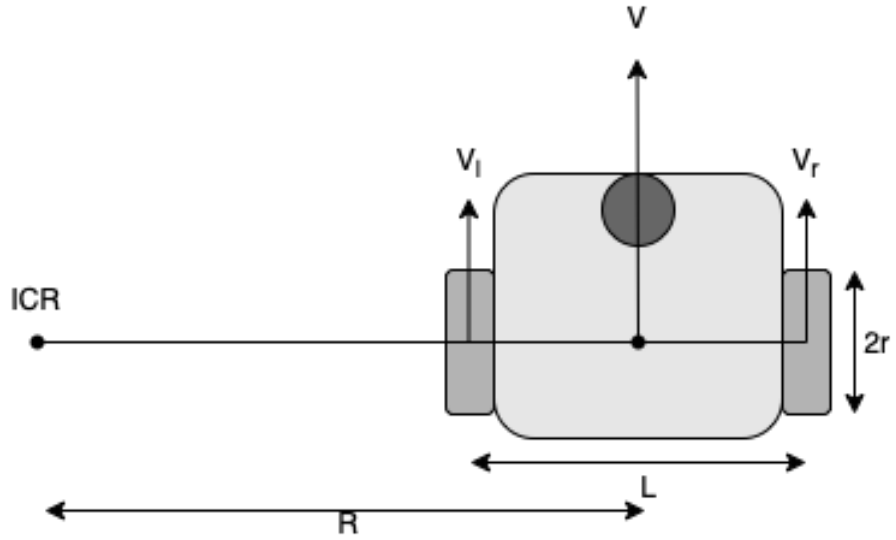


Figure 3.3: Differential Drive Robot

For a differential-drive robot with wheel base L and wheel radius r as shown in figure 3.3, ICR is the instantaneous center of rotation with radius R at any moment. The robot's linear velocity is V. If the robot's angular velocity is ω_v, the linear velocity is given as

$$V = \omega_v \cdot R$$
$$V_l = \omega_v \cdot (R - \frac{l}{2})$$
$$V_r = \omega_v (R + \frac{l}{2})$$

So,

$$\omega_v = \frac{(V_r - V_l)}{L} = (\omega_{vr} - \omega_{vl}) \cdot \frac{r}{L}$$

$$R = \frac{(\omega_{vl} r)}{2} \cdot (\omega_{vr} - \omega_{vl})$$
$$V = (\omega_{vr} + \omega_{vl}) \cdot \frac{r}{2}$$

Taking velocity components in x and y direction,

$$\dot{x} = \frac{r}{2} \cdot cos(\theta) \cdot (\omega_{vl} + \omega_{vr}) \tag{3.4}$$
$$\dot{y} = \frac{r}{2} \cdot sin(\theta) \cdot (\omega_{vl} + \omega_{vr}) \tag{3.5}$$
$$\dot{\theta} = \frac{r}{2} \cdot (\omega_{vl} + \omega_{vr}) \tag{3.6}$$

Here, wheel angular velocity for left and right wheels can be obtained as:

$$\omega_{vr} = \frac{2V + \omega L}{2r} \tag{3.7}$$
$$\omega_{vl} = \frac{(2V - \omega L)}{2r} \tag{3.8}$$

Where V, forward velocity and ω, angular turning velocity are the control inputs from the controller designed for unicycle robot. Analysis of response of a differential drive robot with wheel-base (L) = 0.15m and wheel radius (r) = 0.05m is performed and the results are discussed in the following pages.

3.1.2 Road Tyre Interaction

Forces and moments are developed in vehicle tires due to the action of friction. These forces and moments are very important parameters as they influence the handling and overall dynamics of the vehicle. Longitudinal and lateral forces developed on the tires are of particular importance as they have highest influence in the vehicle dynamics. These forces are observed to be the function of vertical load on tire (F_z), and slip ratio(κ). Slip ratio is defined as the ratio between slip velocity and vehicle velocity. Mathematically, it can be expressed as:

$$\text{Slip ratio}(\kappa) = \frac{(V_{wheel} - V_{vehicle})}{V_{vehicle}}$$

Slip ratio generally exists during braking, acceleration and cornering maneuver. Slip angle(α), is defined as the angle between the tire orientation and velocity vector of the vehicle. If V_x is the tire velocity component in the direction of tire orientation and V_y is the velocity component perpendicular to tire orientation, the slip angle (α) is given as:

$$\text{slip angle } (\alpha) = \frac{V_y}{V_x}$$

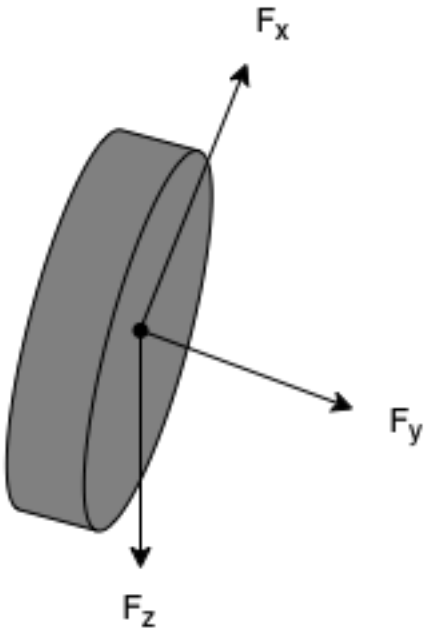


Figure 3.4: Forces acting on wheel

Various experiments have attempted to find the relation between tire forces and tire slip angle. Of the many mathematical tire force models, Magic formula tire model developed by Hans B. Pacejka (H.Pacejka, 2012) has been widely used to calculate tire force and moment characteristics. Longitudinal (F_x) and lateral (F_y) forces on a tire as given by magic formula are as follows:

$$F_x = D{\cdot}sin(C{\cdot}tan^{-1}(B\kappa - E[B\kappa - tan^{-1}(B\kappa)])){\cdot}F_z \tag{3.9}$$

$$F_y = D{\cdot}sin(C{\cdot}tan^{-1}(B\alpha - E[B\alpha - tan^{-1}B\alpha])){\cdot}\mu{\cdot}F_z \tag{3.10}$$

The constant coefficients B, C, D, E are used to appropriately represent the curvature characteristics representing respectively stiffness, shape, peak and curvature characteristics. Values of these constants depend upon the kind of contact characteristics between contact surface and tyre. μ is the coefficient of friction between road and tire.

Lateral tire force has a very complex, empirical, non-linear relationship with tire slip angle as illustrated by the figure. But as can be observed in the figure 3.5, for small slip angle(<8 degrees) the lateral force can be assumed to be linear function of slip angle. Small slip angles occur in lower velocity region. Hence, linear approximation of lateral tire force with tire slip angle is appropriate for our low-speed application.

Linear approximation of lateral tire force is given as:

$$F = c{\cdot}\alpha$$

Where, c is tire cornering stiffness and α is slip angle.

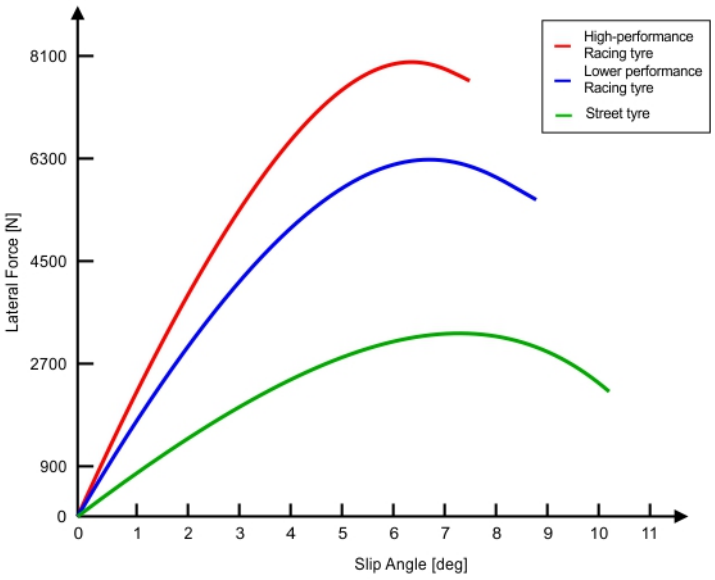


Figure 3.5: Lateral tire force vs. tire slip angle

3.1.3 Dynamic Vehicle Model

Four-wheel car like vehicle with Ackerman Steering can be modelled using forces acting on the four wheels of the vehicle. The forces as given by Magic formula causes the motion of vehicle and vehicle maneuvering can be obtained by appropriately changing the components of the forces acting on the tires (Lynch M. and C., 2013)

By the use of Newton's law of motion and some basic geometric relationships, the longitudinal velocity $v_x(t)$, the lateral velocity $v_y(t)$ and the yaw rate $r(t)$ measured around the Center of Gravity (CG) of the vehicle can be described by the following three differential equations:

$$\ddot{x} = \dot{y}{\cdot}r + \frac{1}{m}((F_{xFL}+F_{xFR}){\cdot}cos(\theta) - (F_{yFL}+F_{yFR}){\cdot}sin(\theta) + F_{xRL} + F_{xRR}) \ldots \quad (3.11)$$

$$\ddot{y} = -\dot{x}{\cdot}r + \frac{1}{m}((F_{xFL}+F_{xFR}){\cdot}sin(\theta) + F(_{yFL}+F_{yFR}){\cdot}cos(\theta) + F_{yRL} + F_{yRR}) \ldots \quad (3.12)$$

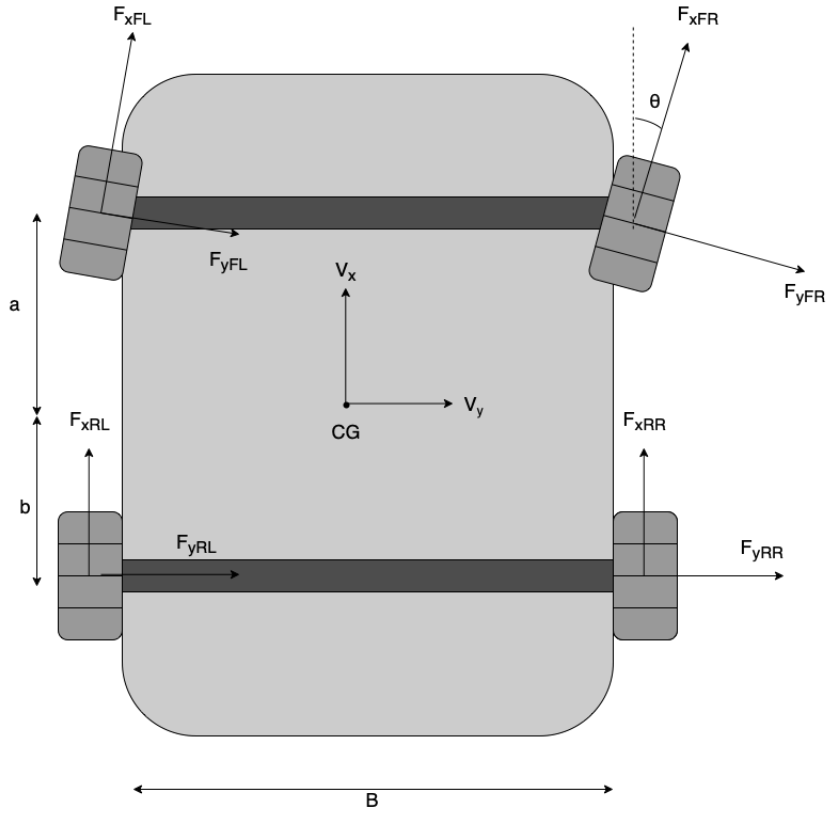


Figure 3.6: Schematic View of vehicle dynamic system

$$\dot{r} = \frac{1}{I}(a{\cdot}((F_{xFL} + F_{xFR}){\cdot}sin(\theta) + F(_{yFL}{+}F_{yFR}){\cdot}cos(\theta) - b{\cdot}(F_{yRL} + F_{yRR}))) \ldots \tag{3.13}$$

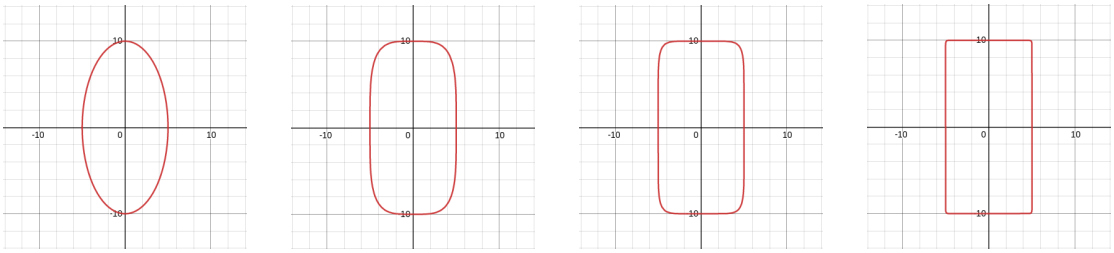
Here, m is mass and I is moment of inertia of the vehicle.

3.2 Simulated Obstacles

Dynamic as well as stationary obstacles were modeled and simulated in the environment. There are also restrictions on state and control values, which are treated as obstacles in the configuration space. Then the objective of our optimization problem is to find a set of control-state that minimizes a cost function subject to dynamic/kinematic equation constraints, obstacle constraints as well as constraints on state and control.

$$\left\{((x - x_c)/a)^p + ((y - y_c)/b)^p\right\} = 1 \tag{3.14}$$

For even number values of $p > 1$, the above mathematical equation 3.14 represents a 2 dimensional closed shape and can be modeled as an obstacle in the map. (x_c, y_c) determine the center location of the obstacle. The value of a and b determine the size of the obstacle, and p determines the shape of the obstacle.



(a) $p = 2$ (b) $p = 4$ (c) $p = 8$ (d) $p = 50$

Figure 3.7: Shapes obtained at different values of p for $\{(x/5)^p + (y/10)^p\} = 1$

Figure 3.11 shows how the shape can be changed by altering the value of p. The center location of $(x_c, y_c) = (0, 0)$ and size parameters $a = 5$ and $b = 10$ are used in the equation 3.14. The equation represents an ellipse for $p = 2$, and as the value of p is increased the shape tends to resemble a rectangle of size 10*20.

Circular shaped obstacle with center at (x_c, y_c) and a radius of r is modeled by substituting $a = b = r$ and p = 2 in equation 3.14.

$$(x - x_c)^2/r^2 + (y - y_c)^2/r^2 = 1 \tag{3.15}$$

$$h(x, y) = ln\{(x - x_c)^2/r^2 + (y - y_c)^2/r^2\} = 0 \tag{3.16}$$

For all values of robot position (x, y), it is required that $h(x, y) > 0$. Further, for robustness, a function $r(x, y)$ is defined as: $r(x, y) = e^{\alpha * e^{-h}}$ where, α is a parameter to be tuned. The value of $r(x, y)$ is to be minimized.

3.3 Obstacle detection and Modelling

Detection and modelling of obstacles is necessary to identify stationary as well as moving obstacles and maintain the condition that the obstacles are avoided while generating the trajectory. The obstacles present in the environment are detected and mathematically modelled to incorporate them into the constraints and into objective functions.

Sensing of obstacles and the environment was performed using YDLIDAR. LIDAR sensor detects the obstacles in point cloud form.Sensing of obstacles and the environment was performed using YDLIDAR. We have used the “obstacle_detector” ROS package to model the obstacles from the point cloud data received from the laser scan. The ROS package models obstacles into circular and line segments and gives coordinates of center of the circle and radius in case of circular obstacle models and coordinates of end-points

of line segments in local coordinate frame of the scanner. The package allows for tuning of certain scan parameters such as minimum and maximum scan range, minimum and maximum range of obstacles modelled. The scanned point cloud data and modelled obstacles can be visualized through the "obstacle_visualizer" node.

The obstacle_detector package models obstacles in two geometrical shapes: circle and straight-line segment. The coordinates obtained from the package are first converted to global frame of reference. Then, the circular obstacles are represented using an equation of circle as:

$$(x-h)^2 + (y-k)^2 = r^2$$

Where, (h, k) represents the center of circle, r the radius and (x, y) the current robot location.

Line segment information is obtained in the form of coordinates of the end point. The coordinates are transformed into global frame and obstacle is modelled as an ellipse centered at mid-point of two end-points and with major axis the line segment itself, the minor axis is manually provided. Equation of elliptical obstacle in global frame of reference is:

$$\begin{aligned}(x-h)^2\left(\frac{cos^2\theta}{a^2}+\frac{sin^2\theta}{b^2}\right)+2(x-h)(y-k)sin\theta cos\theta\left(\frac{1}{a^2}-\frac{1}{b^2}\right)\\ +(y-k)^2\left(\frac{sin^2\theta}{a^2}+\frac{cos^2\theta}{b^2}\right)=1\end{aligned} \tag{3.17}$$

where, for two end points of line segment $P(x_1, y_1)$ and $Q(x_2, y_2)$,

$$\text{Ellipse centre: } (h,k) = (\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}) \tag{3.18}$$

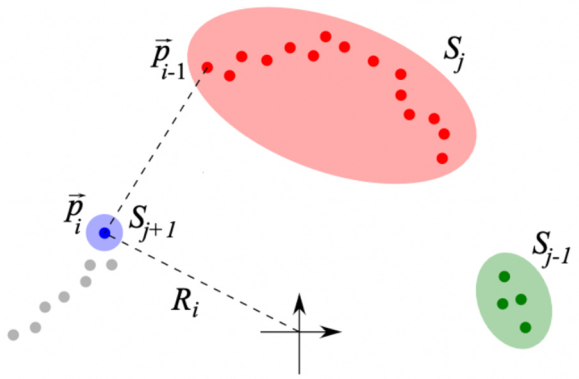
$$\text{Major axis: } a = \frac{1}{2}\sqrt{(x_2-x_1)^2+(y_2-y_1)^2} \tag{3.19}$$

$$\text{Ellipse orientation: } \theta = tan^{-1}(\frac{y_2-y_1}{x_2-x_1}) \tag{3.20}$$

3.3.1 Modelling Of line segment obstacles

Modelling of obstacles from LIDAR csan data points into geometric shapes are carried out in accordance with the work of (Przybyla, 2017). Modelling of obstacles starts with

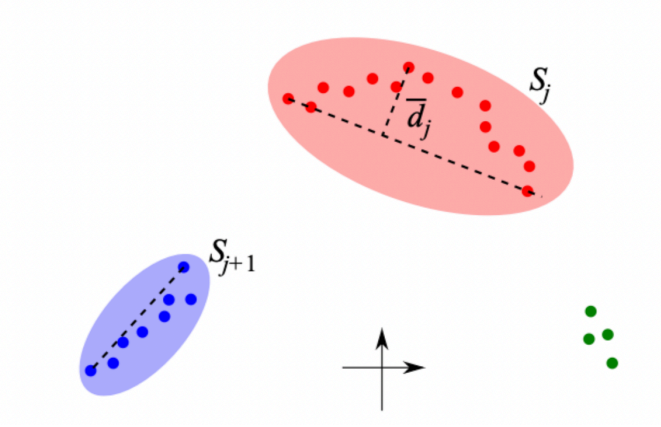
grouping of cloud data points. Groups are distinguished by examining the distance between two consecutive points. Grouping is done to provide a collection of point subsets possibly representing separate objects. Any point P_i is assigned to a group S_j of point P_{i-1} if the user defined criterion of minimum Euclidean distance is satisfied as shown in 3.8. If the criterion is not met, a new group S_{j+1} is created and the whole grouping process is started again. Resulting points constitute point subsets that are subject to the process of splitting.



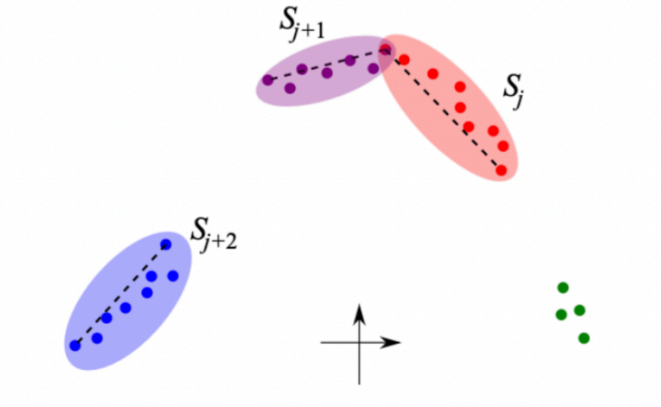
Source: (Przybyla, 2017)

Figure 3.8: Grouping of cloud data points

Each point subset S_j is examined for possible splitting into two separate subsets. Groups consisting of less than N_{min} points are not subject to this procedure. The procedure of splitting relies on iterative end point fit algorithm, which draws a leading line between two extreme points of a point subset, and seeks the point of group that lies farthest from the line. Then the subset is split into two subsets if it meets the criterion of maximum distance between line and the point examined. This procedure is repeated recursively for each new subsets until no more splitting occurs. Then a line segment is drawn between two extreme points of a subset which acts as a line segment obstacle later modelled into an elliptical obstacle.



(a) Testing for split requirement



(b) Splitting of a group into two subsets

Source: (Przybyla, 2017)

Figure 3.9: Splitting Process of a Point cloud group

3.3.2 Modelling of circles

For modelling of circular obstacles, for each line segment previously created, an equilateral triangle is constructed with its base coinciding with the segment and its central point placed away from the origin. Next, on the basis of this triangle, an inscribed circle is constructed with radius :

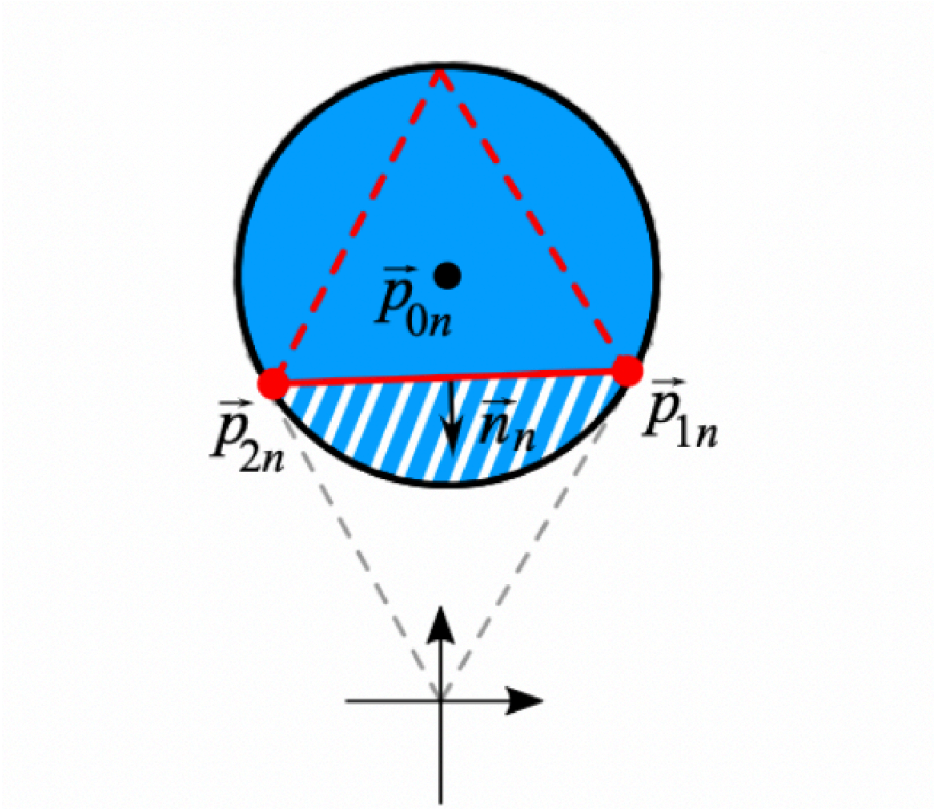
$$r_n = \frac{1}{\sqrt{3}} l_n \tag{3.21}$$

Where l_n denotes segment length and with centre at:

$$\vec{p_{0n}} = \frac{1}{2}(\vec{p1n} + \vec{p2n} - r_n \vec{n_n}) \tag{3.22}$$

Where represents segment normal vector pointing towards origin. Then the resulting circle is enlarged with a user defined margin r_d. If the resulting radius is smaller than the maximum user-defined threshold r_{max}, the circle is selected as the obstacle and original

line segment is discarded.



Source: (Przybyla, 2017)

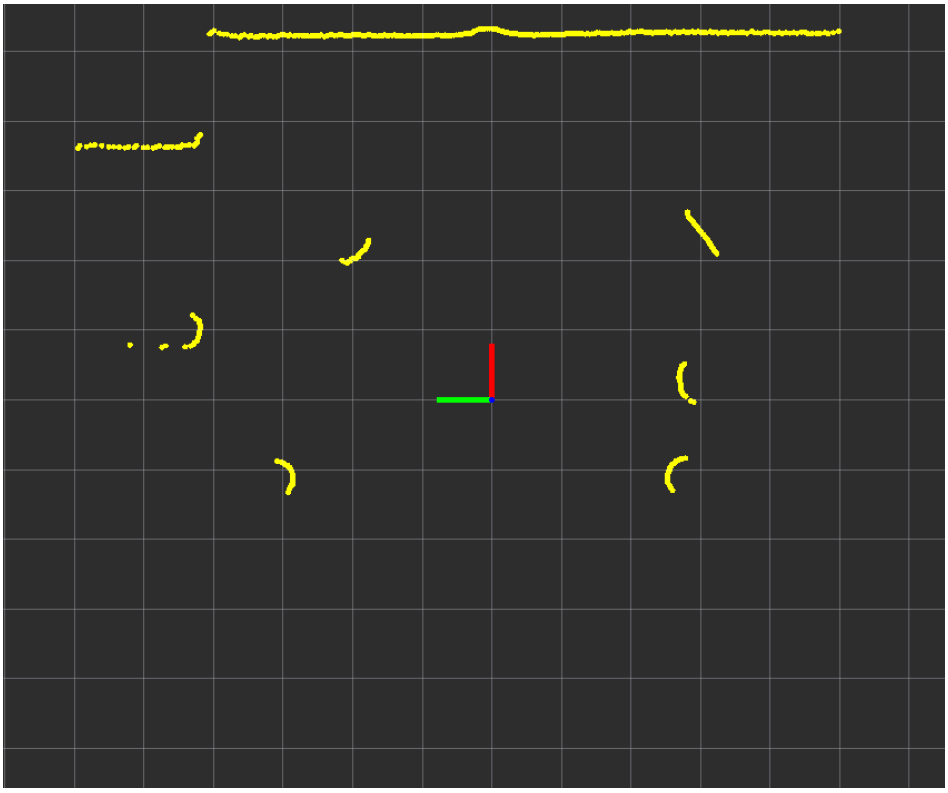
Figure 3.10: Modelling circular obstacle

ROS communication architecture

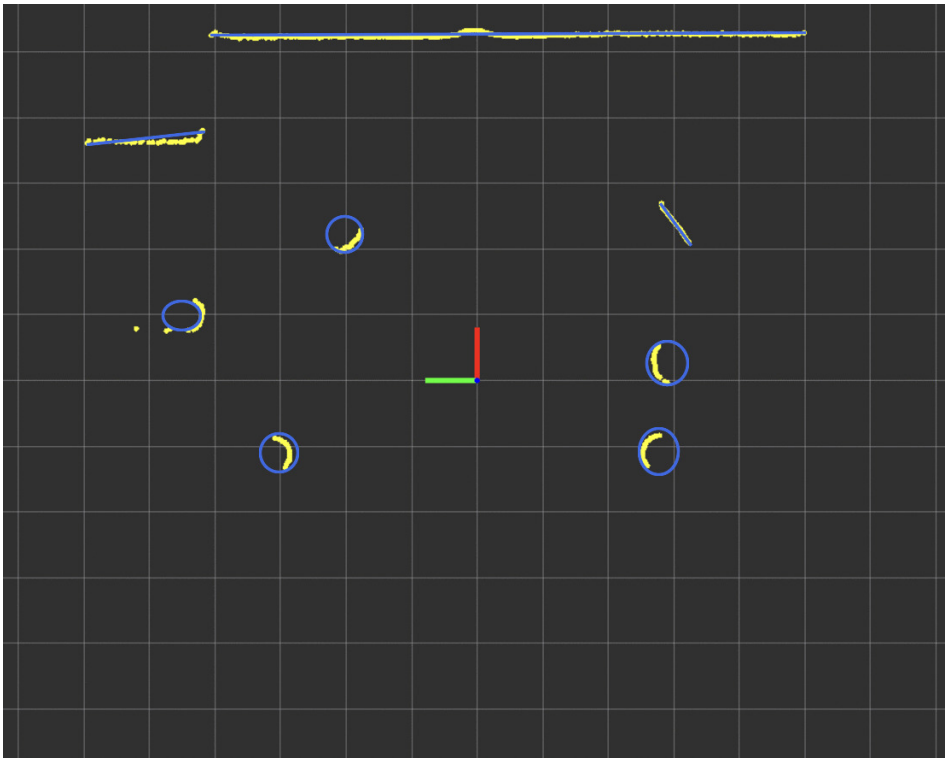
ROS communication architecture for detection and modelling of obstacles is explained as follows:

The main node, obstacle_detector, which converts messages of type sensor_msgs/ LaserScan from topic scan or messages of type sensor_msgs/PointCloud from topic pcl into obstacles, which are published as messages of custom type obstacles_detector/Obstacles under topic obstacles. The point cloud message must be ordered in the angular fashion, because the algorithm exploits the polar nature of laser scanners. The obstacle_detector node modells the scan data nto circular and line segment data and gives following custom messages:

1. CircleObstacle
    - geometry_msgs/Point center



(a) Obstacle Cloud points



(b) Modelled Obstacles

Figure 3.11: Modelling of obstacles from scan result

- float64 radius

2. SegmentObstacle
   - geometry_msgs/Point first_point
   - geometry_msgs/Point last_point

3.4 Problem Definition

Cost Function:

The cost function is a Performance Measure designed such that its minimization ensures a desirable performance of the system. The cost function (J) is designed for following set of problem scenario:

a. The goal state of the robot is $X(t = t_f) = X_f$.

b. The control effort U is to be kept as low as possible.

c. Obstacle should be avoided. i.e. $r(x, y) = (e^{\alpha * e^{-h}})$ is to be minimized.

Following cost function is designed to incorporate above objectives:

$$Minimize : J = \int_{t_0}^{t_f} \big\{c_1[x(t) - x_f]^2 + c_2[y(t) - y_f]^2 + c_3[\theta(t) - \theta_f]^2 + c_4[v(t)]^2 + c_4[\omega(t)]^2 + r(x, y)\big\} dt. \tag{3.23}$$

Constraints:

a. Dynamics constraint: $\dot{X} = f(X, U)$

b. Initial state: $X(t = t_0) = X_0$

c. Box Constraint: $x_{min} \leq x \leq x_{max}$ and $y_{min} \leq y \leq y_{max}$

d. Velocity limit: $v_{min} \leq v \leq v_{max}$

e. Angular velocity limit: $\omega_{min} \leq \omega \leq \omega_{max}$

f. Obstacle Constraints: $h(x, y) > 0$

3.5 Optimal Trajectory Generation

The solution to the optimization problem generates optimal trajectory for the robot as a set of control and states. Pontryagin's minimum principle gives necessary conditions for optima and can be used to analytically solve an optimization problem in terms of state

and co-state vectors. However, it is almost impossible to find an analytical solution if large dimensional state and control vectors are involved. There are various numerical methods and algorithms to solve the optimization problem. The problem (cost as well as constraints) are discretized and transformed into a Non Linear Programming Problem (NLP) before solving for the values of control and states in the nodal points. The Goal of the above nonlinear optimization programming problem is to determine optimal values of X and U such that the objective function J is minimized.

The optimal control problem (3.23) is transcribed into a non linear programming problem by discretizing the objective function and the constraints (Kelly, 2017).

$$Minimize: J = \sum_{k=1}^{N} \big\{ c_1 * [x_k - x_f]^2 + c_2 * (y_k - y_f)^2 + c_3 * (\theta_k - \theta_f)^2 + c_4 * [v_k]^2 + c_4 * [\omega_k]^2 + r(x_k, y_k) \big\}. \tag{3.24}$$

Subject to:

$X_{k=1} = X_0$

for k = 1:N

$X_{k+1} = X_k + f(X_k, U_k) * \Delta t$

$x_{min} \leq x_k \leq x_{max}$

$y_{min} \leq y_k \leq y_{max}$

$v_{min} \leq v_k \leq v_{max}$

$\omega_{min} \leq \omega_k \leq \omega_{max}$

$h(x_k, y_k) > 0$

Goal of the above nonlinear optimization programming problem is to determine optimal values of X and U such that the objective function J is minimized. For a system with n number of states and m number of control inputs, X is a n by $N + 1$ matrix and U is a m by N matrix. So the total number of optimization variables are $n * (N + 1) + m * N$. The above formulation is called Multiple Shooting Method. The other method known as Single Shooting Method uses only the elements of U_{m*N} as optimization variables and X is obtained according to the rule: $X_{k+1} = X_k + f(X_k, U_k) * \Delta t$.

Direct single shooting method approximates the trajectory using a simulation.The decision variables in the nonlinear program are an open-loop parameterization of the control along the trajectory, as well as the initial state. Direct shooting is well suited to applications where the control is simple and there are few path constraints, such as space flight (Betts, 1998). Multiple Shooting Method, also known as parallel shooting, is an extension of the direct single shooting method. Rather than representing the entire trajectory as a single simulation, the trajectory is divided up into segments, and each segment is represented by a simulation. Multiple shooting tends to be much more robust than single shooting, and thus is used on more challenging trajectory optimization problems (Betts, 1998).

Example Problem

Consider a unicycle robot initially at $(x, y) = (0, 0)$ heading towards an axis aligned with the abscissa (i.e $\theta = 0$). We desire to navigate the robot to a new state of $(x, y, \theta) = (10, 10, 0)$. There exists a circular obstacle of diameter 1 unit at location $(x, y) = (5, 5)$. So, $h(x_k, y_k) = ln\{((x_k - 5)/0.5)^2 + ((y_k - 5)/0.5)^2\}$ and $r(x_k, y_k) = (e^{5*e^{-h}})$.

$$Minimize : J = \sum_{k=1}^{N} \{[x_k{-}10]^2{+}(y_k{-}10)^2{+}(\theta_k{-}0)^2{+}0.5{*}[v_k]^2{+}0.5{*}[\omega_k]^2{+}r(x_k, y_k)\}. \tag{3.25}$$

Subject to:

$X_{k=1} = [0, 0, 0]^T$

for k = 1:N

$X_{k+1} = X_k + f(X_k, U_k) * \Delta t$

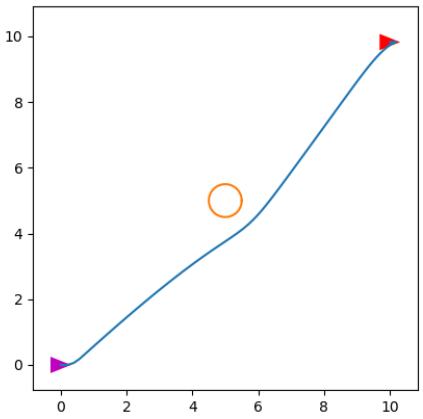
$-1 \leq v_k \leq 1$

$-1.5 \leq \omega_k \leq 1.5$

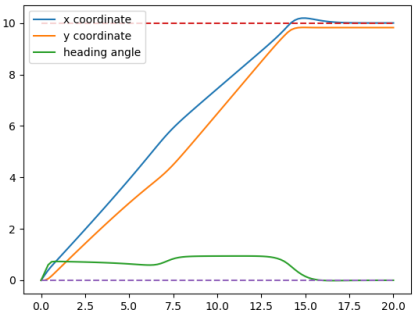
$h(x_k, y_k) > 0$

The above problem is solved for N = 100 and Δt = 0.2 seconds. This means an optimal trajectory (control-state set) is generated for the next 20 seconds. The problem has a total of 503 optimization variables to solve for. The problem is implemented in CasADi framework and solved using the interior point solver. Figure 3.12 illustrates the solution

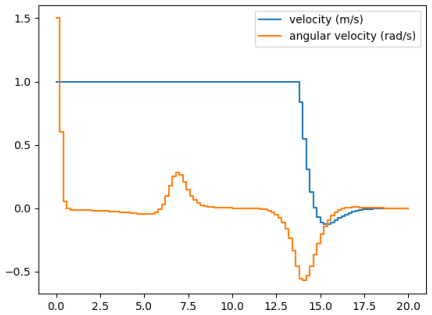
of the example problem. It can be observed that the control signals (v and ω) drive the unicycle robot to final state with minimal error.



(a) Path traced



(b) Variation of states with time



(c)

Figure 3.12: Solution of problem formulated in equation 3.25

Receding Horizon Control

Receding Horizon Control requires solving the nonlinear programming problem repeatedly over a moving finite time horizon (Mattingley et al., 2011) . The NLP is solved

for a finite time horizon (τ) while just implementing it on the current timeslot (Δt) and then repeating the optimization over and again. Also known as Model Predictive Control (MPC), this control scheme is capable of foreseeing future events and taking control actions in response. Since it works on real time information updates, a feedback control system is realized making it more robust to noises and uncertainties.

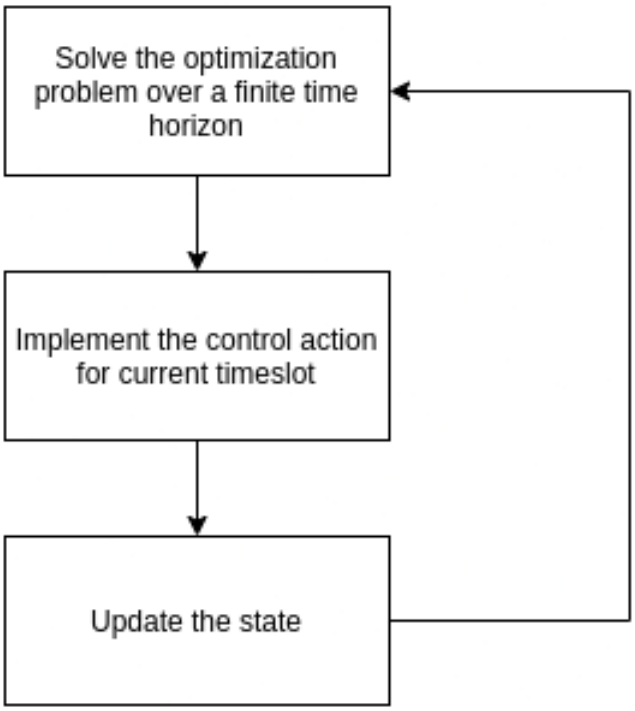
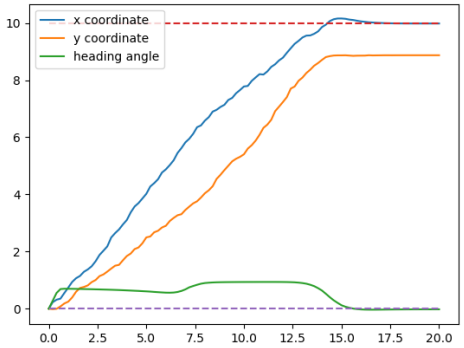


Figure 3.13: Model Predictive Control workflow

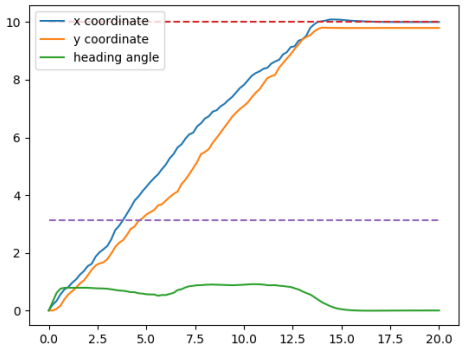
While the open loop optimal control determined as Figure 3.12 works good for a perfectly modeled system, it fails to drive the system to final state if certain degree of uncertainty is present in the model. The system is simulated by adding Gaussian noise with SD proportional to the control signal. It can be noted in figure 3.14 that the closed loop control obtained by MPC ($\tau = 2sec$,Δt=0.2sec) drives the system almost to the final state besides the presence of disturbances.

3.6 Solver Selection

Very often the equations that describe the relationship between the quantities in a process are of non-linear nature. In addition, many of these variables have to stay within certain intervals. These constraints are introduced either to set the physical capabilities (e.g., vehicle speed), meaningfulness of the variables (e.g., mass being positive)



(a) Open Loop Control



(b) Receding Horizon Control

Figure 3.14: Solution of problem formulated in equation 3.25 in presence of system noise

or to guarantee certain operational constraints(e.g., obstacle avoidance). These types of non-linear programs arise in many classes of problems, dynamic optimization being a common case (Frasch, 2015). Finding the optimal, time dependent profile for control of a system, whose states are changing involves solving a non-linear problem. If discretization approach is used, the constraint functions are divided into multiple blocks of process model equations at the same time having to maintain continuous profile of state variables over time. Also, as the number of discretization point increases, the accuracy of the dynamic optimization problem increases, so it becomes desirable to have large number of discretization point at the cost of increasing optimization variables and making the problem more complex.(Andreas Wachter, 2005) The programs that provide the solution of such non-convex, non-linear, continuous and smooth optimization problems are called Non-linear Programs(NLPs). Several methods exist for the solution of such

non-linear problems. One of the most efficient and most commonly used algorithms is Interior Point Optimizer (IPOPT).

IPOPT is an open-source software package for large-scale non-linear optimization. It is used to solve the non-linear optimization problems of the form :

$$\begin{aligned} \min_{x \in R^n} \quad & f(x) \\ \text{s.t.} \quad & g^L \leq g(x) \leq g^U \\ & x^L \leq x \leq x^U \end{aligned} \tag{3.26}$$

where x$\in$ R are the optimization variables with lower and upper bounds of x^L and x^U, f: R $\longrightarrow$R^n is the objective function with non-linear constraints g(x). IPOPT solver can be used to solve the optimization problem with both f(x) and g(x) being either linear or non-linear, convex of non-convex as long as they are twice continuously differentiable. (Andreas Wachter, 2005)

3.7 State Estimation and Localization

A system which is not observable cannot be controlled. Therefore, state estimation and localization are the integral parts of autonomous navigation. Sensor measurements are noisy in nature and can not give the true value of a quantity being measured. This necessitates algorithms to find the best estimate of the measurement given some assumptions about our sensor and the external world.

State estimation of wheeled robot roughly consists of two main tasks: Estimation of vehicle dynamics state and localization of vehicle (Wischnewski et al., 2019). The former is obtained from the system modelling and the latter is the method by which we determine position and orientation of a robot with reference to the environment. By reviewing the literature, a modified version of Kalman Filter called Extended Kalam Filter can be applied to estimate state which can handle non-linearity of state equation. The algorithm relies on linearization of the system and follows the alternating process of prediction and correction of standard Kalman Filter.

At a given time step 'k', the system dynamics of the robot can be written as:

$$x_k = f\big(x_{(k-1)}, u_{(k-1)}, w_{(k-1)}\big) \tag{3.27}$$

This equation can be converted into a linear discrete time state space model of the form:

$$x_k = A_{(k-1)}x_{(k-1)} + B_{(k-1)}u_{(k-1)} + w_{(k-1)} \tag{3.28}$$

Here, $A_{(k-1)}$and $B_{(k-1)}$ are the Jacobian matrices representing how the state of the system changes from time k-1 to k with the absence and presence of the control command respectively.

The observation model is given by:

$$h_k = H_k x_k + v_k \tag{3.29}$$

The process noise $w_{(k-1)}$ and measurement noise v_k are assumed to be a normally distributed white noise (uncorrelated over time) with zero mean. The co-variance matrices of process noise and measurement noise are denoted by Q_k and R_k respectively. The measurement matrix H_k converts the predicted state estimated at time step k into predicted sensor measurement at the same time step. In our simulation we have assumed H_k to be identity matrix signifying the fact that all states are directly measured from the sensor data.

Prediction:

We will denote the maximum likelihood estimate for time step k based on the information available up to time instance k-1 by $\hat{x}_{(k|k-1)}$. Applying the system equation we obtain the best possible estimate at time k.

$$\hat{x}_{(k|k-1)} = A\hat{x}_{(k-1\,|\,k-1)} + Bu_{(k-1)} \tag{3.30}$$

From the definition of co-variance and the underlying random process,

$$P_{(k|k-1)} = AP_{(k-1|k-1)}A^T + Q_k \tag{3.31}$$

Correction:

Kalman gain matrix K_k can be calculated based on statistical properties of system and measurement residual (y_k) which is the difference between actual sensor measurement (z_k) and predicted sensor measurement (h).

$$y_k = z_k - h \tag{3.32}$$

$$S_k = H_k P_{(k|k-1)} H_k^T + R_k \tag{3.33}$$

$$K_k = P_{(k|k-1)} H_k^T S_k^{-1} \tag{3.34}$$

Update:

By incorporating both system dynamics and measurement residual with optimal gain, new state estimation can be obtained.

$$\hat{x}_{(k|k)} = \hat{x}_{(k|k-1)} + K_k y_k \tag{3.35}$$

$$P_{(k|k)} = (I - K_k H_k) P_{(k|k-1)} \tag{3.36}$$

To summarize, the designed filter function algorithm which estimates the new state from the knowledge of previous states and sensor measurements is as follows:

1. Input $\hat{x}_{(k|k-1)}$, $P_{(k-1|k-1)}$, Z_k, A_k, Q_k,H_k, R_k
2. Predict state covariance matrix $P_{(k|k-1)}$.
3. Obtain measurement model, h_k
4. Calculate measurement residue, y_k and its covariance, S_k
5. Calculate near optimal Kalman gain, K_k
6. Update estimated state $\hat{x}_{(k|k)}$ and its covariance $P_{(k|k)}$.
7. Return $\hat{x}_{(k|k)}$ and $P_{(k|k)}$

A key thing to note is that, the filter already takes predicted state estimation at time step 'k' which is obtained from discretization and linearization of nonlinear state equation by either Euler or RK-4 method. While implementing the filter for various kinematic and dynamic vehicle model, the value of A Matrix changes, for example for unicycle model its value is identity. The measurement matrix H_k and hence the measurement model h depends on the types and the number of sensors used in our vehicle. The estimation of state, number of sensors to be incorporated and the state to be measured or estimated from the observational model fundamentally relies on the observability analysis.

3.8 Simulation and Visualisation

Algorithms discussed were first tested using programming and numeric computing platform MATLAB. Simulations were performed for different test cases. Finally, the code was reiterated in Python to facilitate the ROS implementation and Gazebo simulation.

3.8.1 Turtlebot3

Turtlebot is a low-cost, personal robot kit with open-source software that runs on the ROS standard platform (Ackerman, 2017). The TurtleBot3 was created in 2017 with

features to fill the absent functionalities of its predecessors, as well as user needs. TurtleBot3 is a compact, modular, inexpensive, and programmable next generation mobile robot for use in education, research, hobby, and product development (Guizzo and Ackerman, 2017).

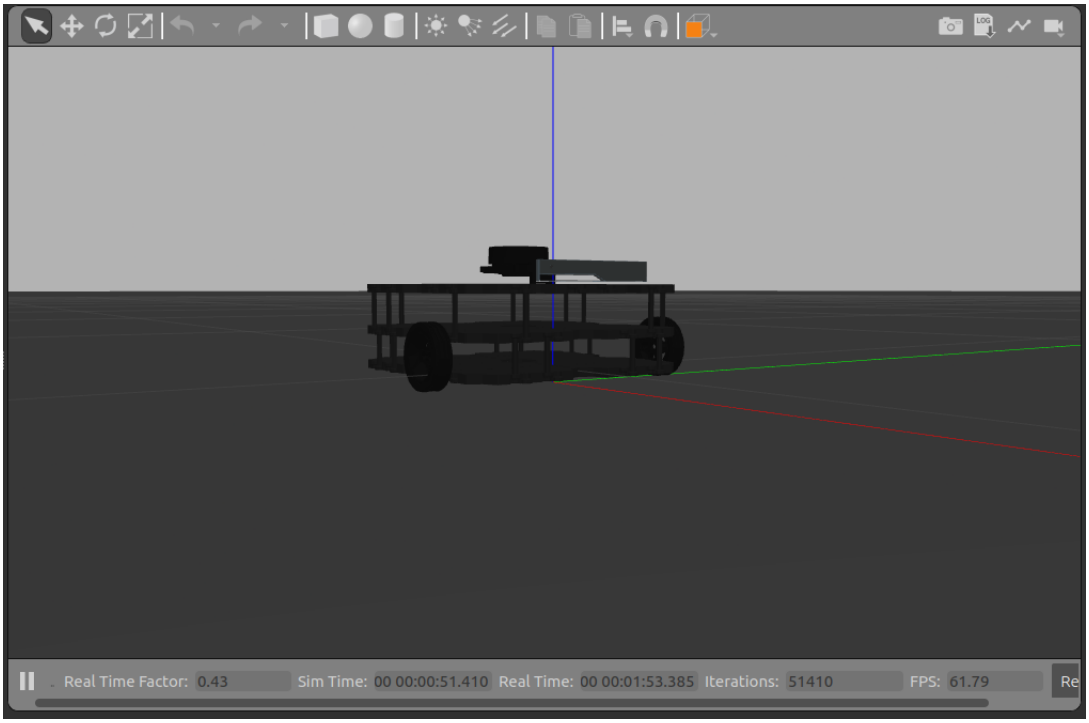


Figure 3.15: Turtlebot3 Waffle in Gazebo Empty World

The ”burger” and ”waffle” variants of Turtlebots are the two major configurations that are available. The waffle model is more expensive than the burger model due to its greater size and additional camera sensor (Amsters and Slaets, 2019). Our control system is tested on the Turtlebot3 Waffle model. Since the Turtlebot3 Gazebo Simulation relies on the ROS Gazebo package, the necessary Gazebo version for ROS is installed before the simulation can be conducted (Guizzo and Ackerman, 2017).

The turtlebot subscribes to $/cmd\_vel$ topic to take velocity commands and publishes the state updates to the $/odom$ topic. On the other hand, the trajectory planner node subscribes the $/odom$ topic, solves for control-state pair, and publishes the control to $/cmd\_vel$ topic.

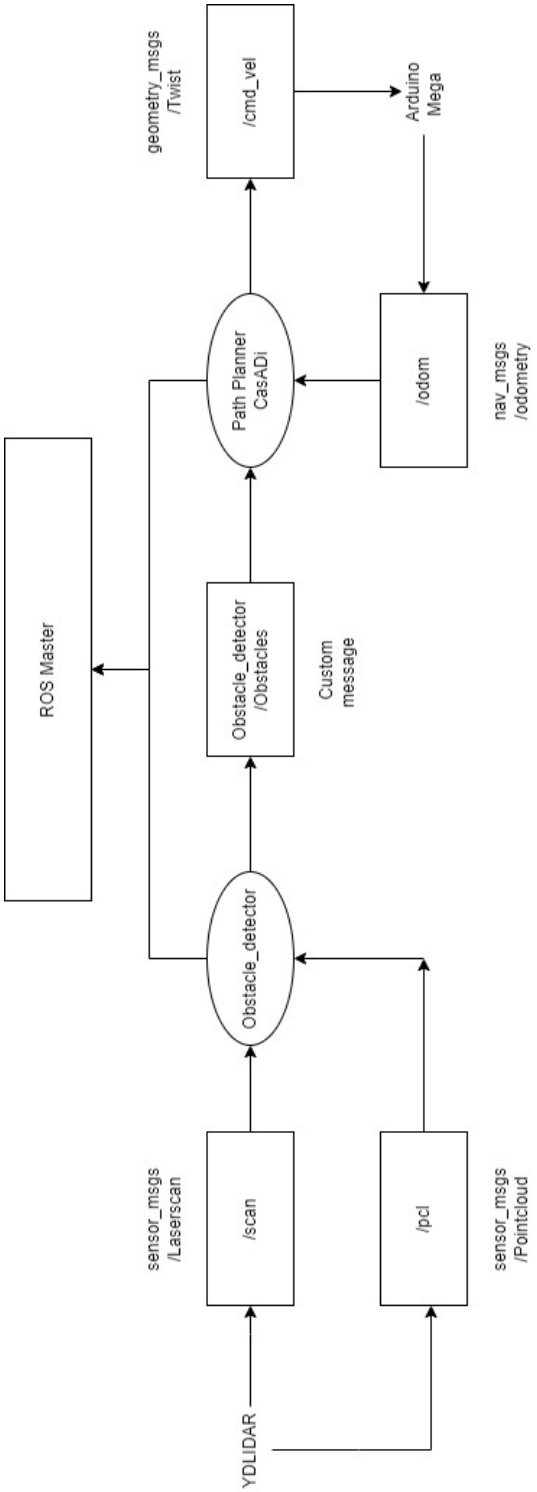


Figure 3.16: ROS node architecture

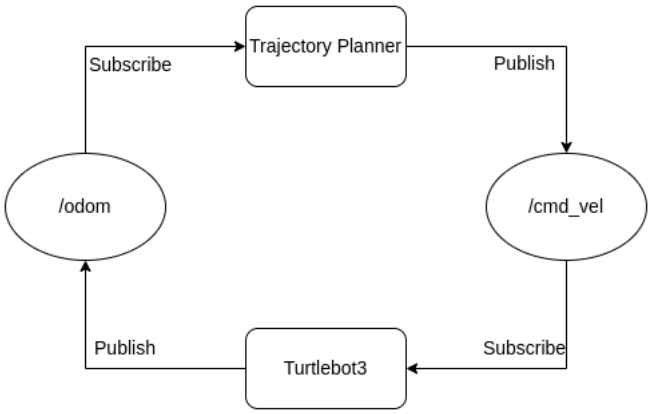


Figure 3.17: Basic architecture of Turtlebot3 simulation

3.9 Hardware Integration

After navigating the Turtlebot_3 Waffle robot with a favorable outcome in the ROS environment for different test scenarios, various mechanical and electrical components were selected and assembled to fabricate a differential drive-wheeled robot. The main parts of differential drive robot include sensors, controllers, actuators, and the power source.

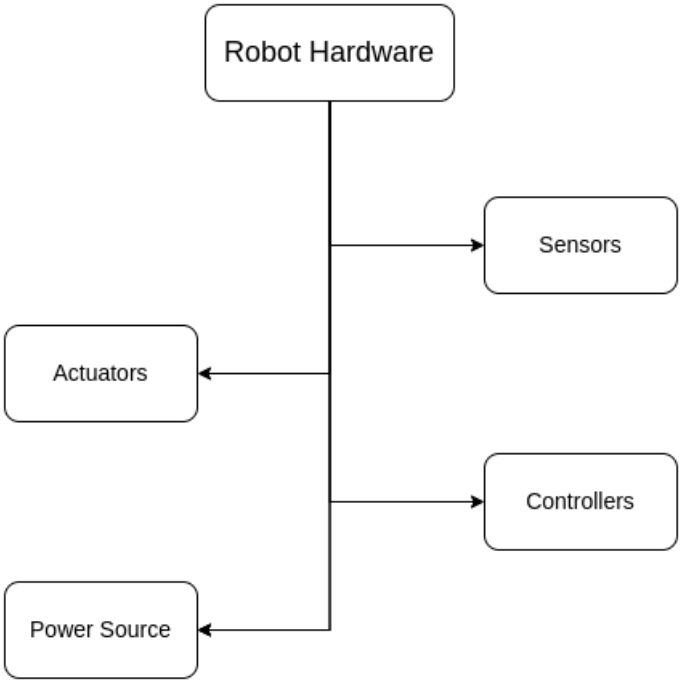


Figure 3.18: Robot Hardware Components

3.9.1 Sensing Units

To gather information of surrounding and to acquire knowledge of robot states, various sensors were equipped on board.

Lidar

YDLIDAR G4 was used to model the environment and detect obstacles. It is a 360° 2D Lidar having high precision scanning capabilities in both clockwise and anticlockwise direction. Based on the principle of triangulation, it can scan environment and the smallest obstacles at a range of 16m with scanning rate up to 9,000 times per second. It provides wireless data communication by employing OptoMagnetic technology.

To be able to observe point cloud data scanned by G4, it is integrated to ROS Framework via USB Type-C Cable and USB Adapter Board. It further requires additional auxiliary power of +5V for stable operation. It operates within 5-12Hz frequency with starting current of 1A. Equipped with a brushless motor and encoder disc mounted on bearings, it draws maximum sleeping current of 50mA when system sleeps. When motor is running at 7Hz it typically draws 500mA of maximum current

It can detect obstacles just above 41mm from its base surface at minimum ranging distance changing from 0.12m to 0.28m depending on ranging frequency. It typically has 2cm systematic error when range is less than 1m and statistical error of 2% for range smaller than 8m. Because of its high precision scanning, and wide-ranging range with adaptive scanning frequency it is well suited for this project work.

Encoder

Closed-loop operation of the robot demands the application of an encoder that measures the rotation of the shaft/motor and translates it into a series of pulses. Robot's states are feedback to the controller using 600 Pulse per revolution (PPR) rotary optical encoders attached to the individual motor shafts. It is an absolute encoder that provides 1200 counts per revolution.

Previously, we tested brush encoders, where brushes make contact with conductive seg-

Figure 3.19: YDLIDAR G4 model

ments on the moving surface and provide the measurement. However, the counts of such encoders are limited to six. Therefore, a high-precision rotary optical encoder (Model: E6C3-C) was used.

IMU

One of the most prevalent sensors in the navigation sector is the inertial measurement unit. It has a gyroscope and an accelerometer (sometimes also a magnetometer and rarely also a barometer). The first is in charge of measuring acceleration, while the second is in charge of measuring angular velocity. Because each of the measures is represented by a three-axis coordinate system, they combine to form a six-dimensional measurement time series stream. IMU Sensor (GY-87) with MPU-6050 chip was used in the robot.

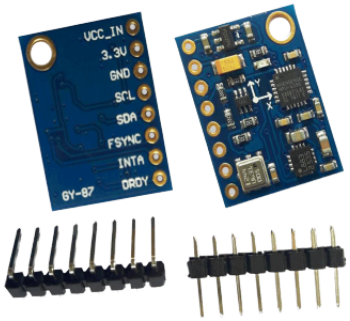


Figure 3.20: IMU Sensor GY-87

3.9.2 Controller

Raspberry Pi

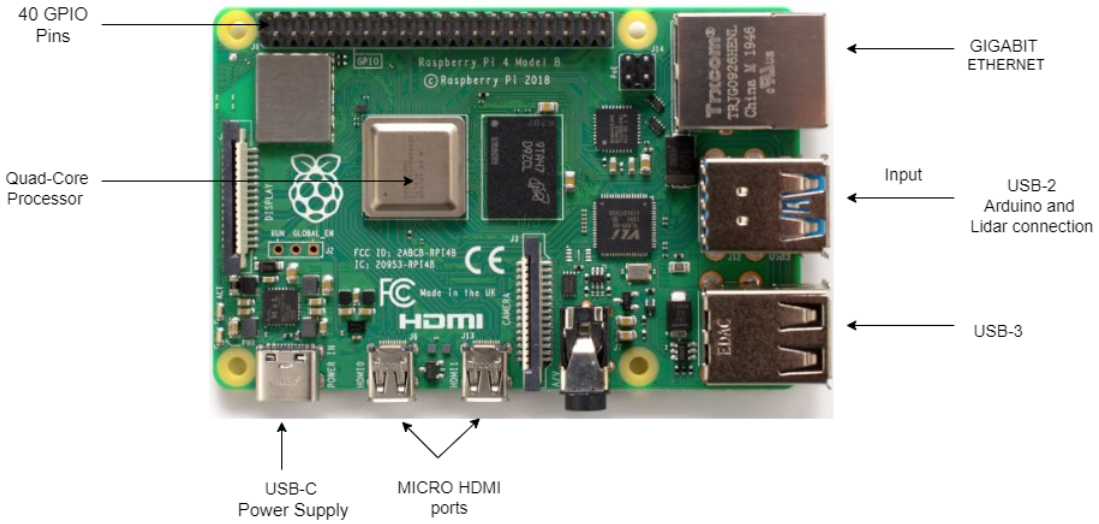


Figure 3.21: Raspberry Pi 4 Model B

Path planner algorithm is implemented on a single board computer, Raspberry Pi 4 Model B. It has a high-performance 64-bit Broadcom BCM2711 quad-core ARM Cortex-A72 processor having a speed 1.5GHz along with gigabit ethernet ports, a USD-C power supply, two USB-2 and USB-3, two Micro HDMI ports supporting two 4k displays, and 8 GB RAM. 5V DC can be supplied either from Type-C USB or via General Purpose Input/Output (GPIO) header. It consists of a standard 40-pin GPIO header that is fully backward-compatible with previous versions of Pi boards.

Arduino Mega

Interpretation of encoder data and transmission of Pulse Width Modulation signal to the motor driver demands Arduino Mega 2560. It is a microcontroller board equipped with At mega 2560 controller having more memory space and input/output pins than other Arduino boards.

Mega supports three types of communication protocol; serial, SPI, and I2C protocol. There are 4 hardware serial ports out of which we require 2 that are not available in other types of Arduino boards. The first pins of the serial port are pins 0 and 1 where

pin 0 is Rx and Pin 1 is Tx. Rx is used to receive data and Tx is to transmit serial data. The USB port is internally connected to these hardware serial pins therefore we cannot upload code in Arduino mega keeping the connection of any serial modules at these pins.

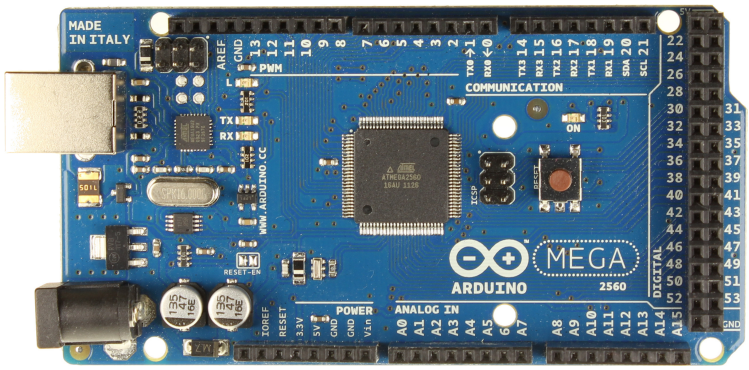


Figure 3.22: Arduino mega 2560 pinout

For quick response of encoder data, hardware interrupt pins 2, 3, 18 and 19 are used for encoders. Mega provides 4 hardware interrupt pins that interrupt the ongoing process when a certain signal is detected on the pins and executes some other code design to react to external stimulus. Interrupt pins are more reliable than coding on software for the same case. There are 13 PWM pins on board including interrupt pins among which pins 9 and 10 are used to transmit PWM signal to the Motor driver.

Arduino Shield

It is designed and manufactured in Robotics Club to house the connection between Arduino and other hardware interfaces like encoder, motor driver, and Bluetooth connection. Arduino mega consists of only one 5v pin but we require three 5V pins, one for connecting Bluetooth and rest for encoders. Therefore, 2 db9 connectors are attached to pin 9. One connector sends encoder data to Arduino and other is used to transmit PWM signal to motor driver.

Motor Driver

Dual-channel Cytron MDD20A was used to drive the motors. Operating between 6V to 30V, the Cytron MDD20A allows bidirectional control of two high-power brushed

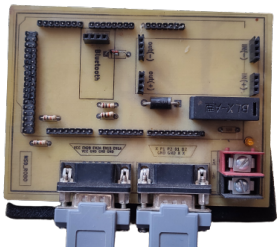


Figure 3.23: Arduino shield

DC motors. This motor driver can sustain 20Amp continuously for each motor without a heatsink due to its complete discrete NMOS H-Bridge architecture. PWM and DIR inputs can be used to control this motor driver. With input logic voltage range of 1.8V to 12V, the driver is compatible with a broad number of controllers.

The arduino library for Cytron Motor Drivers facilitated writing PWM signals directly

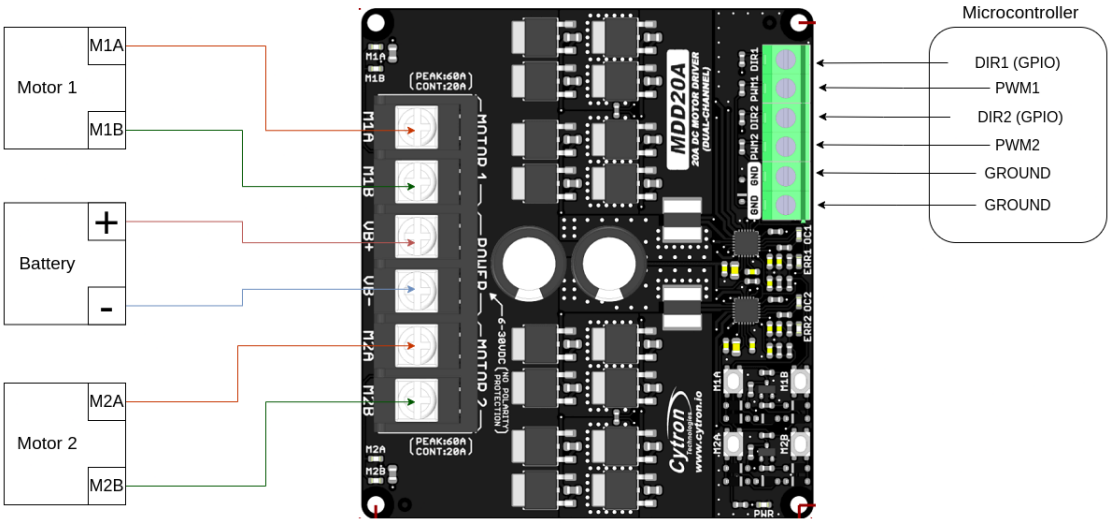


Figure 3.24: Block Diagram Representation of Motor Driver I/O

using the arduino. It accepts a value from 0 to 255 and the motor speed is set accordingly.

Table 3.1: Motor Driver Truth Table

PWMx	DIRx	Output A (MxA)	Output B (MxB)	Motor x
Low	-	Low	Low	Brake
High	Low	High	Low	Forward
High	High	Low	High	Backward

3.9.3 Actuator

Motor

The motor used in the robot is a 24 V DC hub motor.

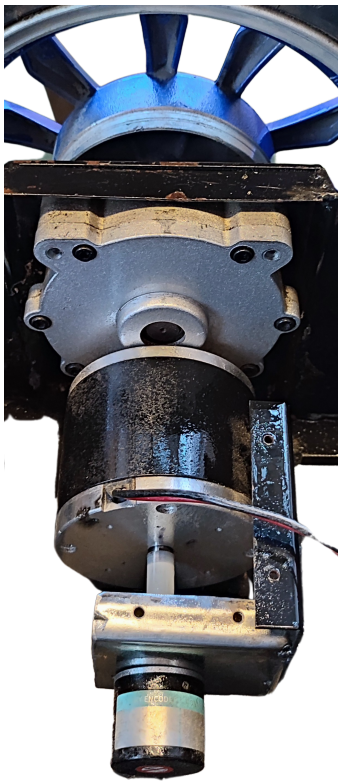


Figure 3.25: Motor Encoder and wheel assembly

3.9.4 Other components

Buck

It is a potentiometer that steps down the voltage of battery from 24V to operating voltage of controllers 5V. Doing so, both controllers Raspberry Pi and Arduino Mega can be powered by same battery.

Fuse

A fuse having current rating of 20A is used. If the motor draws more current than this rated value it melts and breaks the circuits. Since the motor is an inductive device, its characteristics vary with speed and loading. When there is higher resistance, it draws more current. However, the continuous current rating of the motor driver is 20A and

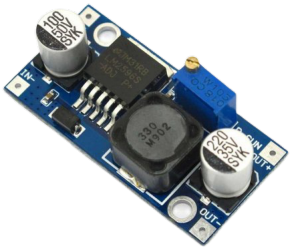


Figure 3.26: Buck

that of the motor is 6A in unloaded condition. Therefore, A fuse of rating current 20A limits the current drawn from the battery.

To enhance the safety further, a ceramic capacitor is used in the motor that prevents the back current flow through the motor.

Battery

A 24V and 6000mAh LiPo battery is used to power the robot. The battery is connected to the buck to meet the individual voltage requirement of each robot component.

Figure 3.27: Battery

3.10 Software Implementation and Architecture

The above mentioned hardwares come together and work simultaneously to make a fully functional robot. Figure 3.28 shows the basic working architecture of the robot.

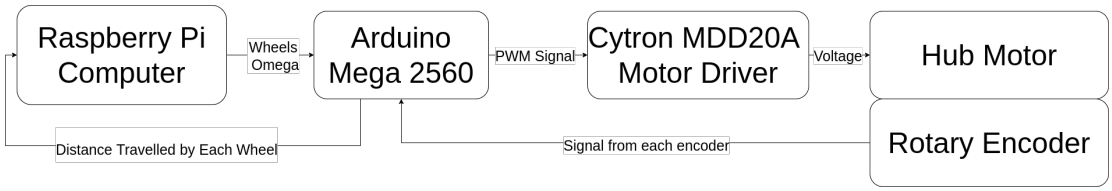


Figure 3.28: Robot Working Architecture

3.10.1 Arduino

Arduino mega sends control signal to actuators and processes sensor data. It reads the encoder data in the form of counts from hardware interrupt pins 2,3, and 18, 19 via serial communication. The data is then processed to give calculate the distance travelled by each wheels. In one revolution wheels travel 1.276m and encoder publishes 1200 counts, therefore the distance travelled by each wheel is given by Equation 3.37.

$$\text{Distance travelled by wheels} = 1.276 \times 0.5 \times \frac{count}{1200 \times 16} \tag{3.37}$$

Here, 16 is the gear box reduction.

3.10.2 Raspberry Pi

We have used Raspberry Pi 4 Model B with 8GB RAM as our main processor. The path planning algorithm with ROS architecture runs on the pi and the lower level control, i.e. generation of PWM signal with direction to motor driver is done by Arduino as mentioned in section 3.9.2. The main function of Raspberry pi is to MPC implementation with CasADi. For that it needs feedback of the location it is currently in. The feedback data is taken from the Arduino as shown in figure 3.28 through Serial communication. It then uses the data to compute the control signal and sends to the arduino, also through serial communication.

Now lets jump to the internal architecture and working of Pi. It has 4 cores and supports multi threading by architecture. Figure 3.30 shows the working architecture inside Raspberry Pi main working script. Multi-threaded programs can improve performance compared to traditional parallel programs that use multiple processes. Furthermore, improved performance can be obtained on multiprocessor systems using threads. Inter-thread communication is far more efficient and easier to use than inter-process communication. Because all threads within a process share the same address space, they need not use shared memory. In our case, we need to update the robot state in real-time and also detect the obstacle in real time. If we were to use serial processes, then there would be lag in real-time processes. So the use of multi threading technique enabled us to plan the trajectory in real-time with minimum lag.

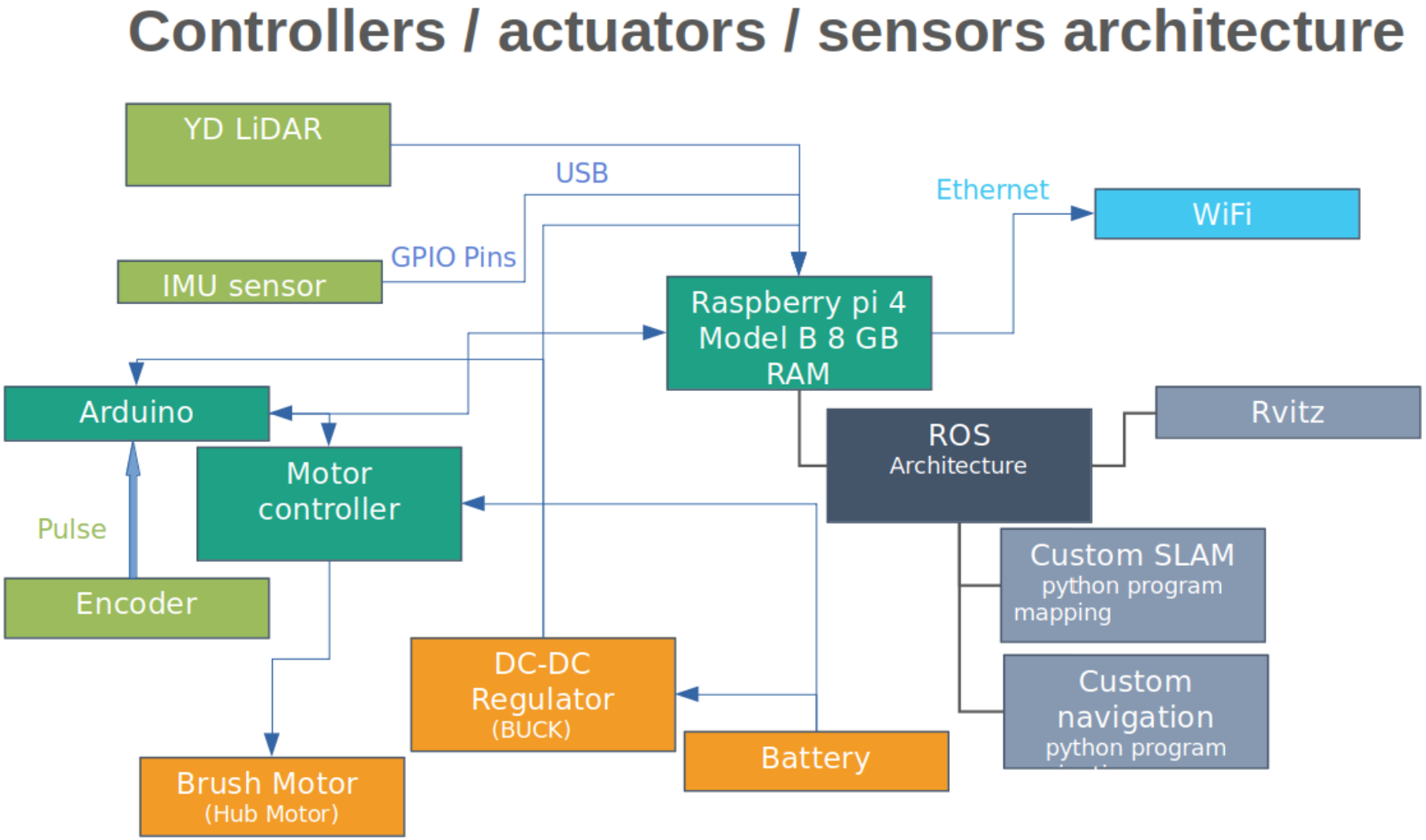


Figure 3.29: Detailed Robot Architecture

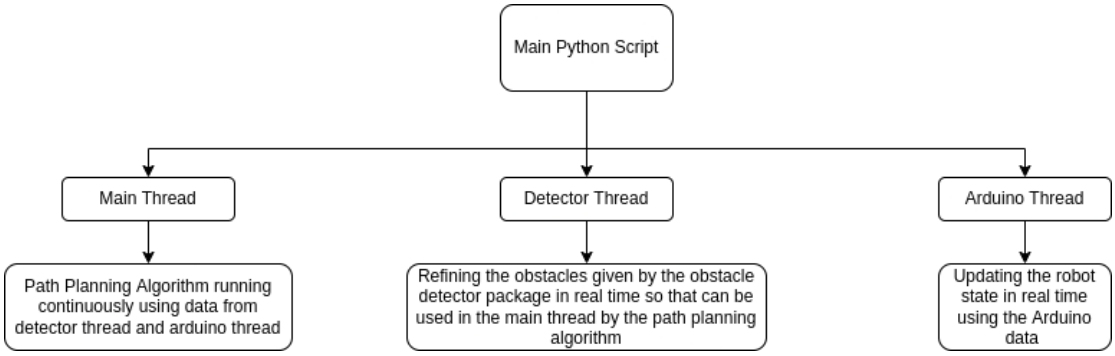


Figure 3.30: Working Structure in Raspberry Pi

So, a python script was written which works on multiple threads as shown in figure 3.30. The threads are named as main thread, detector thread and arduino thread. Before talking about the main thread, let us talk about detector and arduino thread.

Detector Thread

This thread subscribes the data from obstacle detector package in the first place. The YD lidar package publishes the scanned data to the /scan topic which is running in the separate thread of ROS. It continuously publishes the data and the published data is then subscribed by the obstacle detector package. The working of the obstacle detector package is explained in 3.3. The obstacle's locations are published in class of classes. So the retrieving of data from those locations is done in this thread. The retrived data is the location of obstacles from the lidar frame of reference. This means they are in local frame of reference and are to be converted to the global frame of reference. The data are then converted to global frame of reference and are stored as the global variable in real time so that it can be used by the main thread during the path planning process with the updated obstacles locations.

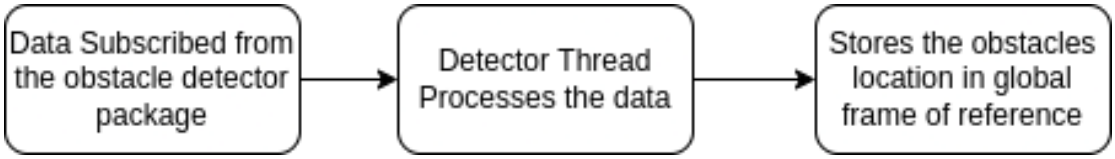


Figure 3.31: Working of detector thread

Arduino Thread

The main function of the arduino thread is to receive the data coming from the arduino through serial communication and then process it. The incoming data is in utf-8 encoding.UTF-8 is a variable-width character encoding used for electronic communication. So the first process is reading the characters until the end of line, decoding it and then striping and splitting the string to get the wheels information. The data are then transformed to the robot current location in global frame of reference and are published to \odom topic. These values are updating in real time.

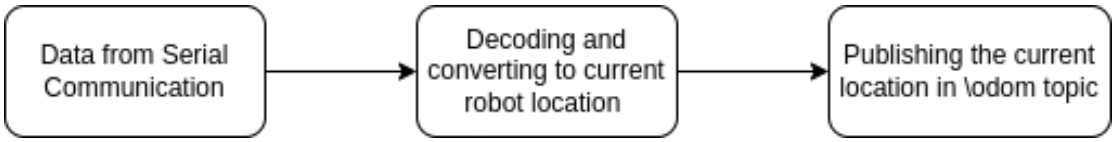


Figure 3.32: Working of arduino thread

IMU thread

IMU thread is basically a ROS node working in background. It is not written in the main script. This node senses the sensor data from the IMU pins and perform IMU kinematics to get the position, velocity and acceleration of the robot. Thus obtained data is being continuously published in \imu topic. This data is later subscribed by EKF package.

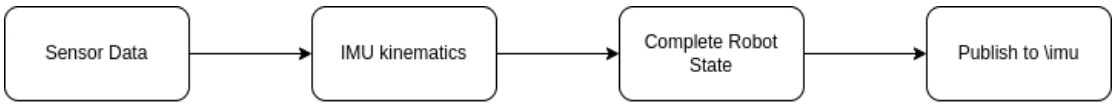


Figure 3.33: Working of IMU node

Main thread

The main algorithm runs in this thread. This thread runs until the robot reaches the final location and then the whole system stops. That means, it will shut main working script and also the IMU ROS node.

We are using CasADi as our solver working environment. We can define paramaters so that they can be updated later while running the MPC loop. The first step is to initialise the problem with the current robot state. This is done by subscribing the

\odom_combined topic published by the EKF package and assigning the state parameter. If the EKF node is not running then the thread directly uses the data from arduino thread. Also the obstacles parameters are assigned with the current obstacle location usind the data from the Detector thread. Then the solution is initialised and then solved. After solving, we get two control inputs, heading velocity and the rate of change of orientation. These values are then converted to wheels omega using kinematics and sent to arduino through serial communication. Then again it subscribes the current robot state and solve and send to arduino. In this mannner, the loop continues.

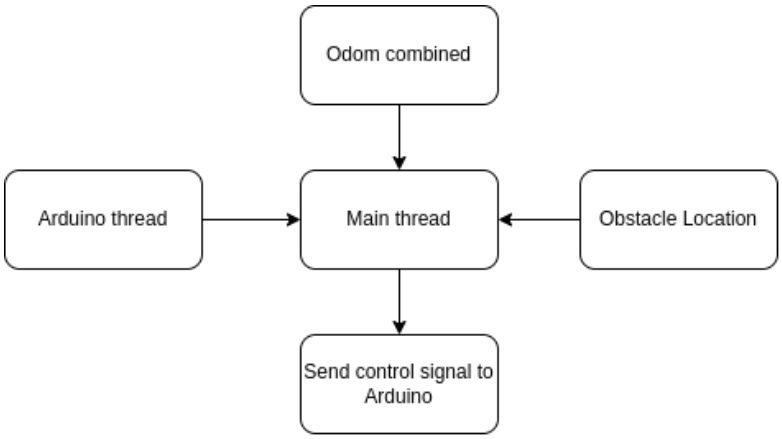


Figure 3.34: Main thread data architecture

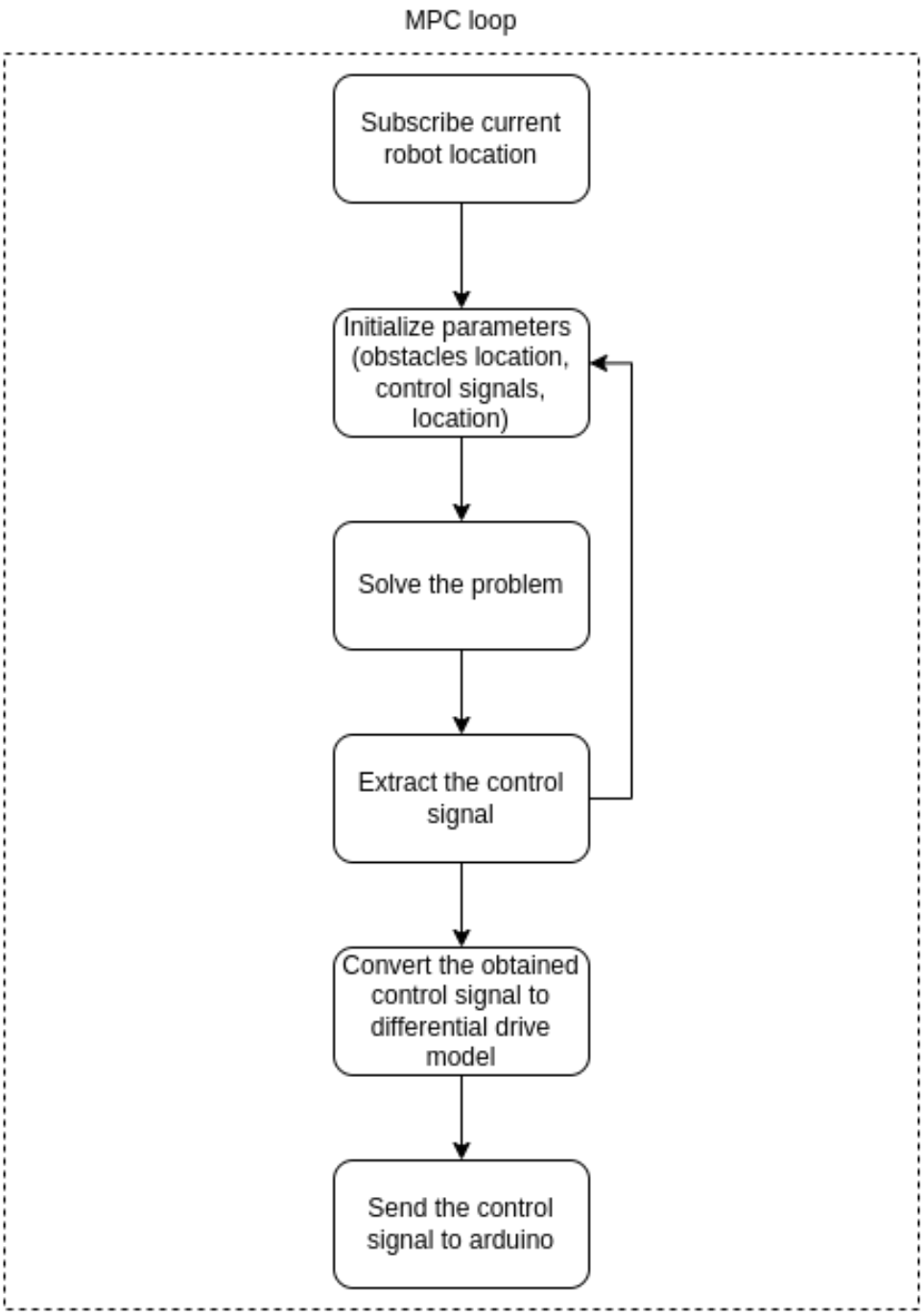


Figure 3.35: Working of main thread

CHAPTER FOUR : RESULT AND DISCUSSION

4.1 Trajectory Simulations

The methodology discussed in chapter 3 was carried out for simulation in Python. The following tables and equation summarize the problem that is being considered:

Table 4.1: Initial and final state of formulated problem

States	x	y	θ
Initial	0	0	0
Final	10	10	π

Table 4.2: Specifications of modelled obstacles

Obstacle No.	x coordinate	y coordinate	radius	Type
1	3	5	0.5	Static, Circular
2	8	3	0.5	Static, Circular
3	7	7	0.5	Static, Circular

$$Minimize: J = \sum_{k=1}^{N} \left\{ [x_k - 10]^2 + (y_k - 10)^2 + (\theta_k - \pi)^2 + 0.5 * [v_k]^2 + 0.5 * [\omega_k]^2 + r(x_k, y_k) \right\}. \tag{4.1}$$

Subject to:

$X_1 = X_0$

for k = 1:N

$X_{k+1} = X_k + f(X_k, U_k) * \Delta t$

$0 \leq x_k \leq 12$

$0 \leq y_k \leq 12$

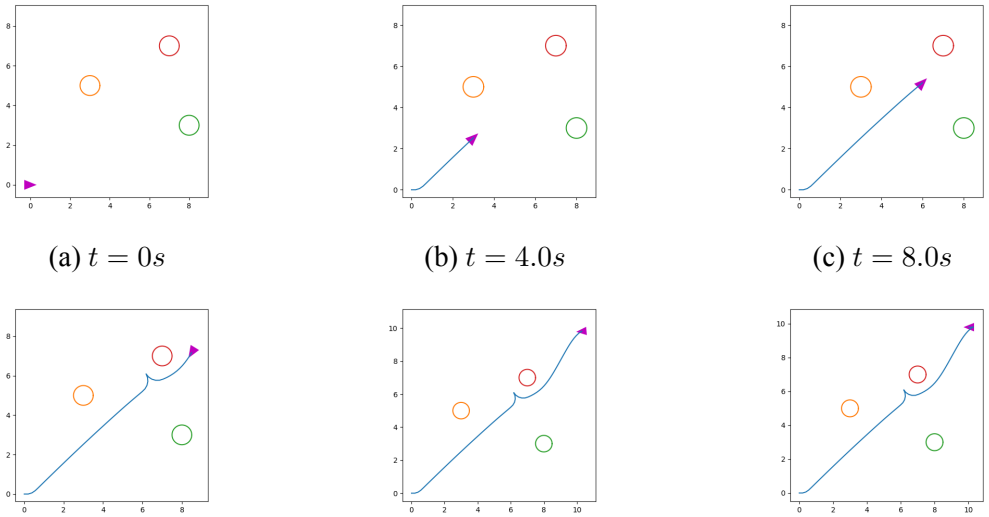
$-1 \leq v_k \leq 1$

$-1.5 \leq \omega_k \leq 1.5$

$h(x_k, y_k) > 0$

Figure 4.1 shows the path generated by the algorithm. It takes around 0.5 seconds to

plan the trajectory for the 20 seconds time interval. Figure 4.2 show how the state of the robot is planned to change throughout the 20 seconds. The generated trajectory tend to avoid the obstacles and converge near to the prescribed final state. The open loop control signal required to describe the path is depicted in figure 4.3. The control signals are well within their upper and lower bounds. Noticeably, the trajectory is generated such that the value of heading velocity hits $\pm 1 m/s$ most of the time and finally both the input signals settle to 0.



(a) $t = 0s$ (b) $t = 4.0s$ (c) $t = 8.0s$

(d) $t = 12.0s$ (e) $t = 16.0s$ (f) $t = 20.0s$

Figure 4.1: State of robot at different time stamps (Open Loop Solution 0.55 seconds)

Also, receding horizon control scheme was applied to navigate the robot from initial to final position. The finite path-planning horizon (τ) was chosen to be 2 seconds. The 2 second timestamp is discretized into 10 subintervals of 0.2 seconds (i.e N=10 and Δt=0.2 seconds). The following sequence is followed by the robot:

a. Generate the trajectory of next τ seconds.

b. Follow the generated trajectory for next Δt seconds.

c. Update the current state.

d. Repeat.

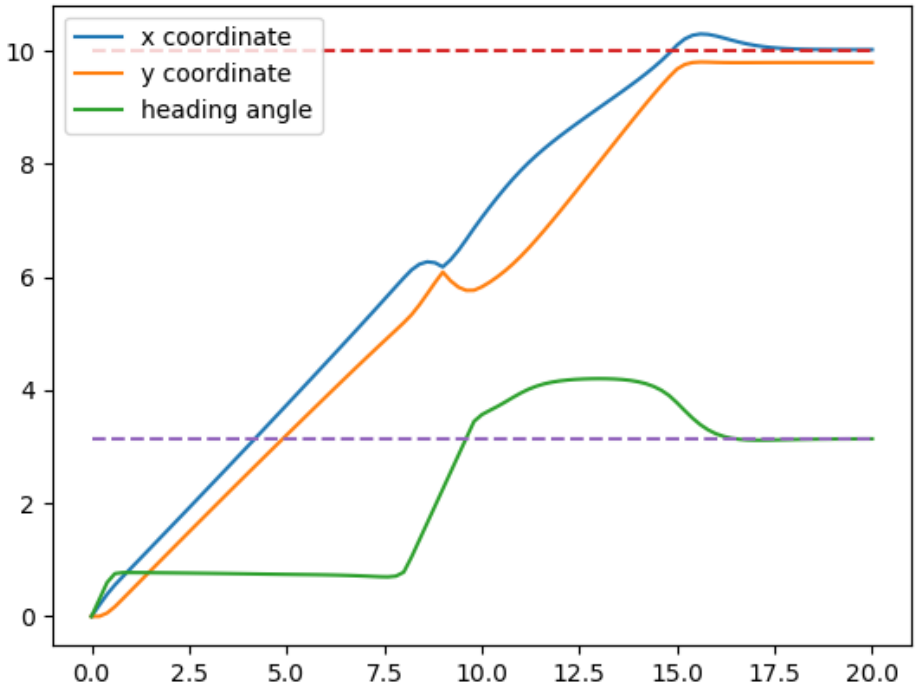


Figure 4.2: Variation of robot states with time

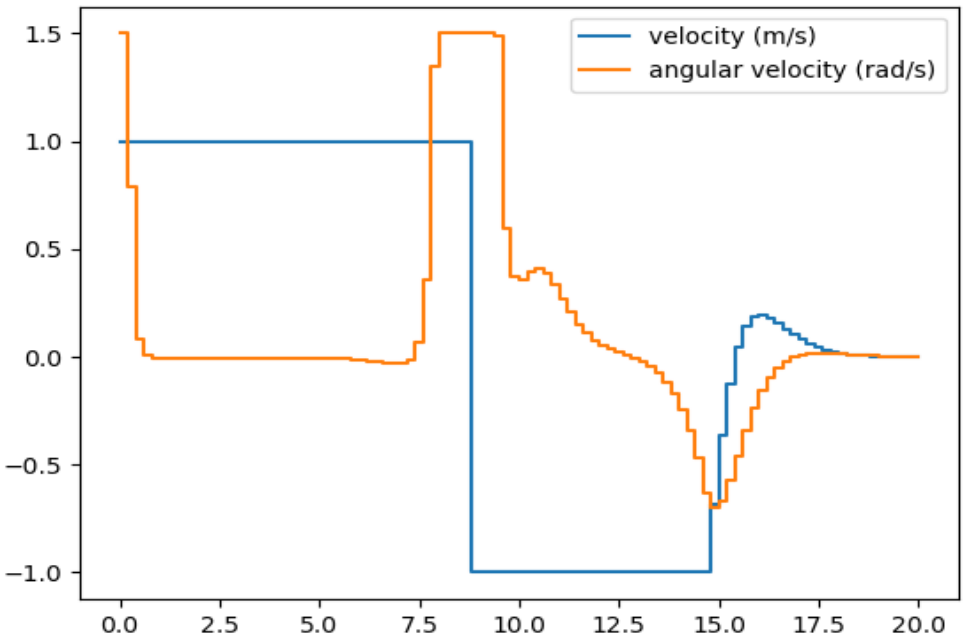
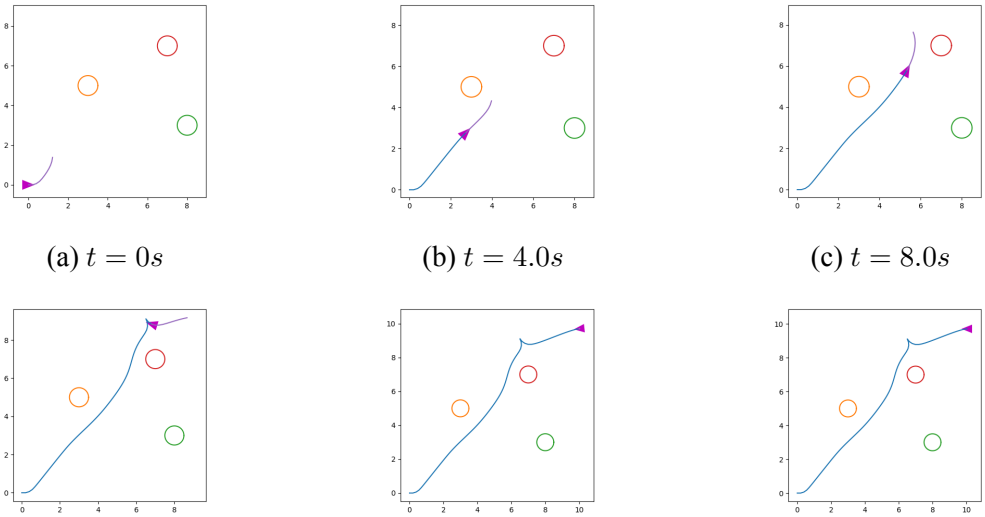


Figure 4.3: Variation of control signals for unicycle robot with time

Figure 4.4 shows the state of robot at intermediate time stamps. Two different colours

are used to indicate the traversed path and the planned path. Similarly, Figure 4.6 and Figure 4.7 illustrate how the robot states and control signal change throughout the 20 seconds travel. The control signals for unicycle can be manipulated to calculate the control inputs for differential drive robot as explained in the previous sections. Figure 4.8 shows the discrete control signals in terms of left wheel velocity and right wheel velocity of the differential drive robot. The cumulative computation time during the simulation was just around 1.2 seconds.



(a) $t = 0s$ (b) $t = 4.0s$ (c) $t = 8.0s$

(d) $t = 12.0s$ (e) $t = 16.0s$ (f) $t = 20.0s$

Figure 4.4: State of robot at different time stamps

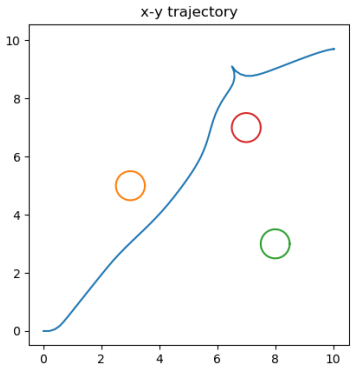


Figure 4.5: Path traced by the robot

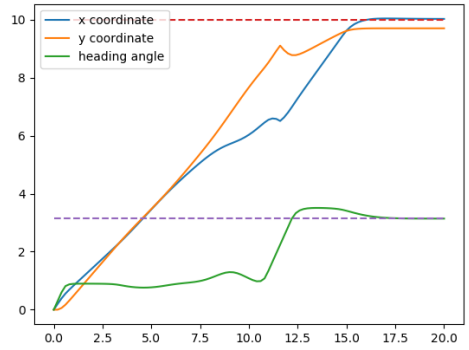


Figure 4.6: Variation of robot states with time

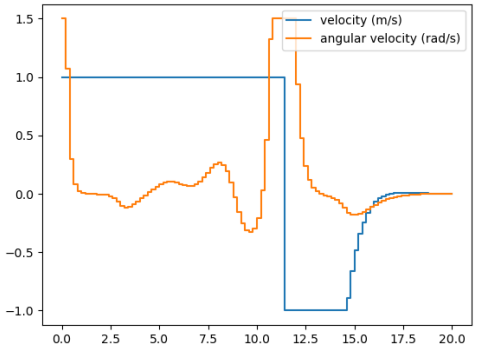


Figure 4.7: Variation of control signals for unicycle robot with time

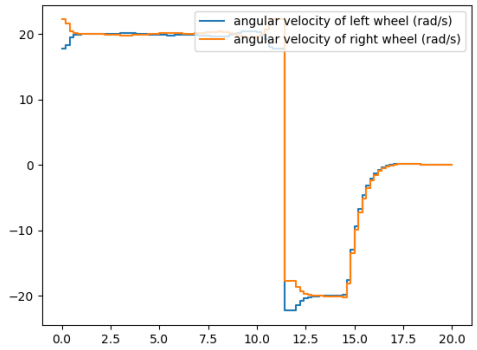
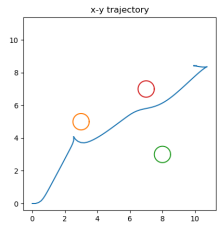


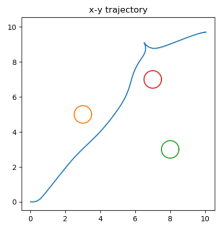
Figure 4.8: Variation of control signals for differential drive robot with time

Also, the path traced by the robot for different lengths of path planning horizon is plotted in Figure 4.9. Figure 4.10 shows how states of the robot change with time as the horizon length is changed. When the value of τ is kept 1 second, the robot fails to reach the goal

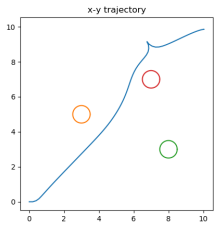
y-position. The error is negligible and change is minimal on further increasing the value after $\tau = 2seconds$. However, the computational cost increases drastically on further increasing τ. This is illustrated by figure 4.11. Paths traced by the robot in other map configurations and different end conditions are tabulated in table 4.3.



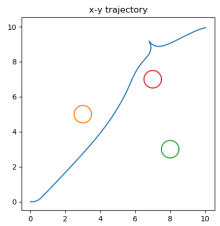
(a) $\tau = 1.0s$



(b) $\tau = 2.0s$

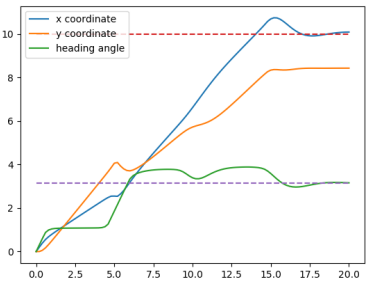


(c) $\tau = 4.0s$

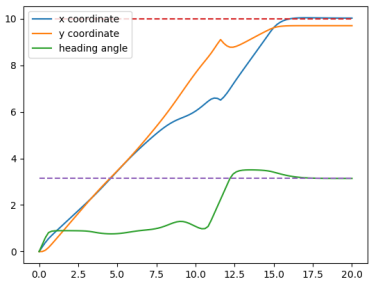


(d) $\tau = 8.0s$

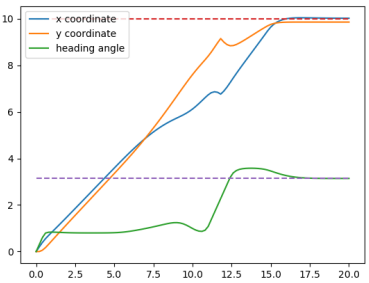
Figure 4.9: Path traced by robot for different values of planning horizon



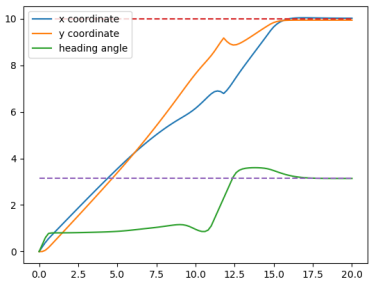
(a) $\tau = 1.0s$



(b) $\tau = 2.0s$



(c) $\tau = 4.0s$



(d) $\tau = 8.0s$

Figure 4.10: States of robot at different time stamps for different values of planning horizon

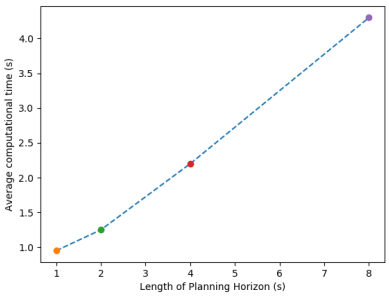
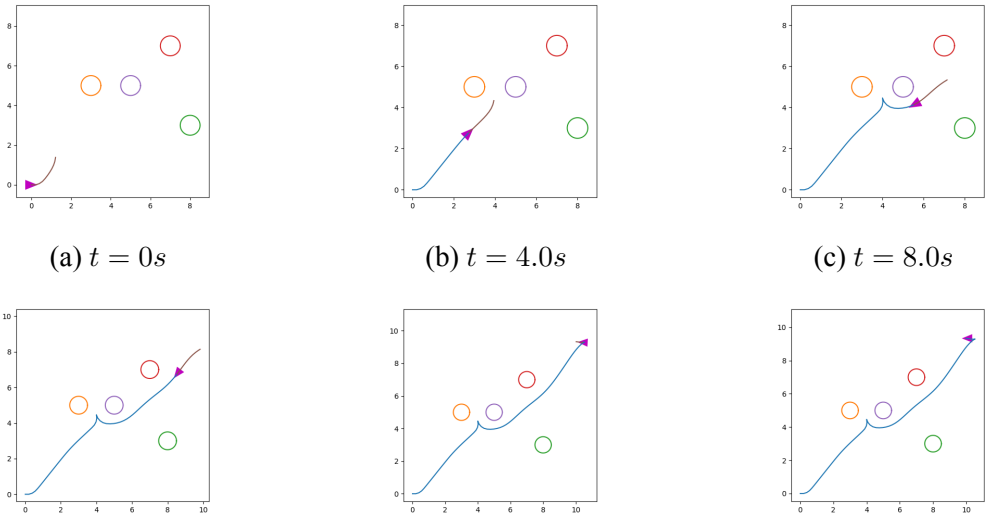


Figure 4.11: Computational time required for different lengths of planning horizon

Table 4.3: Paths traced by robot in different test cases (τ= 2sec, Δt= 0.2sec)

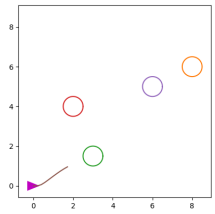
SN	Initial State	Final State	No. of Obstacles	Cumulative Computation Time	Figure
1	[0,0,0]	[10,10,π]	4	1.29 seconds	4.12
2	[0,0,0]	[10,8,0]	4	1.53 seconds	4.13
3	[1,3,π]	[10,10,0]	5	1.46 seconds	4.14



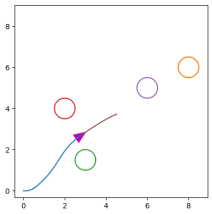
(a) $t = 0s$ (b) $t = 4.0s$ (c) $t = 8.0s$

(d) $t = 12.0s$ (e) $t = 16.0s$ (f) $t = 20.0s$

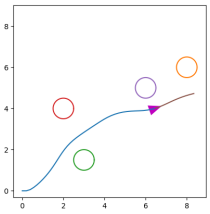
Figure 4.12: State of robot at different time stamps (Table 4.3.1)



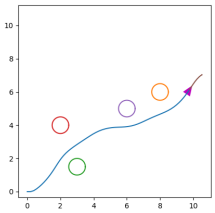
(a) $t = 0s$



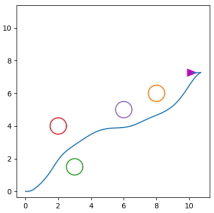
(b) $t = 4.0s$



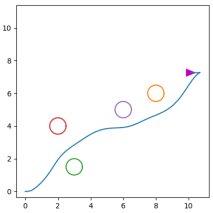
(c) $t = 8.0s$



(d) $t = 12.0s$

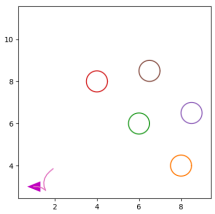


(e) $t = 16.0s$

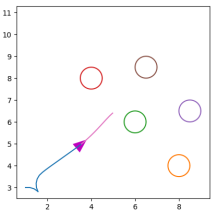


(f) $t = 20.0s$

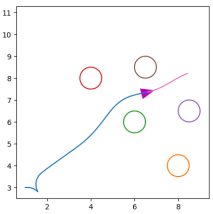
Figure 4.13: State of robot at different time stamps (Table 4.3.2)



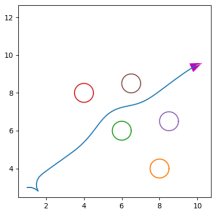
(a) $t = 0s$



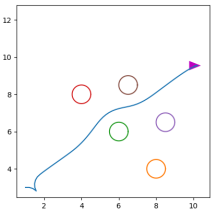
(b) $t = 4.0s$



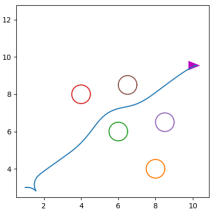
(c) $t = 8.0s$



(d) $t = 12.0s$



(e) $t = 16.0s$



(f) $t = 20.0s$

Figure 4.14: State of robot at different time stamps (Table 4.3.3)

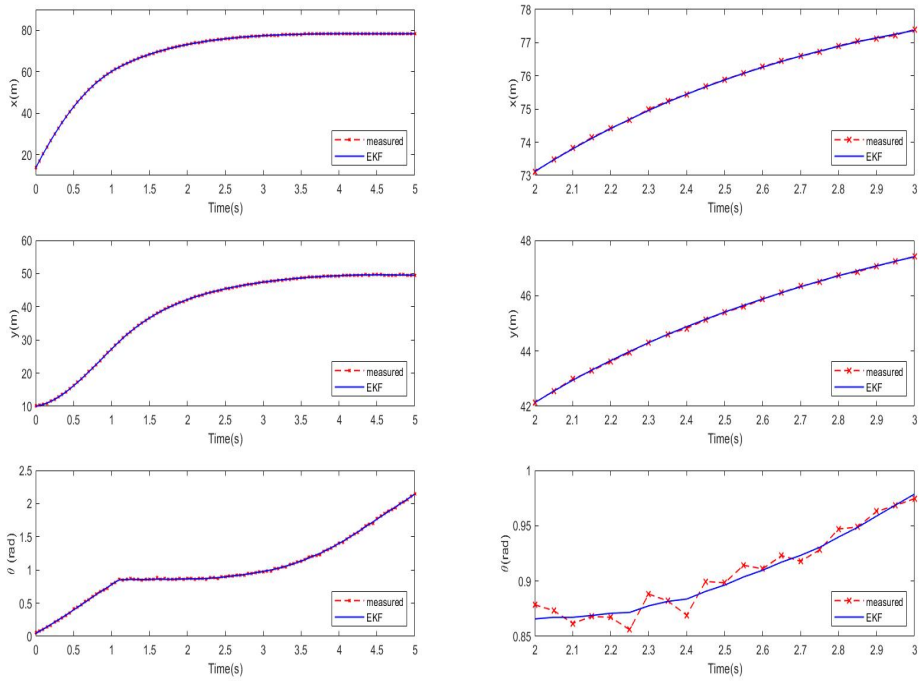


Figure 4.15: Extended Kalman Filter results for unicycle model. Right: Magnified plot

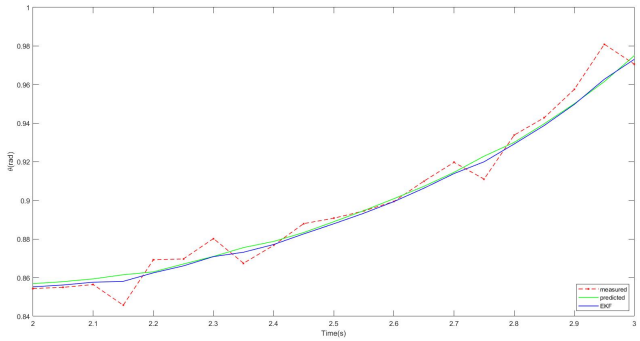
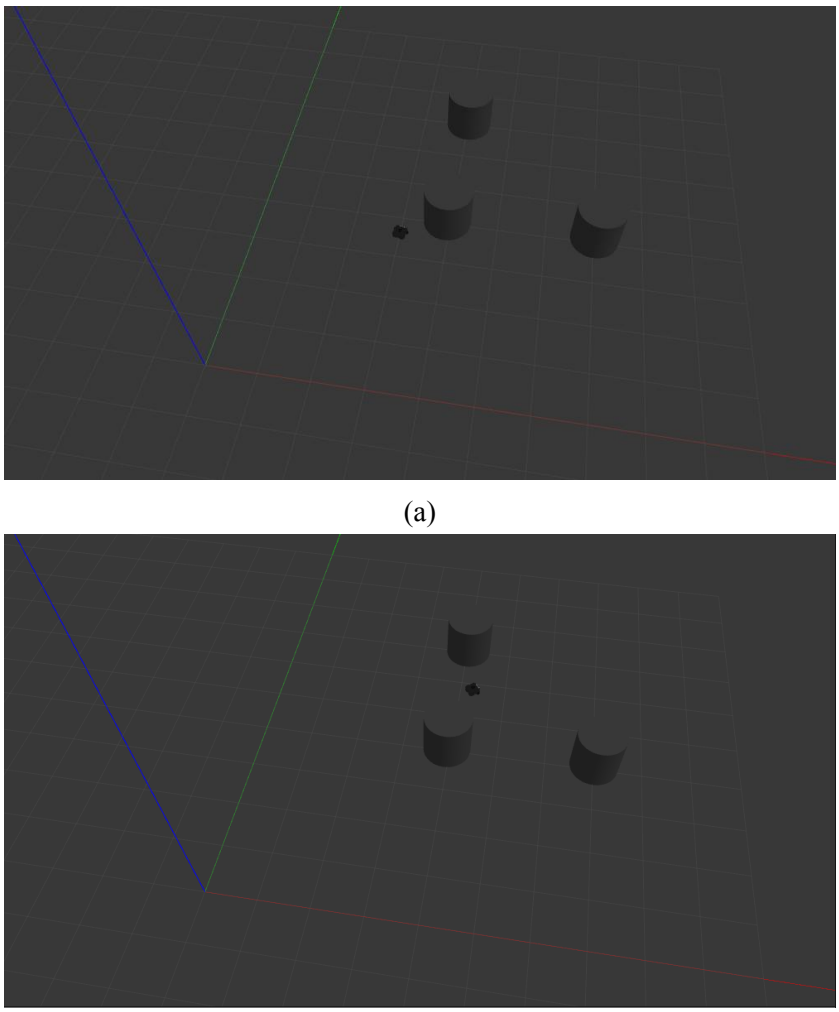


Figure 4.16: Estimation of orientation by sensor measurement and system dynamics

Figure 4.15 shows the result of the designed filter to estimate the states. The filter is applied to a unicycle model having a different problem formulation whose states vary with time as indicated by charts on the left. The chart on the right is the magnified view. The filter has smoothened the noisy sensor measurement.

The working of the EKF can be visualized clearly from the Figure 4.16. The filter has fused the information of sensor measurement and predicted state from system dynamics, and given their weighted average.

4.2 ROS/Gazebo Simulation



(a)

(b)

Figure 4.17: Gazebo Simulation of Turtlebot3 Waffle

The algorithms simulated in the previous section is now tested against the Turtlebot3 Waffle robot in Gazebo empty world environment with cylindrical obstacles. Figure 4.17 shows two instances of the Waffle robot navigating in the environment. As it proceeds towards the goal state defined as $(10, 10, \pi)$, the intermediate states are recorded and plotted in Figures 4.18 and 4.19. The path traced by the robot in x-y coordinates is shown in the figure 4.18 and the variation of all three states (x, y, θ) is graphed in he Figure 4.19. Although the trajectory planner successfully leads the robot to its desired final state avoiding the obstacles, there seem to be some problems in stabilizing the robot near the goal. This is due to the inertia of the robot that the planner does not account; the robot keeps turning in one direction unless explicitly forced to stop. It takes

3 redundant turns before finally reaching the goal state.

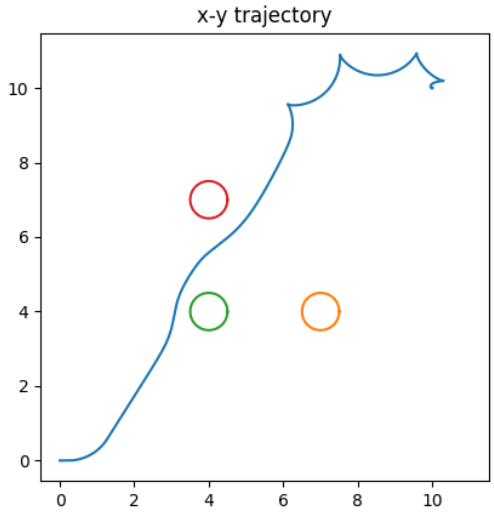


Figure 4.18: Path traced by the Turtlebot3 Waffle

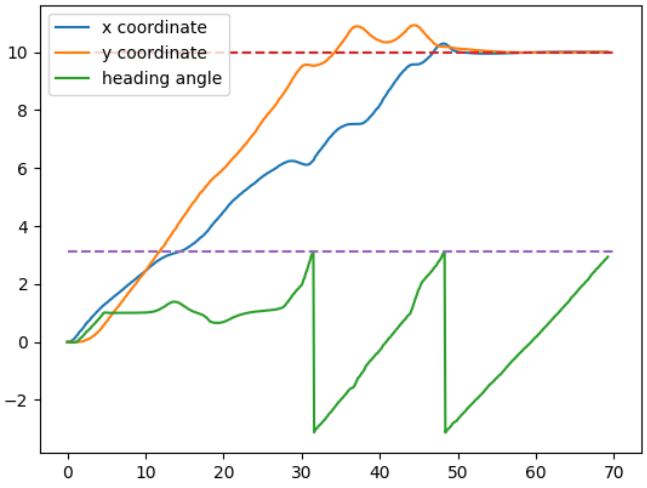


Figure 4.19: Variation of Turtlebot3 Waffle robot states with time

4.3 Robot Testing

A robot frame was manufactured and all the hardware components were assembled into a robot whose image is attached in Figure 4.20. The robot parts are as described in the preceding chapters of the report. The robot is subjected to different test conditions tabulated in Table 4.4.

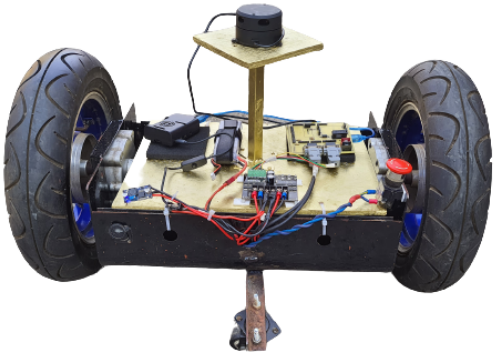


Figure 4.20: Robot with all the components assembled

Table 4.4: Paths traced by robot in different test cases (τ= 2sec, Δt= 0.2sec)

SN	Initial State	Final State	Obstacles	Figure
1	[0,0,0]	[-5,0,0]	(-1,0), (-2,0), (-3,0)	4.21
2	[0,0,0]	[10,10,π]	(2,2)	4.22
3	[0,0,0]	[10,10,0]	(2,2), (4,4)	4.23

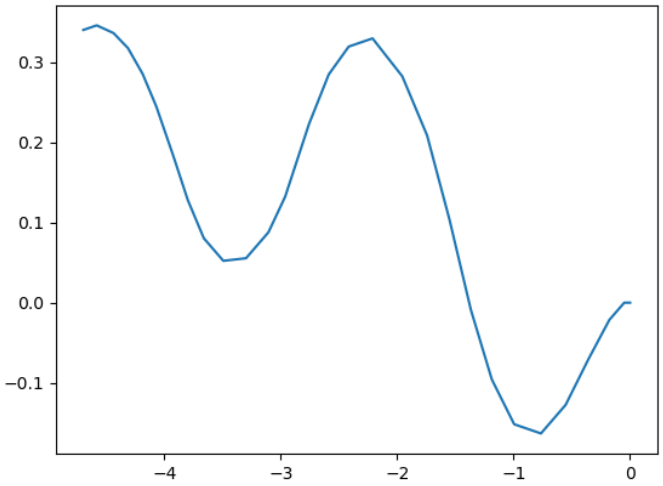


Figure 4.21: Path traced by the robot from (0,0,0) to (-5,0,0) (Table 4.4.1)

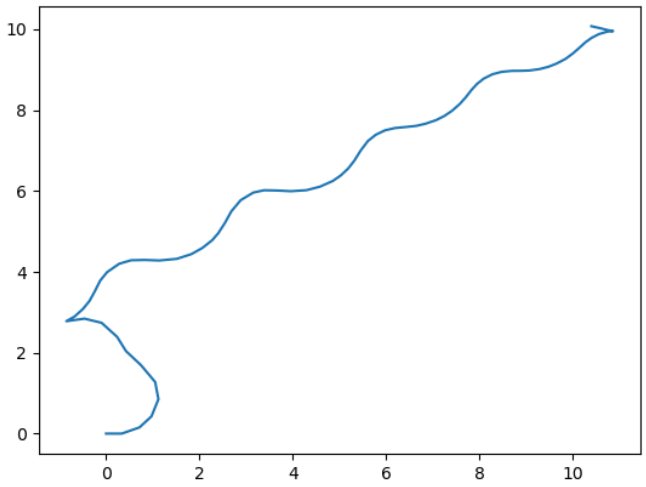


Figure 4.22: Path traced by the robot from (0,0,0) to (10,10,π) (Table 4.4.2)

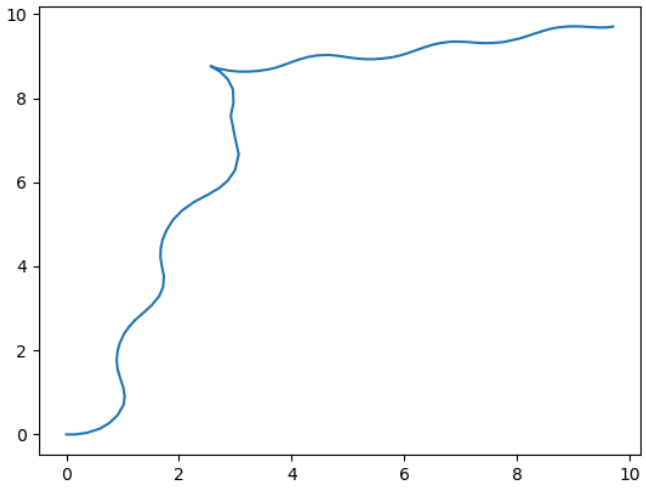


Figure 4.23: Path traced by the robot from (0,0,0) to (10,10,0) (Table 4.4.3)

CHAPTER FIVE : CONCLUSION AND RECOMMENDATIONS

5.1 Conclusions

Robots used in simulation and physical implementation were modelled based on their kinematic and dynamic properties. Of the modelled robots, Kinematic model of differential drive robot was used for simulation and physical implementation.

Trajectory generation using the optimal control formulations was performed using the methodologies explained above. The formulated optimal trajectory generation problem was transcribed as non-linear programming problem with above mentioned costs and constraints and solved using a numerical technique to obtain optimal control signals.

The interior point solver algorithm was used to solve the nonlinear optimization problem with prediction horizon of 2 seconds discretized into 10 parts each of time step 0.2 seconds. Prediction horizon of 2 seconds was found to be a suitable trade-off between computational cost and the steady state error. Model Predictive Control employed in solving the optimization problem was found to be very effective and efficient. The planner was successful in generating a trajectory from initial states to goal states, while also satisfying all the imposed constraints and the control inputs remaining within the prescribed limits. Simulation of robot in Gazebo environment was performed using ROS implementation. The *Waffle* variant of Turtlebot3 was used for testing the results of the trajectory planner. The path planner was able to lead the robot to goal states successfully avoiding all the obstacles with slight overshooting of target which was later corrected with additional control inputs.

Hardware implementation for validation of the proposed algorithm was performed by designing and developing a differential drive robot, with appropriate sensing units, controllers and actuators. While testing on real environment with real obstacles, the robot was able to navigate to its final position and orientation while avoiding obstacles with slight error. Obstacle detection algorithm was also fairly successful with some issues in bright environment.

5.2 Recommendation

The following recommendations can be suggested for future works:

1. Optimal control algorithm can be applied to robots with other means of maneuverability and handling.
2. Research on the use of other sophisticated solvers to improve the computational efficiency of the algorithm.
3. Objective function can be redesigned to include further optimization parameters.
4. Obstacle detection and modelling algorithm can be improved to reduce the obstacle processing time and include more geometric shapes of obstacles.
5. Castor wheel orientation can be adjusted to reduce the slippage of main driving wheel.

REFERENCES

Ackerman, E. (2013). Turtlebot inventors tell us everything about the robot. *IEEE Spectrum*, *26*, 2013.

Adhikari, M. P., & Ruiter, A. H. d. (2020). Real-time autonomous obstacle avoidance for fixed-wing uavs using a dynamic model. *Journal of Aerospace Engineering*, *33*(4), 04020027.

Amsters, R., & Slaets, P. (2019). Turtlebot 3 as a robotics education platform. In *International conference on robotics in education (rie)* (pp. 170–181).

Andersson, J. (2013). *A General-Purpose Software Framework for Dynamic Optimization* (PhD thesis). Arenberg Doctoral School, KU Leuven, Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium.

Andreas Wachter, L. T. B. (2005). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, *106*.

Arbo, M. H., Grøtli, E. I., & Gravdahl, J. T. (2019). Casclik: Casadi-based closed-loop inverse kinematics. *arXiv preprint arXiv:1901.06713*.

Arora, J. (2004). *Introduction to optimum design*. Elsevier.

Betts, J. T. (1998). Survey of numerical methods for trajectory optimization. *Journal of guidance, control, and dynamics*, *21*(2), 193–207.

Bucsuházy, K., Matuchová, E., Zvala, R., Moravcová, P., Kostíková, M., & Mikulec, R. (2020). Human factors contributing to the road traffic accident occurrence. *Transportation research procedia*, *45*, 555–561.

Cusumano, M. A. (2020). Self-driving vehicle technology: progress and promises. *Communications of the ACM*, *63*(10), 20–22.

Diamond, S., & Boyd, S. (2016). Cvxpy: A python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research*, *17*(1), 2909–2913.

Eliwa, M., Adham, A., Sami, I., & Eldeeb, M. (2017). A critical comparison between fast and hector slam algorithms. *REST Journal on Emerging trends in Modelling and Manufacturing*, *3*(2), 44–49.

Filipenko, M., & Afanasyev, I. (2018). Comparison of various slam systems for mobile robot in an indoor environment. In *2018 international conference on intelligent systems (is)* (pp. 400–407).

Frasch, S. S. D. M., Janick V. (2015). A parallel quadratic programming method for dynamic optimization problems. *Mathematical Programming Computation*, *7*.

Gauss, K. F. (1963). Theory of the motion of the heavenly bodies moving about the sun in conic section. *New York: Dover*.

G. Gim, P. N. (1990). An analytical model of pneumatic tires for vehicle dynamic simulations: Part 1.pure slips. *International Journal of Vehicle Design, Vol. 11, No. 6, 589-618*.

G. Gim, P. N. (1991). An analytical model of pneumatic tires for vehicle dynamic simulations: Part 1.comprehensive slips. *International Journal of Vehicle Design, Vol. 12, No. 1, 19-39*.

Gill, P. E., Murray, W., & Saunders, M. A. (2005). Snopt: An sqp algorithm for large-scale constrained optimization. *SIAM review*, *47*(1), 99–131.

Guizzo, E., & Ackerman, E. (2017). The turtlebot3 teacher [resources_hands on]. *IEEE Spectrum*, *54*(8), 19–20.

Hardy, J., & Campbell, M. (2013). Contingency planning over probabilistic obstacle predictions for autonomous road vehicles. *IEEE Transactions on Robotics*, *29*(4), 913–929.

H.Pacejka. (2012). *Tire and vehicle dynamics*. Elsevier.

Jaroszek, P., & Trojnacki, M. (2015). Localization of the wheeled mobile robot based on multi-sensor data fusion. *Journal of Automation Mobile Robotics and Intelligent Systems*, *9*.

Joseph, L. (2018). *Robot operating system (ros) for absolute beginners*. Springer.

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems.

Kelly, M. P. (2017). Transcription methods for trajectory optimization: a beginners tutorial. *arXiv preprint arXiv:1707.00284*.

Koch, O., & Weinmüller, E. (2003). The convergence of shooting methods for singular boundary value problems. *Mathematics of computation*, *72*(241), 289–305.

Kohlbrecher, S., Meyer, J., von Stryk, O., & Klingauf, U. (2011, November). A flexible and scalable slam system with full 3d motion estimation. In *Proc. ieee international symposium on safety, security and rescue robotics (ssrr)*.

Kushleyev, A., & Likhachev, M. (2009). Time-bounded lattice for efficient planning in dynamic environments. In *2009 ieee international conference on robotics and automation* (pp. 1662–1668).

Leyffer, S., & Mahajan, A. (2010). Nonlinear constrained optimization: methods and software. *Argonee National Laboratory, Argonne, Illinois*, *60439*.

Lynch, K. M., & Park, F. C. (2017). *Modern robotics*. Cambridge University Press.

Lynch M., K., & C., P. F. (2013). *Modern robotics mechanics, planning, and control*. Cambridge University Press.

Madås, D., Nosratinia, M., Keshavarz, M., Sundström, P., Philippsen, R., Eidehall, A., & Dahlén, K.-M. (2013). On path planning methods for automotive collision avoidance. In *2013 ieee intelligent vehicles symposium (iv)* (pp. 931–937).

Meriem, M., Bouchra, C., Abdelaziz, B., Jamal, S. O. B., Nazha, C., et al. (2016). Study of state estimation using weighted-least-squares method (wls). In *2016 international conference on electrical sciences and technologies in maghreb (cistem)* (pp. 1–5).

Müller, M. A., & Worthmann, K. (2017). On quadratic stage costs for mobile robots in model predictive control. *PAMM*, *17*(1), 825–826.

Noga, S. (2006). Kinematics and dynamics of some selected two-wheeled mobile

robots. *Archives of Civil and Mechanical Engineering*, *6*(3).

Polack, P., Altché, F., Novel, B., & de La Fortelle, A. (2017, 06). The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles? In (p. 812-818). doi: 10.1109/IVS.2017.7995816

Przybyla, M. (2017). Detection and tracking of 2d geometric obstacles from lrf data. *2017 11th International Workshop on Robot Motion and Control (RoMoCo)*, 135-141.

Purwin, O. (2008). Real-time trajectory generation and control for autonomous vehicles.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., … others (2009). Ros: an open-source robot operating system. In *Icra workshop on open source software* (Vol. 3, p. 5).

Robo-rats locomotion: Odometry. (2001). Retrieved from `https://groups.csail.mit.edu/drl/courses/cs54-2001s/odometry.html`

Rubio, V. F., Francisco, & Albert, l. (2019). A review of mobile robots: Concepts, methods, theoritical framework, and application. *International Journal of Advanced Robotic Systems*.

Shalev-Shwartz, S., Ben-Zrihem, N., Cohen, A., & Shashua, A. (2016). Long-term planning by short-term prediction. *arXiv preprint arXiv:1602.01580*.

Subchan, S. S. (2011). A direct multiple shooting method for missile trajectory optimization with the terminal bunt manoeuvre. *IPTEK The Journal for Technology and Science*, *22*(3).

Thrun, S. (2007). Simultaneous localization and mapping. In *Robotics and cognitive approaches to spatial mapping* (pp. 13–41). Springer.

Tong, L. (2012). An approach for vehicle state estimation using extended kalman filter. In *International computer science conference* (pp. 56–63).

Wan, E. A., & Van Der Merwe, R. (2000). The unscented kalman filter for nonlinear estimation. In *Proceedings of the ieee 2000 adaptive systems for signal*

processing, communications, and control symposium (cat. no. 00ex373) (pp. 153–158).

Worthmann, K., Mehrez, M. W., Zanon, M., Mann, G. K., Gosine, R. G., & Diehl, M. (2015). Model predictive control of nonholonomic mobile robots without stabilizing constraints and costs. *IEEE transactions on control systems technology*, *24*(4), 1394–1406.

You, C., Lu, J., Filev, D., & Tsiotras, P. (2019). Advanced planning for autonomous vehicles using reinforcement learning and deep inverse reinforcement learning. *Robotics and Autonomous Systems*, *114*, 1–18.

Ziegler, J., Bender, P., Schreiber, M., Lategahn, H., Strauss, T., Stiller, C., … others (2014). Making bertha drive—an autonomous journey on a historic route. *IEEE Intelligent transportation systems magazine*, *6*(2), 8–20.

APPENDIX A: ARDUINO CODE

```
include ”CytronMotorDriver.h”
#include <Encoder.h>
#include <math.h>

Encoder myEncoder1(2, 3);
Encoder myEncoder2(18, 19);

// Configure the motor driver.
CytronMD motor1(PWM_DIR, 9, 11);
// PWM 1 = Pin 9, DIR 1 = Pin 11.
CytronMD motor2(PWM_DIR, 10, 12);
// PWM 2 = Pin 10, DIR 2 = Pin 12.

int myEraser = 7;
int myPrescaler = 1;
int count=0;
float cmd1=0, cmd2=0;

const unsigned long timePeriod = 10;
unsigned long startTime;
long startPosition1 , startPosition2;
float omega1, omega2, distance1=0, distance2=0;
float fact1 , fact2;
double output1 , output2;

void setup() {
  TCCR2B &= ~myEraser;  //Changing pin frequency 9 and 10
  TCCR2B |= myPrescaler;//Changing pin frequency 9 and 10
```

```
  Serial.begin (9600);
  startPosition1 = myEncoder1.read();
  startPosition2 = myEncoder2.read();
  startTime = millis();

  }
  long oldPosition1 = -999;
  long oldPosition2 = -999;

void loop() {
    String data =Serial.readString();
    String xval = getValue(data, ',', 0);
    String yval = getValue(data, ',', 1);

    cmd1=xval.toFloat();
    cmd2=yval.toFloat();

   fact1 = 15*1.0575;        // Motor omega correction factor
   fact2 = -15;
  motor1.setSpeed(fact1*cmd1);
  motor2.setSpeed(fact2*cmd2);

  unsigned long now = millis();
  long newPosition1 = myEncoder1.read();
  long newPosition2 = myEncoder2.read();

  if (newPosition1 != oldPosition1) {
    oldPosition1 = newPosition1;
```

```
    distance1 = (1.276*0.5*newPosition1)/(1200*16);

    startPosition1 = newPosition1;
  }
   if (newPosition2 != oldPosition2) {
    oldPosition2 = newPosition2;
    distance2 = (1.276*0.5*newPosition2)/(1200*16);
    startPosition2 = newPosition2;

  }
      String topi=String(distance1)+","+String(distance2);
      Serial.println(topi);

  }

  String getValue(String data, char separator, int index)
{
    int found = 0;
    int strIndex[] = { 0, -1 };
    int maxIndex = data.length() - 1;

    for (int i = 0; i <= maxIndex && found <= index; i++) {
        if (data.charAt(i) == separator || i == maxIndex) {
            found++;
            strIndex[0] = strIndex[1] + 1;
            strIndex[1] = (i == maxIndex) ? i+1 : i;
        }
    }
    return found > index ?
```

```
        data.substring(strIndex[0], strIndex[1]) : "";
    }
```

APPENDIX B: TRAJECTORY PLANNER

```
#!/usr/bin/python3
from sys import path
path.append(r"/home/prajwal/Libraries/Casadi_Py_Lib/")
from casadi import *
import matplotlib.pyplot as plt;
import numpy as np;
import time;

class Traj_Planner(object):
    def __init__(self, opti, prbls):
        self.opti = opti;
        self.N = prbls['N']; self.dt = prbls['dt'];
        self.No_obs = prbls['No_obs'];
        N = self.N;
        x = self.opti.variable(1, N+1);
        y = self.opti.variable(1, N+1);
        theta = self.opti.variable(1, N+1);
        self.X = vertcat(x, y, theta);
        v = self.opti.variable(1, N);
        w = self.opti.variable(1, N);
        self.U = vertcat(v,w);
        self.X_0 = self.opti.parameter(3,1);
        self.X_f = self.opti.parameter(3,1);
        self.X_dot = self.f_x_u();
        self.subject_to_dynamic_constraints();
        self.subject_to_static_constraints();
        self.Obs_pos = self.opti.parameter(2,self.No_obs);
        self.Obs_rad = self.opti.parameter(1,self.No_obs);
```

```
        cost_obstacles = self.Obstacle();
        self.opti.minimize( 1*sum2((x-self.X_f[0])**2) +
         1*sum2((y-self.X_f[1])**2) +
         1*sum2((theta-self.X_f[2])**2) +
         0.5*sum1(sum2((self.U)**2)) +
         cost_obstacles);
        p_opts = {"expand":True};
        s_opts = {"max_iter": 1000};
        opti.solver("ipopt", p_opts, s_opts);

    def return_solution(self, X_now, X_ref, Obstacles,
                Initialization=None):
        if not(Initialization is None):
            self.opti.set_initial(self.U,Initialization['U']);
            self.opti.set_initial(self.X, Initialization['X']);
        if self.No_obs>0:
            self.opti.set_value(self.Obs_pos,
                 Obstacles[0:2,0:self.No_obs]);
            self.opti.set_value(self.Obs_rad,
                 Obstacles[2,0:self.No_obs]);
        self.opti.set_value(self.X_0, X_now);
        self.opti.set_value(self.X_f, X_ref);
        sol = self.opti.solve();
        return sol;



    def f_x_u(self):
```

```
        x = MX.sym('x',3);
        u = MX.sym('u',2);
        f_x_u = vertcat(u[0]*cos(x[2]), u[0]*sin(x[2]), u[1]);
        X_dot_act = Function('X_dot',[x, u],[f_x_u]);
        return X_dot_act;

    def subject_to_static_constraints(self):
        self.opti.subject_to(self.X[:,0]==self.X_0);
        self.opti.subject_to(0<=self.X[:]);
        self.opti.subject_to(self.X[:]<=12);
        self.opti.subject_to(self.U[0,:]<=1);
        self.opti.subject_to(-1<=self.U[0,:]);
        self.opti.subject_to(self.U[1,:]<=1.5);
        self.opti.subject_to(-1.5<=self.U[1,:]);


    def subject_to_dynamic_constraints(self):
        for k in range(self.N):
            self.opti.subject_to( self.X[:,k+1]== self.X[:,k] +
                self.dt *
                self.X_dot(self.X[:,k], self.U[:,k])  );


    def Obstacle(self):
        obst_cost = 0;
        if self.No_obs>0:
            No_obs = self.No_obs;
            Obs_pos = self.Obs_pos; Obs_rad = self.Obs_rad+0.1;
            for i in range(No_obs):
```

```
                h =  log(
                        ((self.X[0,:]- Obs_pos[0,i])/Obs_rad[i])**2
                        +((self.X[1,:]-Obs_pos[1,i])/Obs_rad[i])**2
                        );
                self.opti.subject_to(h>0);
                obst_cost = obst_cost + sum2(exp(5*exp(-h)));
        return obst_cost;



def main():
    opti=casadi.Opti();
    dt = 0.2;
    prbls = {'N':10, 'dt':dt, 'No_obs':3};
    simtime = 20;
    traj = Traj_Planner(opti, prbls);
    print(traj.opti);
    X_now = [0,0,0];
    X_ref = [10,10,pi];
    Obstacles = horzcat([7,4,0.5],[4,4,0.5], [4,7,0.5]);
    #MPC loop:
    iter = int(simtime/prbls['dt']);
    X_es = horzcat(X_now, DM(3, iter));
    U_s = DM(2,iter);
    sol = traj.return_solution(X_now, X_ref, Obstacles);
    Initialization ={'X':sol.value(traj.X),'U':sol.value(traj.U)};
    tic = time.time();
    future_traj = [];
    for i in range(iter):
```

```
        sol = traj.return_solution(X_now, X_ref,
                                   Obstacles, Initialization);
        u_set = sol.value(traj.U);
        x_set = sol.value(traj.X);
        Initialization = {'X':x_set, 'U':u_set};
        # X_now = DM(X_now) + dt*traj.X_dot(X_now,u_set[:,0]).full()
        X_now = x_set[:,1]; future_traj.append(x_set);
        X_es[:,i+1] = X_now; U_s[:,i] = u_set[:,0];
    toc = time.time(); print(toc-tic,'seconds elapsed')
    x = X_es[0,:];
    y = X_es[1,:];
    theta = X_es[2,:];
    v = U_s[0,:]; omega = U_s[1,:];
    t = [i*dt for i in range(iter+1)];
    print(type(x))
    plt.plot(x.T,y.T);
    for i in range(prbls['No_obs']):
        angl = [k/100*2*pi for k in range(101)];
        plt.plot(Obstacles[0,i] + Obstacles[2,i]*cos(angl),
                 Obstacles[1,i] + Obstacles[2,i]*sin(angl));
    plt.axis('square'); plt.title('x-y trajectory');
    plt.savefig('figure_traj.png'); plt.show();
    plt.plot(t, x.T, label="x coordinate");
    plt.plot(t, y.T, label="y coordinate");
    plt.plot(t, theta.T, label="heading angle");
    plt.plot([0, 20], [10,10], '--');
    plt.plot([0, 20], [pi,pi], '--');
    plt.legend(loc="upper left");
    plt.savefig('xandyandtheta.png');
```

```
    plt.show();
    plt.step(t, np.append(v, v[-1]),
             label="velocity (m/s)");
    plt.step(t, np.append(omega, omega[-1]), label="angular velocity
    plt.legend(loc="upper right"); plt.savefig('vandomega.png');
    plt.show();
    plot_diffdrive_control(v, omega, t);
    showprogression(x.T,y.T, theta.T, prbls, Obstacles, future_traj)

def plot_diffdrive_control(v, omega, t):
    L = 0.15; #15 cm
    r = 0.05; #5 cm
    omega_r = (2*v + omega*L)/2/r;
    omega_l = (2*v - omega*L)/2/r;
    plt.step(t, np.append(omega_l, omega_l[-1]),
        label="angular velocity of left wheel (rad/s)");
    plt.step(t, np.append(omega_r, omega_r[-1]),
        label="angular velocity of right wheel (rad/s)");
    plt.legend(loc="upper right"); plt.savefig('omega_landr.png');
    plt.show();




def showprogression(x,y,theta, prbls, Obstacles, future_traj):
    L=0.5; H=0.625;
    interval = len(x.full())//5;
    for i in range(0, len(x.full())+1, interval):
        tri = givemexandy((x[i],y[i]),L,H,theta[i]);
```

```
        plt.plot(x[0:i+1],y[0:i+1]);
        for j in range(prbls['No_obs']):
            angl = [k/100*2*pi for k in range(101)];
            plt.plot(Obstacles[0,j] + Obstacles[2,j]*cos(angl),
            Obstacles[1,j] + Obstacles[2,j]*sin(angl));
        plt.fill(tri[0,:], tri[1,:], "m");
        if i<len(future_traj):
            curr_fut = future_traj[i];
            plt.plot(curr_fut[0,:],curr_fut[1,:]);
        plt.axis('square'); plt.savefig('figure'+str(i*prbls['dt'])+
        plt.show();

def givemexandy(center_location,L,H,phii):
    center0=center_location[0];
    center1=center_location[1];
    R = np.array([[np.sin(phii), np.cos(phii)],
                  [-np.cos(phii), np.sin(phii)]]);
    X=([-L/2, L/2, L/2, -L/2]);
    Y=([-H/2, -H/2, H/2, H/2]);
    T = np.zeros([2,4]);
    for i in range(4):
        T[:,i] = np.dot(R, np.array([ [X[i], Y[i]] ]).T).reshape(2);
    x_lower_left=center0+T[0,0];
    x_lower_right=center0+T[0,1];
    x_upper_right=center0+T[0,2];
    x_upper_left=center0+T[0,3];
    x_upper = (x_upper_left + x_upper_right)/2;
    y_lower_left=center1+T[1,0];
    y_lower_right=center1+T[1,1];
```

```
    y_upper_right=center1+T[1,2];
    y_upper_left=center1+T[1,3];
    y_upper = (y_upper_left + y_upper_right)/2;
    x_coor =[x_lower_left, x_lower_right, x_upper];
    y_coor =[y_lower_left, y_lower_right, y_upper];
    xandy = np.array([x_coor, y_coor]);
    return xandy

if __name__ == '__main__':
    main();
```