Self-Parking Car in a Smart Parking Lot

CPES Project Group - 23

Team Members
Jahnavi - S20170010107
Bhavana - S20170010025
Manasa Gavini - S20170020198
Prasanna - S2017001088
K V Navya Sree - S20170010065
Thoyiba - S20170010165

Objective

To model, design, and analyze a self-parking car in a smart parking lot. The car that we modeled will find an empty slot and parks itself there. The car is parked properly in the slot as we have defined the path already.

In our model we have used one ultrasonic sensor in the front of the car to detect the obstacles and also calculate the distance between the car and the object. After computing, if the distance is less/more than the threshold, specific actions may be taken to avoid the obstacle.

We use a line detection sensor to detect the path to be followed.

For path modeling we have a used Simulation Map Generator App which takes a picture and returns a map after which obstacles may be placed.
Therefore, the complete project has been divided into 3 components

1. The car
2. The line following technology
3. The object detection technology

Software Requirements

- MATLAB
- Simulink
- Symbolic Math Tool Box
- Mobile Robotics Training Tool Box
- Simulation Map Generator
- Image Processing Toolbox
- Robotics System Toolbox
- Control System Toolbox
- System Identification Toolbox

Basic Car

This basic model of the car has 2 DC motors attached to the left and right motors of the car. We used encoder sensors for localization. We compute the number of rotations of the wheel to determine the distance traveled

Here an encoder is a device connected to wheel motor shaft such that when the wheel rotates 360 degrees the encoder counts discrete number of ticks

In our case 9 ticks = 1 revolution

Number of Rotations = Total encoder ticks/ Number of ticks per rotation.

Distance traveled = Number of rotations * 2*pi* wheel radius

Map Simulation

For simulating a map we have used a Simulation Map Generator App and loaded the picture of our map and specified the size of the map in meters [3 1.5] to map the real world distance to the map that is to be generated. An appropriate threshold is applied and obstacles are placed in the map. Now the map is exported to .mat file. After exporting a variable named mapForSim is loaded to the simulation workspace.

The line following technology

We used the VEX light Sensor which allows the robot to sense the ambient light in a room. Unlike the Line Tracking Sensor, the Light Sensor does not generate any light, it only senses the amount of light already present in an area. The Light Sensor is an analog sensor, and it returns values in the range of 0 to 4095. Higher values indicate that the sensor is in a darker area, and lower values indicate lighter areas.

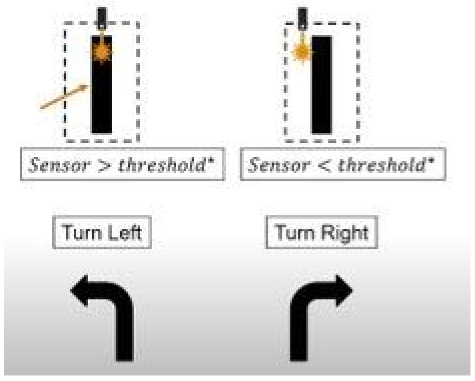
Its value is given by, threshold value = (Environment value + Line value)/2

Designing the line following Algorithms

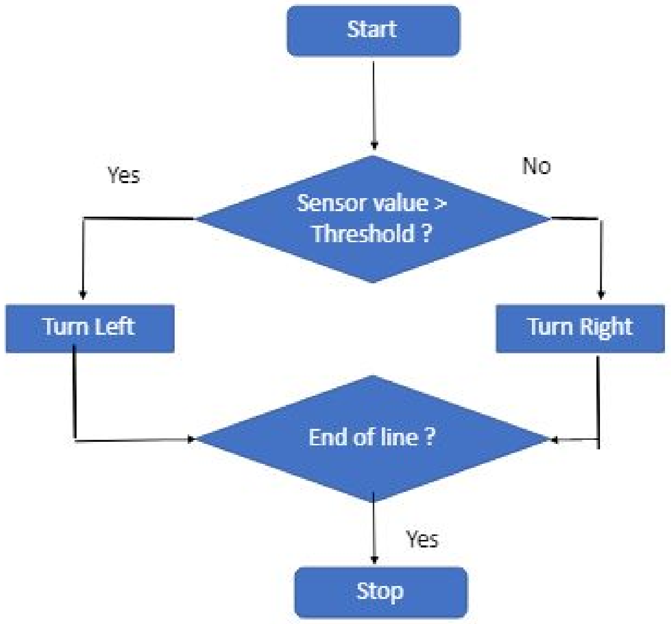
We used two types of algorithms

1. On and Off Algorithm

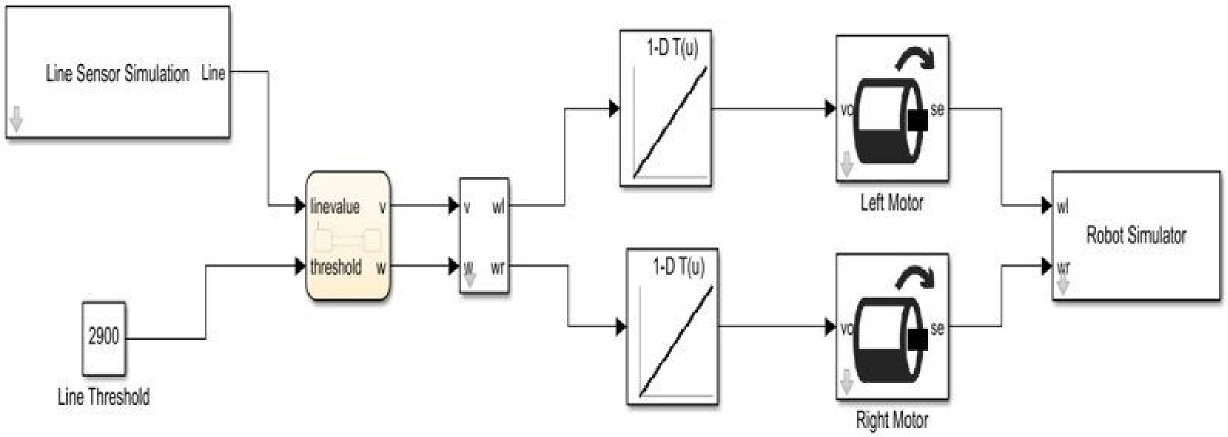
The basic identification is if the Sensor value is greater than the threshold the car should move left and if it’s less than the threshold value it should take a right turn.

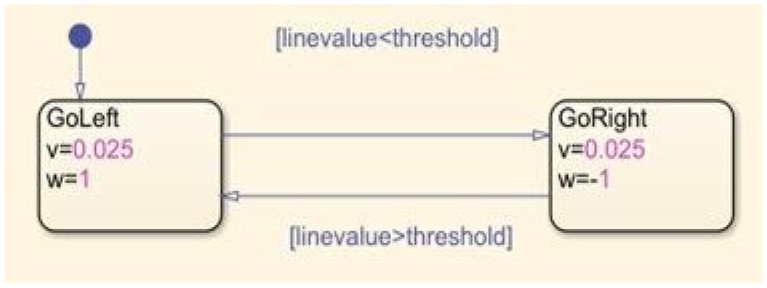


The flow chart given below explains the algorithm.



Simulation of the Path following Robot





- logic chart

Obstacle Avoidance Technology

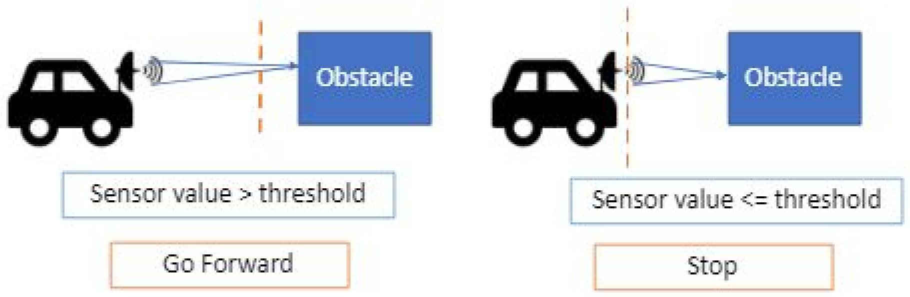
While parking if the robot finds that there is a car already present in the slot it should not park there. To check whether the slot is empty or no we used an ultrasonic sensor. It is used to detect objects at a distance, without the need for the robot to actually contact them. The Ultrasonic sensor uses sound pulses to measure distance, in a similar way to how bats or submarines find their way around. By emitting an ultrasonic pulse and timing how long it takes to hear an echo, the ultrasonic sensor can accurately estimate how far away the object is.

Distance = speed of sound x time taken /2

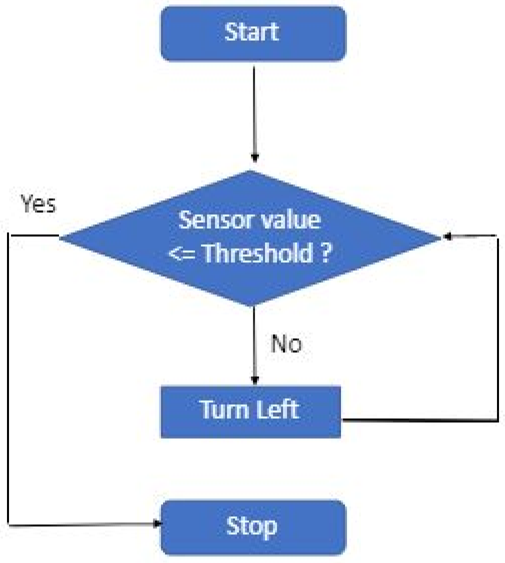
Designing the Obstacle Avoidance Algorithms

We used two types of algorithms

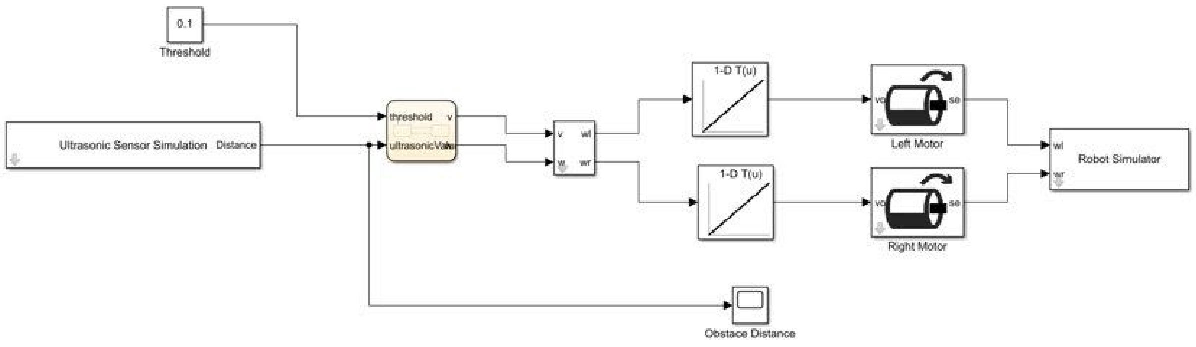
1. On and Off Algorithm

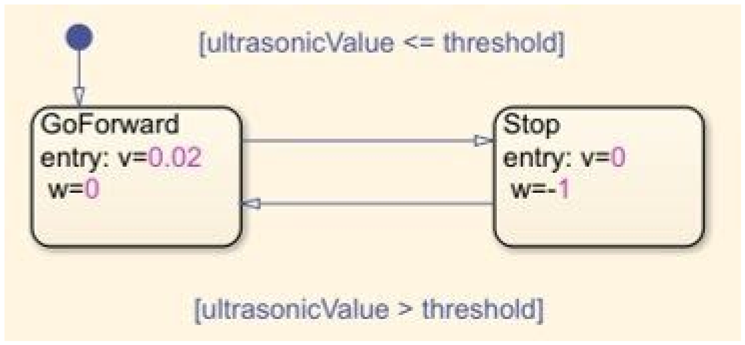


The flow chart given below explains the algorithm.



Simulation of the Obstacle Avoidance Robot



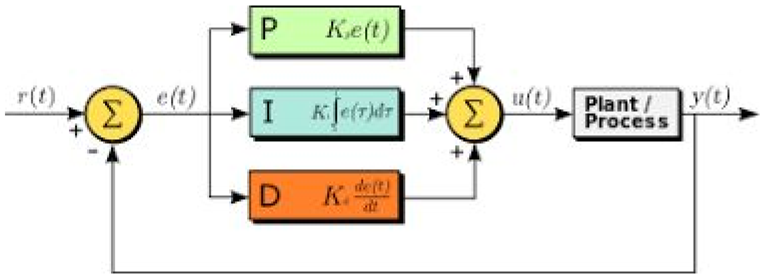


- logic chart

PID Controllers

A proportional–integral–derivative controller is a modulated control feedback mechanism to regulate the process variables (distance and velocity in our case). PID controllers are often used in applications which require precise and accurate decisions to be made which otherwise may lead to fatal accidents in real life.

- Working :



The proportional, integral and derivative parameters of the controller must be individually tuned. A correction factor is calculated and this is compared with the required value (corrections are made according to error value calculated)

Proportional tuning involves correcting a target proportional to the difference. The target value is never achieved because as the difference approaches zero, so too does the applied correction.

Integral tuning attempts to remedy this by effectively cumulating the error result from the "P" action to increase the correction factor. For example, if the oven remained below temperature, “I” would act to increase the head delivered. However, rather than stop heating when the target is reached, "I" attempts to drive the cumulative error to zero, resulting in an overshoot.

Derivative tuning attempts to minimize this overshoot by slowing the correction factor applied as the target is approached.

- **Integrating PID with Supervisory Logic**
  We are using stateflow to represent the different segments of path navigation.We are currently using simulink functions to bring our PID controllers into the path navigation model. Followed by this, we call the functions from within the states in the flow chart appropriately to enable complete working of the car prototype.

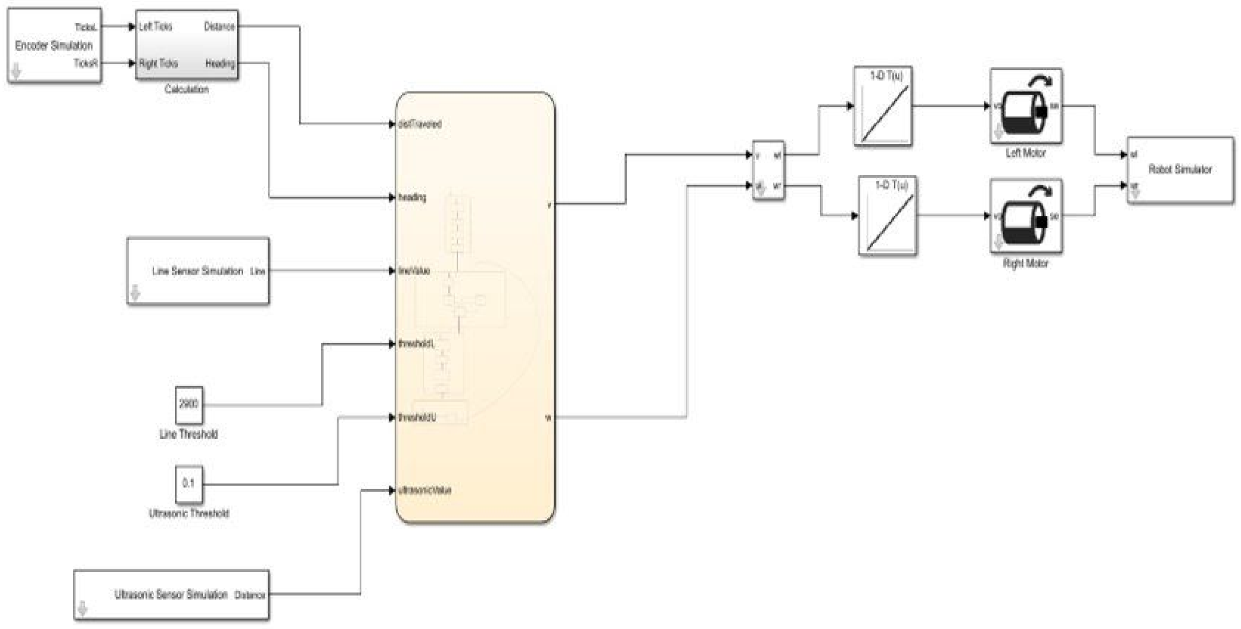
$$P + I \cdot T_s \frac{1}{z-1} + D \frac{N}{1 + N \cdot T_s \frac{1}{z-1}}$$

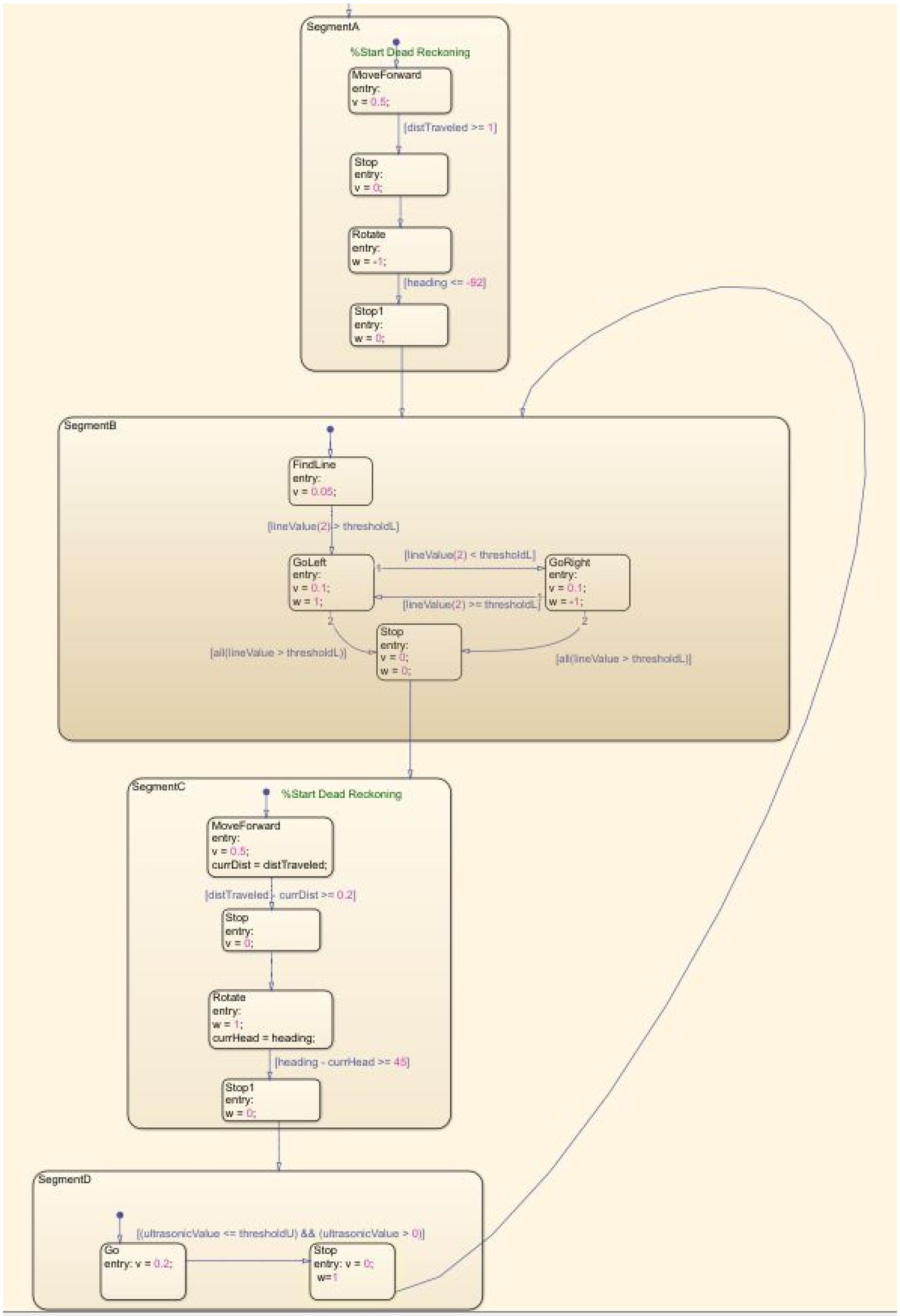
PID

$$P + D \frac{N}{1 + N \cdot T_s \frac{1}{z-1}}$$

$$P + I \cdot T_s \frac{1}{z-1}$$

Complete Simulation of the Robot




SegmentA
%Start Dead Reckoning
MoveForward
entry:
v = 0.5;
[distTraveled >= 1]
Stop
entry:
v = 0;
Rotate
entry:
w = -1;
[heading <= -92]
Stop1
entry:
w = 0;
SegmentB
FindLine
entry:
v = 0.05;
[lineValue(2) > thresholdL]
GoLeft
entry:
v = 0.1;
w = 1;
[lineValue(2) < thresholdL]
1
[lineValue(2) >= thresholdL]
1
GoRight
entry:
v = 0.1;
w = -1;
2
2
[all(lineValue > thresholdL)]
Stop
entry:
v = 0;
w = 0;
[all(lineValue > thresholdL)]
SegmentC
%Start Dead Reckoning
MoveForward
entry:
v = 0.5;
currDist = distTraveled;
[distTraveled - currDist >= 0.2]
Stop
entry:
v = 0;
Rotate
entry:
w = 1;
currHead = heading;
[heading - currHead >= 45]
Stop1
entry:
w = 0;
SegmentD
[(ultrasonicValue <= thresholdU) && (ultrasonicValue > 0)]
Go
entry: v = 0.2;
Stop
entry: v = 0;
w=1

References

- https://in.mathworks.com/help/supportpkg/arduino/ref/ultrasonicsensor.html
- Abdulhamid, Mohanad & Mutheke, Muindi. (2018). Design of Digital Control System for Line Following Robot. Technological Engineering. 15. 48-52. 10.1515/teen-2018-0018.

Individual Contribution

- Modelling path following part - Navya, Thoyiba
- Design of path following - Navya, Manasa
- Designing of obstacle detecting - Bhavana, Jahnavi
- Modelling of obstacle detecting - prasanna, jahnavi
- Analysis of path following robot - Thoyiba, Bhavana
- Analysis of robot parameters - Manasa Prasanna
- Pid of both the parts and complete integration - Jahnavi , Navya