Image Processing and Machine Learning

Miniature Model of Self Driving Car

Documentation

By Himanshu Poddar

Contents

List of Figures

1 Introduction

I was always intrigued about the working of a self driving car. There were always questions that remained unanswered to me, how does the car do lane detection in the areas where we have dirt road, the vision sensors in the car can detect signals from far away, how does it calculates the distance exactly to take decisions eg stopping a car, taking turns. How does the lane detection and lane changing operations work? How does the Tesla Autopark system works? Avoiding obstacles in the roads and calculating the space just enough for the AV car to accurately fit into the space and move forward remains the biggest mathematical question that I was curious about. One of the major causes of accidents are the lane turn process. How does the AV car reduces the risk of accidents during lane changes by monitoring the dangerous blind spot area. The detection of vehicles in the blind spot area when changing lanes using the blind spot monitor were the topic of interests mainly. Are autonomous cars safe enough in all terrains. Would it work in countries like India, Pakistan, Bangladesh where we have narrow roads, no proper infrastructure where traffic violations are a norm rather than an exception? Also let's say the car started and now both left and right vision of the car is hidden by obstacles or other cars which are moving parallely to the car, how would the car identify the lanes in this case, what if a vehicle comes from the sideways and hits the autonomous car. What would be the series of decisions being taken in that case?

With all those questions in my mind I researched about the topics and tried to answer few of these questions myself by developing my own miniature model of a self driving car using the on board computer Raspberry PI and Arduino UNO to drive the wheels of the car. Initial development question would include detection of lanes, making the car stay in the centre of the road, taking turns and the end of lane detection.

2 System Design

2.1 Hardware Workflow Design

Here, the proposed Master-Slave architecture is shown in the Figure 2.1. A power source (battery) powers the Raspberry Pi through a voltage regulator, which is used since a steady, reliable voltage is needed. The car model is first constructed and mounted with a camera module to capture road images. These road images are processed in real time along with this data from other sensors indicating traffic signals, stop signs, obstacles are also taken for ML model. All this data will be fed to a Neural Network. The hence obtained model can predict the action to be taken at any point when new real time data is provided. The prediction will cause the Master device, Raspberry Pi to send a signal to the Slave device, Arduino Uno, which in turn drives the Motor to control the steering wheels of the car. Based on the road lane image and other data obtained from the environment, voltage of the signal sent from the Raspberry Pi to the Arduino Uno will vary and the car will follow a different trajectory and at different speeds accordingly.

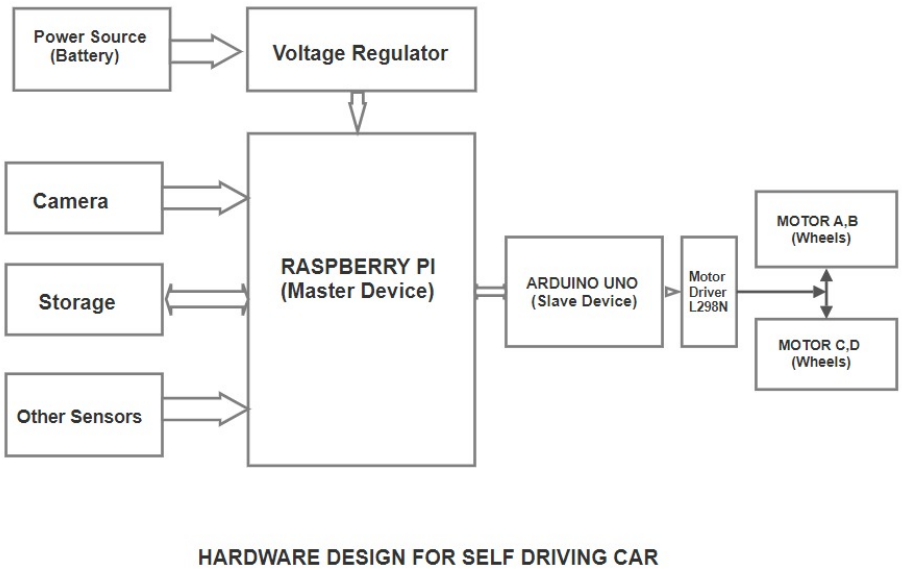

Figure 1: Hardware setup and system design

2.2 Slave Device Configuration

The first and most basic step of the project is to ensure the slave device (Arduino) is able to control the motors driving the movement of the car. Here, a H-bridge is used to interface the the DC Motors to the Arduino board, to prevent the motors (which operate at 3-4V) from being fused. High and Low Pins of H-Bridge connected to the Motors (the digital HIGH or LOW signal is supplied to ensure forward and backward movement). Enable pin is supplied with an analog signal so as to use pulse width modulation to vary the speed of the motor. The code snippets for the same have been shown below.

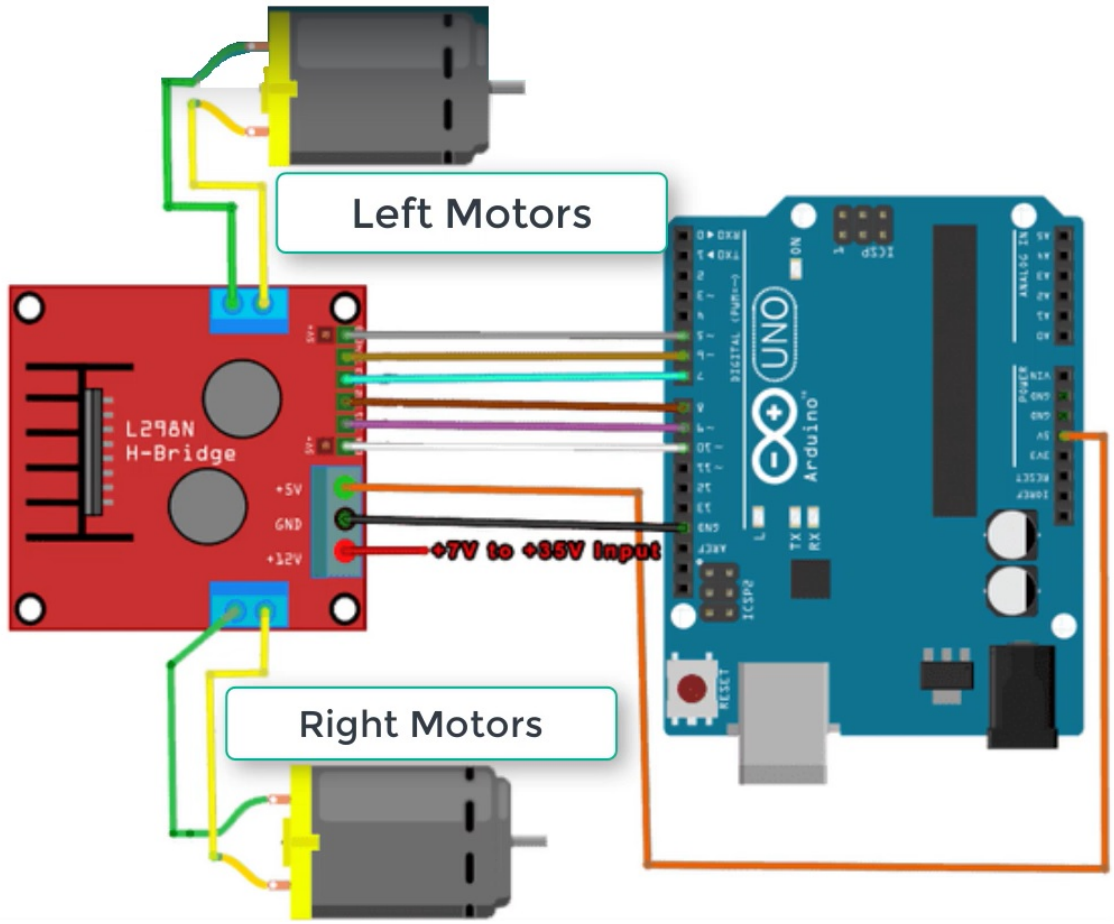

Figure 2: Slave Device Configuration

2.3 Master Device Configuration

The master device, which is the Raspberry Pi, needs to be able to take in video input from the camera module provided, and based on the objects detected in the image, take decisions as to whether to move the motor forward, backward, left or right. First, let's look into the configurations required for camera module input to be available for processing in the Raspberry Pi. Raspicam (to access camera module) and OpenCV (for image processing) need to be installed on the Pi.

2.4 Model Assembly

The proposed model of the assembled model of our robo car along with all the components at the tier I and the tier II level of our model. The lest figure shows the T1 level of the car along with the power source and the right figure shows the T2 level of the car along with the on board computer Raspberry PI. The first step of the implementation is assembling the car model using the hardware specifications mentioned above. Below, a picture detailing the inner assembly design is depicted.

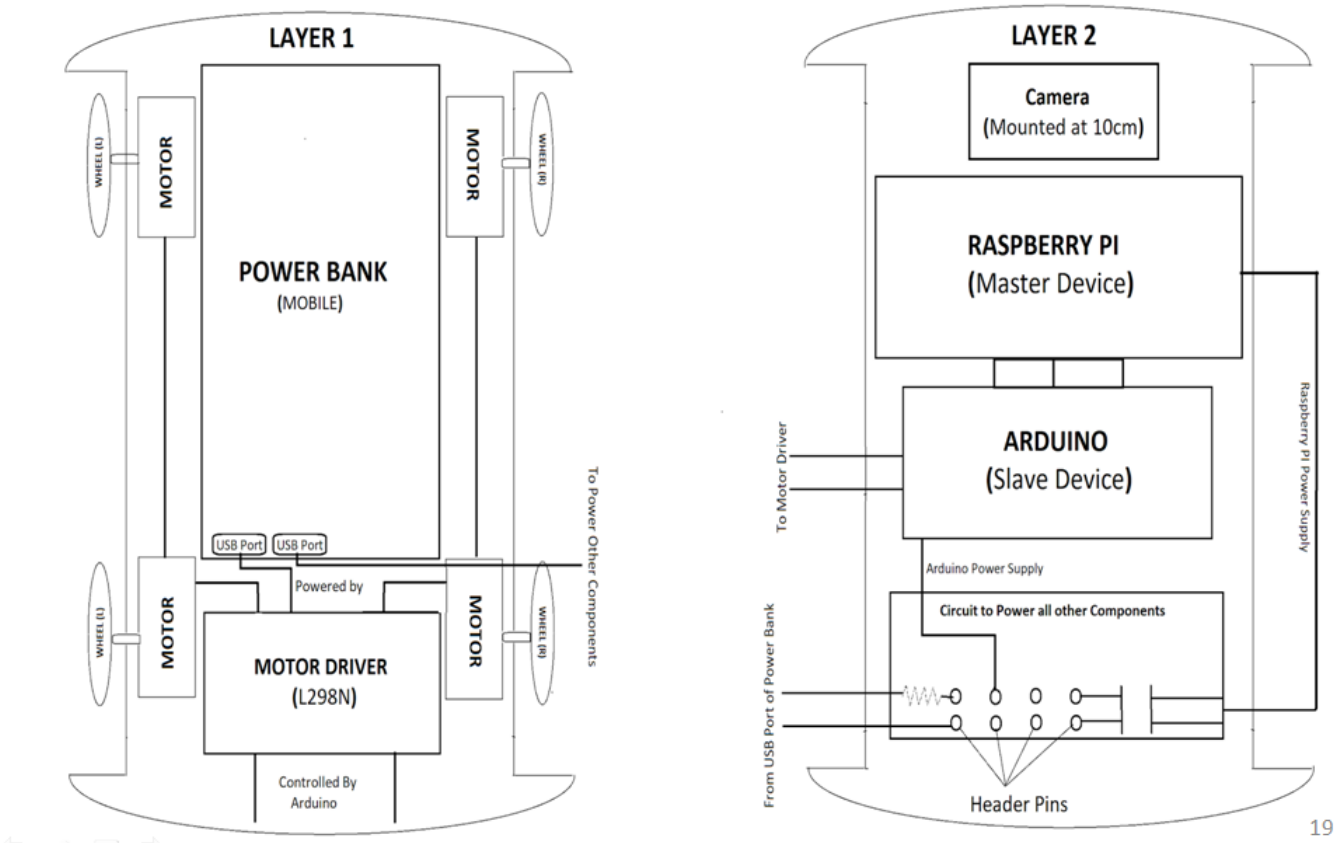

Figure 3: Inner design of car model

2.5 Power Source Circuit Diagram

A common circuit that supplies power to all the components in our car including the Raspberry PI, Arduino, Cooling unit. The circuit power source would be the power bank that we have added in the layer 1 of our model assembly. The capacitor is required in case we have voltage fluctuations, the raspberry PI would start rebooting and hence just to prevent RPI from going down the capacitor would start supplying power in our circuit so that the RPI gets continuous supply of power without interruptions.

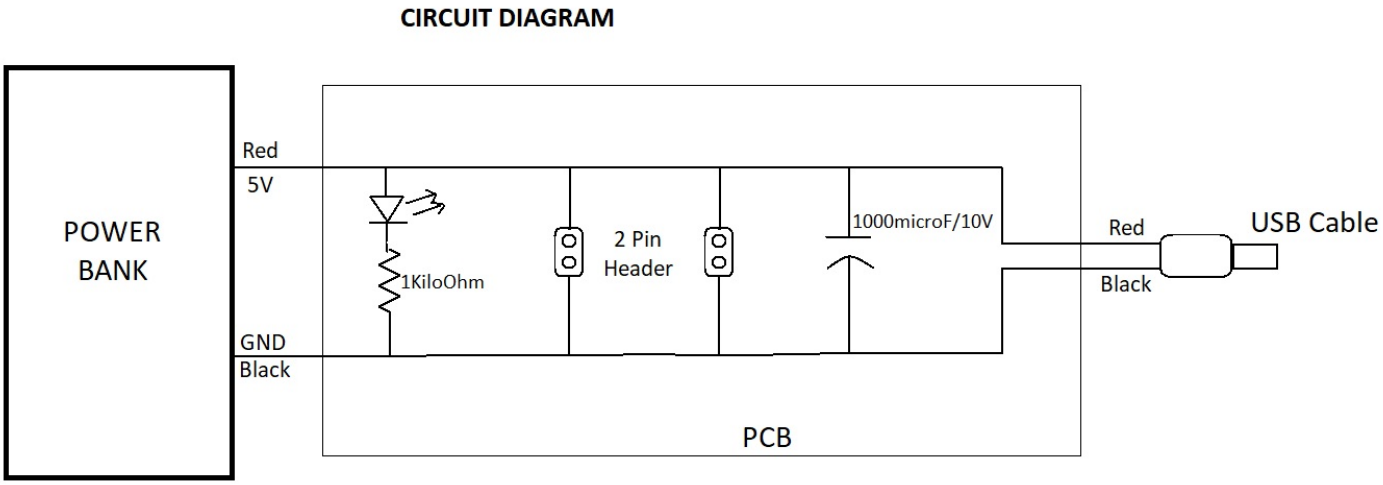

Figure 4: Circuit Diagram for Powering devices

3 Hardware and Software Requirements

3.1 Hardware Requirements

1. Robot Chassis (Comprising of 2 base plastic plates, 4 rubber tyres, 4 DC gear motor, screws, nuts)
2. L298 Motor Driver module
3. Arduino UNO
4. Raspberry Pi 3B, 64-bit 1.2GHz Quadcore processor with WiFi Module
5. Power source: Slim 16800 MAh battery
6. Raspberry Pi Camera Module v2 with CSI cable
7. Camera mounting case
8. Red/Green/Yellow LEDs
9. 2 USB Cable with open ends and one micro USB to power RPI
10. Capacitors(1000 µF, 10V)
11. 16 GB Class 10 SD Card
12. Double Sided Tape
13. PCB
14. Track comprising of Stop signal, Traffic signal and obstacles
15. Black and White chart paper
17. Soldering machine
18. Multi-meter
19. Male-Header Pins
20. Resistors(1K)
21. Bread Board
22. Raspberry PI cooling fan

3.2 Software Requirements

Coding Languages: C++
Tools: Arduino IDE, Geany Editor, win32 Disk Imager
Libraries and Frameworks: OpenCV4, raspicam_cv, wiringPi

4 Implementation

4.1 Assembly and setup

1. Assemble the Hardware Parts for Robo car kit. Solder the motors of each side as one.
2. Build the track. White lane strip length 9mm.
3. Distance between white lane strips is almost 29cm including the width of while lanes.
4. The circuit we made above(Figure 4), supplies power to the Arduino from 5V pin using the header pin of the circuit.

5. Arduino is connected to HBridge, with which we control the speed of the motors.
Note that : To rotate clockwise/anti-clockwise the Left Right Pin must have opposite polarity, if Left is High then Right Pin must be low. also with the enable pin you can increase/decrease speed of motors. Left and right Enable Pins must be connected to PWM pins of arduino UNO.

6. Raspberry PI 3B with Wi-Fi.

7. Either install Raspbian OS from official website or use my backup.
Use following SD card backup.
restore image on SD card with Win32 Disk Imager tool.
Username : pi
Password : home
Ethernet : raspberrypi.local

8. Packages required : OpenCV for image processing, Raspicam package to control camera module from C++ code, WiringPi library to access the RPI pins
My backup already has all the required package installed.

9. Mounted camera at a certain height.(Length of stick around 15-16 cm), try to avoid the car part in the field of vision.
10. Use win32 Disk Imager to backup the sd card.
11. Enable camera on RPI, to calculate the frame rate you can use the chrono library.

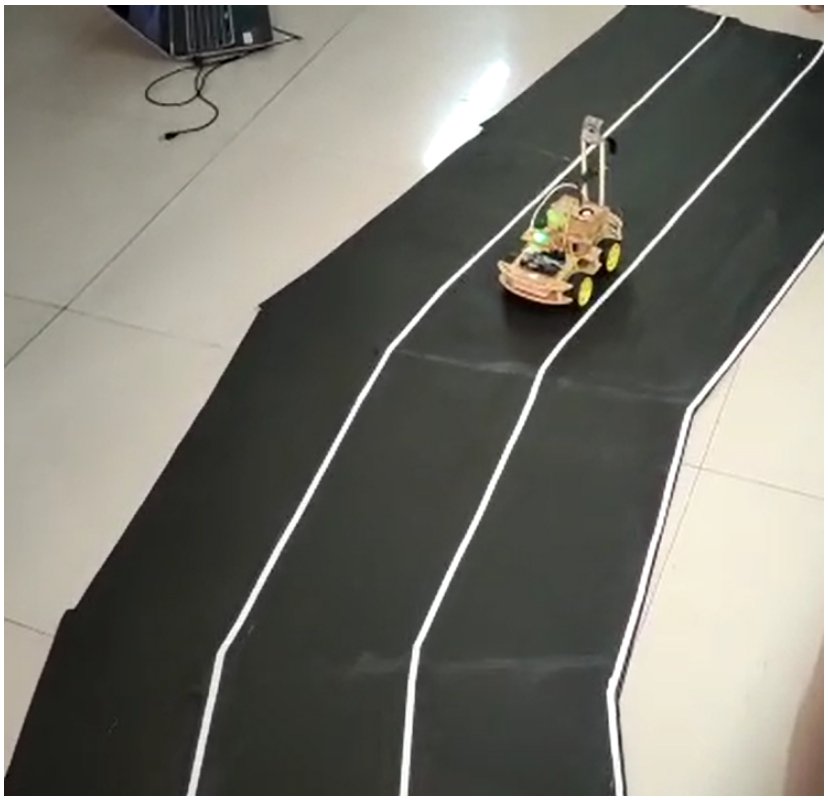

Figure 5: Car on the Track

4.2 Car Model

The car has a camera mounted at 15cm to get a wide angle view of the surrounding. The camera is connected to the RPI using CSI cable.

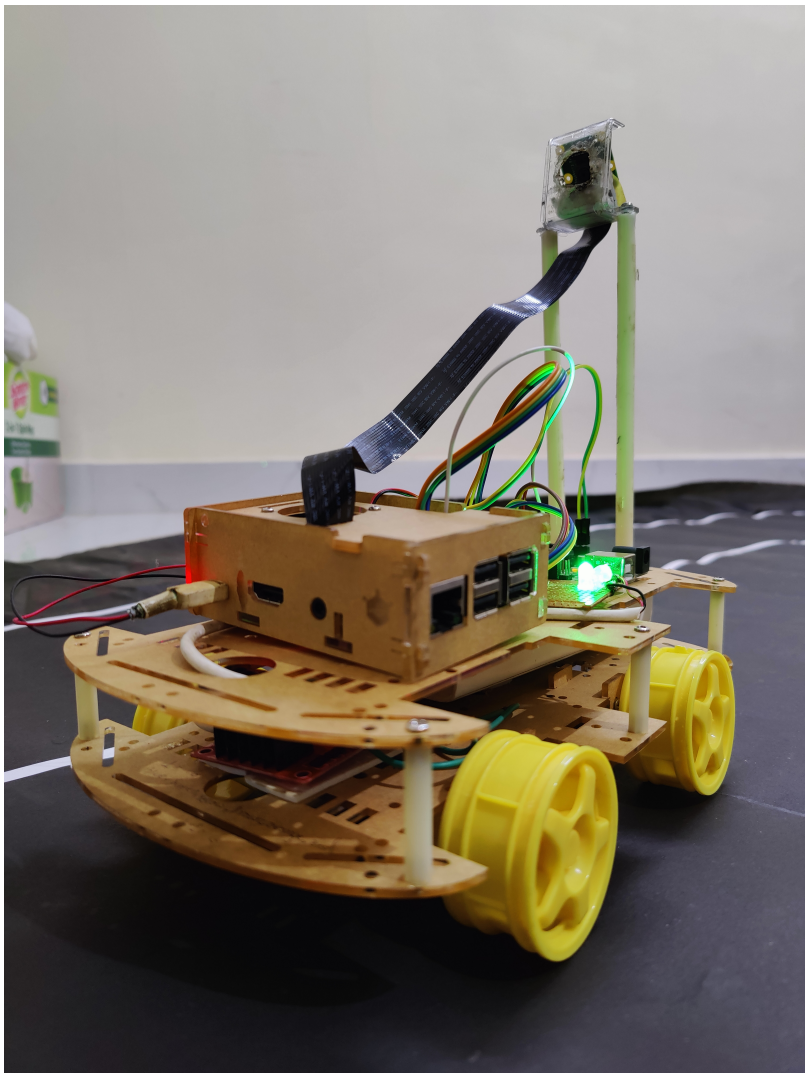

Figure 6: Car Model

4.3 Lane Detection and Image Processing

Now that camera module is initialised, we need to program the car to make its movements based on camera input i.e., we need to develop lane detection functionality.
A summarized procedure to do this would be :
a) Through experimentation, the region of interest boundaries are fixed to the closest approximate value.
b) The lane image could be transformed to a bird's eye view.
c) Image is converted to grayscale and thresholded (binarizing the image)
d) Apply canny edge detection to clearly outline the edges of the lane boundaries, combine this with the thresholded image to obtain well-defined lane boundaries.
e) Use histogram plot (of pixel intensities) to get lane boundary co-ordinates and accordingly determine the lane centre.
f) Find the difference between camera gaze frame centre and the determined lane centre (blue line in the video) and store in a variable 'result.'
g) While result = 0, the car is travelling on the lane, if result != 0, the car is deemed to not be travelling on the lane.

4.3.1 Image Processing Pipeline

In an infinite loop we are capturing the frame from the camera. Check *Capture* method in the code.
The frame captured would contain the entire field of view of the camera. But do we really need the top half part of the road ahead? We are just interested in the road that is immediate to our camera.

Hence we create a region of interest by creating a red color rectangle on the bottom of the captured frame which we will be ultimately be using to make lane following decisions. The process involves the following steps :

- Perspective transformation

  Inside the perspective method first we created the region of interest using the 4 source points. These 4 points are chosen such that our lane perfectly fits inside and there is approximately equal distance of the white lane and the red line we drew.
  Then we take a perspective view of the region of interest to get the bird eye view of the track. To get the coordinates of the 4 points follow a hit and trial approach and keep adjusting the points until you get a perfect view. Make sure your left and right lines are parallel to the lanes. so that we get lanes in parallel in the perspective view.

- Thresholding

  In this step, we try to identify our white lane lines, to do this we can convert our image to grayscale so that the white lane lines are visible at highest intensity in the grayscale frame. Intensity for deep black color will be 0 in the frame matrix and 255 for deep white images.

  In the method *Thresholding*, inRange parameter for min and max values may be different in different lighting conditions.

- Canny edge detection

  We will be using Canny edge detection to get edges in an image. Here we are doing canny and thresholding both and adding the result frames from both on which finally we will be operating on. This is done just to make sure in different lighting condition if thresholding goes wrong and the image obtained does not have clear white lanes then canny edge detection would work fins and give us the lanes perfectly as the intensity values will still be maximum even if only edges are visible for the lanes.
  Now our next requirement would be to find the exact position of our lane lines with respect to the frame. We create a region of interest at the bottom.

- Histogram

  Creating Region of Interest for lane detection and Histogram operations in the image below.

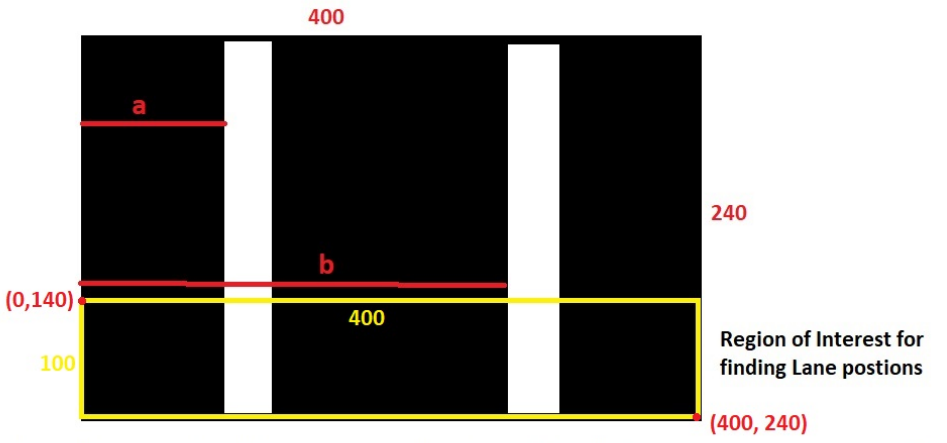

Figure 7: Region of Interest and Lane Detection

The yellow rectangle as shown in image which is our ROI can be taken out from the frame using the Rect command in open cv : (0,140,1,100). But to get the intensity value of the maximum strip which will be our lane we need to divide our yellow ROI in some parts say width of each part would be 1 pixels. So now we can go through each strip of 1pixel width, sum the values and for the strip for which we get highest maximum value will be our very first lane. See Figure 8.
The intensity value of first strip would be = sum of all pixel intensity value = 0 + 0 + ... + 0 (100 times)
Similarly, The intensity value of strip containing whites would be : 255 + 255 + ... + 255 (100 times).
The values would be really large number to deal with, hence we will be normalizing and using the values.

So ultimately, the average intensity value for the ROI would be something like this,
[0, 0, 0, 0, 0, 255*100, 255*100, 0, 0, 0, 0, 0, 255*100, 255*100, 0, 0, 0]

- Lane Finder

To find the lanes in the ROI, we can split the array we got above
[0, 0, 0, 0, 0, 255*100, 255*100, 0, 0, 0, 0, 0, 255*100, 255*100, 0, 0, 0] into two halves.
First half would tell us the first lane and the second half would tell us the second lane.
[0, 0, 0, 0, 0, 255*100, 255*100, 0, ...] and [..., 0, 0, 255*100, 255*100, 0, 0, 0]

With the use of iterator and maximum function, we can find distance of first lane from the 0th point. And similarly we can find the distance of second lane by getting the maximum value from the second part of the array.
The distance is basically the x-axis position of lanes in the 400 pixels width frame.

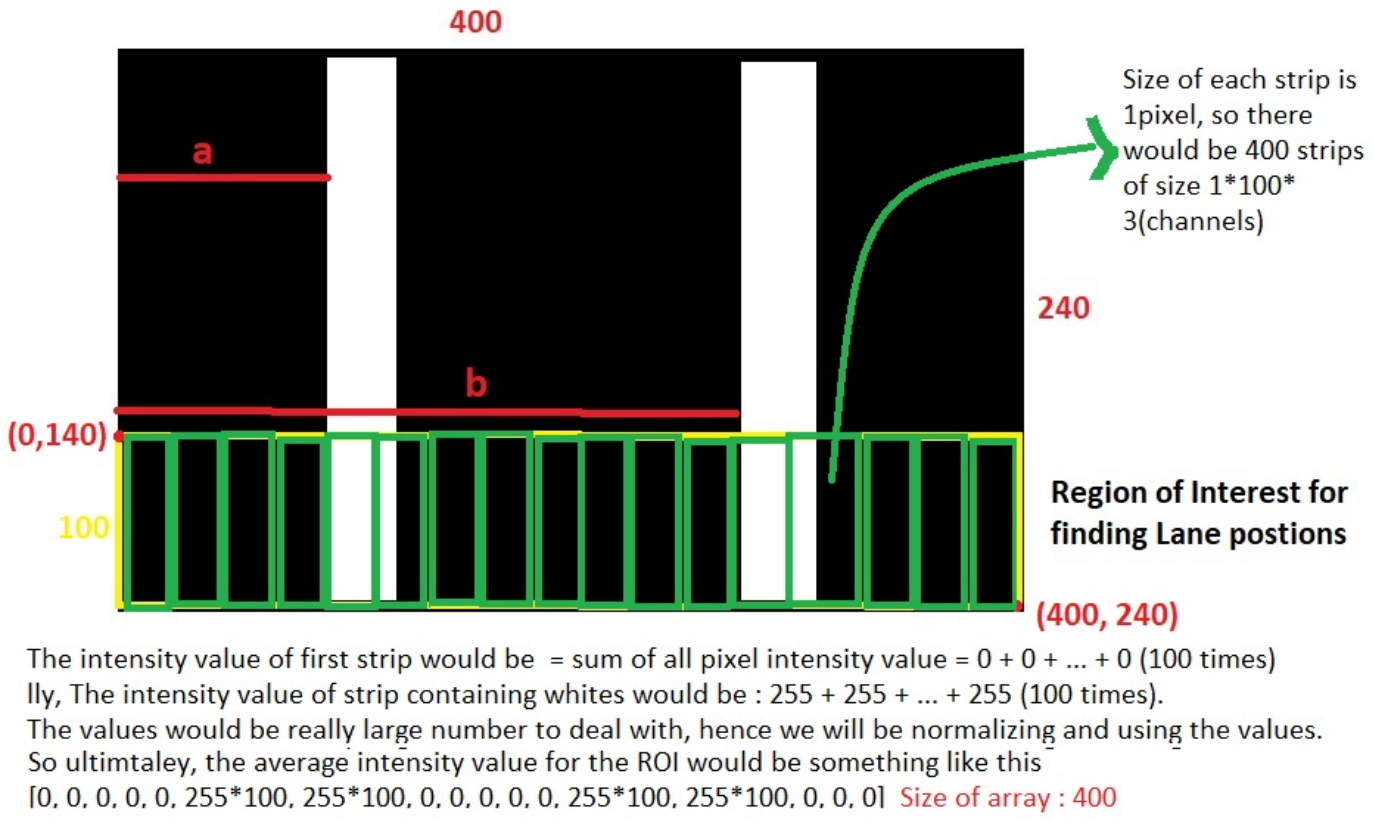

Figure 8: Finding Lane Positions

- Lane Center

  We now know the value of distance of first lane and second lane from 0th position as a and b. So the mid position of lane from 0th position would be calculated as

$$laneCenter = \frac{a + (b - a)}{2} \tag{1}$$

  Now we can also calculate the frameCenter of our frame that would help us calculate the deviation of robo car from the center of the lanes represented by the lane center.

  In the below image Fig : Lane Detection, the left most green line shows the first lane, the rightmost green line shows the second lane. The green line at the center shows the center of the lanes which we obtained. The blue line would show us the center of our frame.

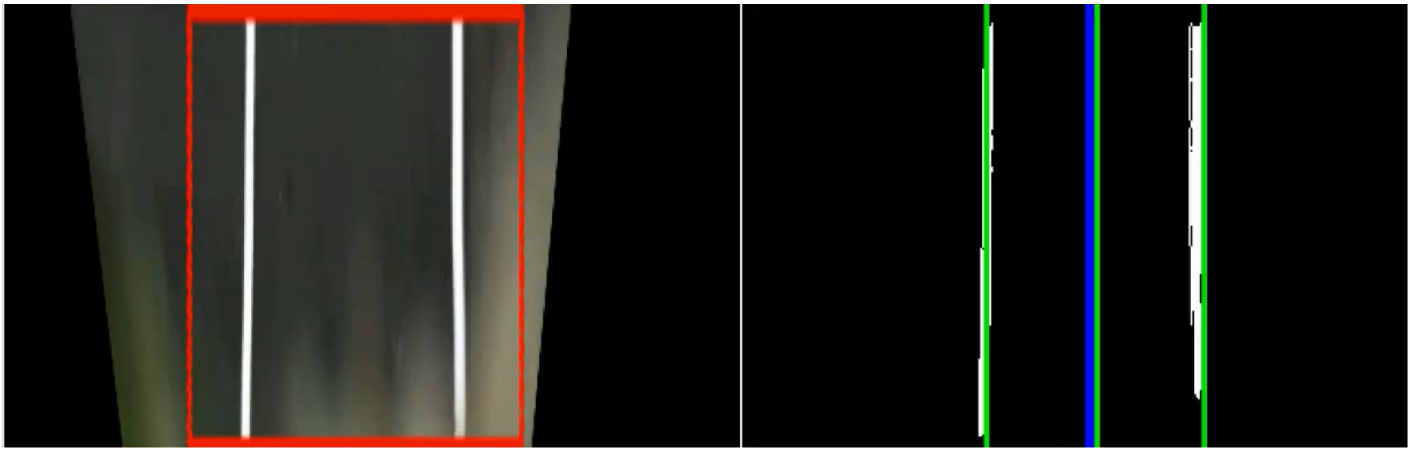

Fig : Perspective view

Fig : Lane Detection

Figure 9: Finding Lane and Frame Center

4.4 Car Movement and Sending Instructions to Slave Device

Now that was all about Image processing and mathematical calculations. Now we have frameCenter and laneCenter, how do we know if the car has to turn left or right.
When lets say the car moves left then our lane center would go to a little right than our frame center. So the value of our laneCenter - frameCenter would be positive which would mean the car needs to go right.
Similarly, When lets say the car moves right then our lane center would go to a little left than our frame center. So the value of our laneCenter - frameCenter would be negative which would mean the car needs to go left. See Figure 10
If the value of result is say between [0, 10] then we may have to move right by a little margin and hence we will give less torque to our motors and say if this value is very high then we would be giving a higher torque to our motors.

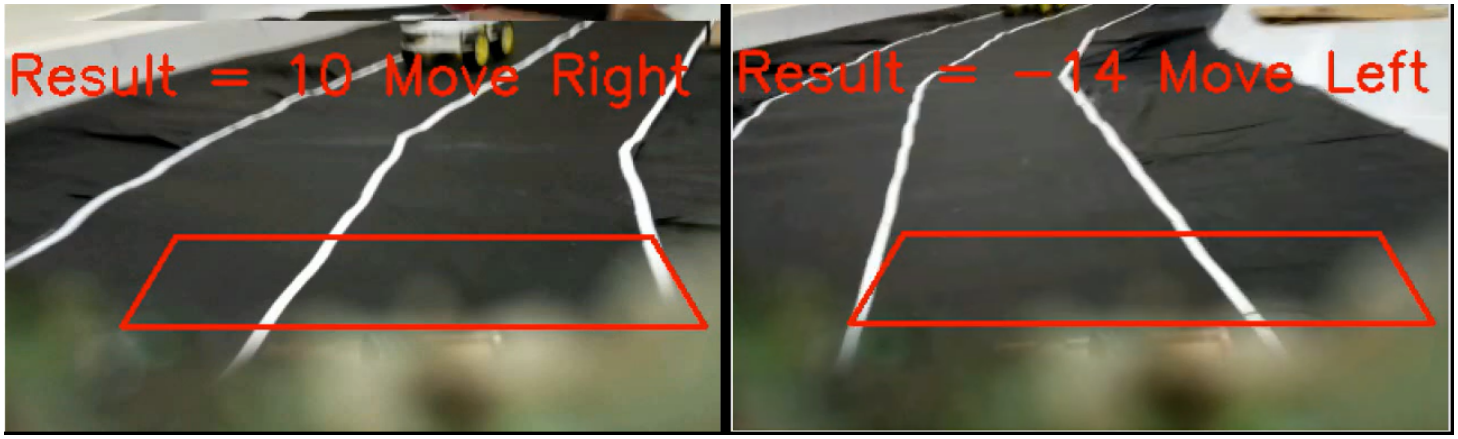

Figure 10: Car Movement

4.4.1 Sending Instructions to Arduino Slave Device

Based on the result we calculated we will send the instruction to Arduino to move left/right/forward. We will connect the digital pins of RPI to digital pin of Arduino to establish a communication link. We can also use serial communication to send the data to Arduino.
We will be using the 4 digital pins using which we can write 16 different possible combination on our Arduino. Here we are going to use the WiringPi pins of our RPI. Keep in mind that WiringPi pins are totally different than the digital pins configuration of our RPI. We will be using WiringPi library to program our RPI digital pins. See Figure 11.

From the Arduino UNO we are using 0,2,3,4, pins and from RPI we are using 21,22,23,24 wiring pi pins.
Note : Make sure Arduino and Raspberry are connected with the same ground pin.

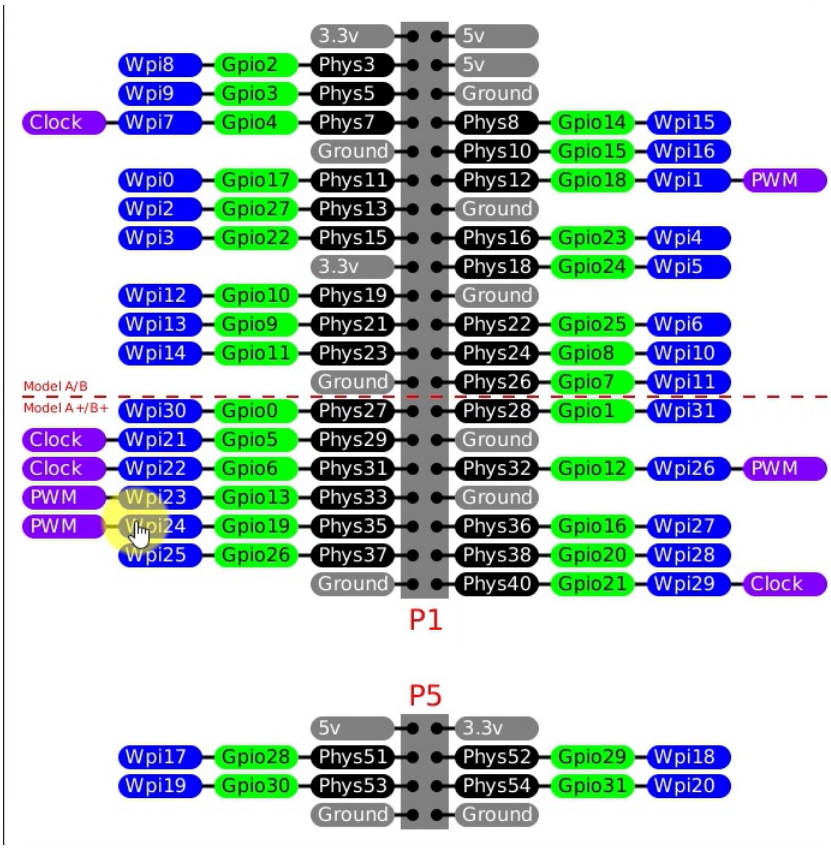

Figure 11: WiringPi Pin Configuration

4.5 Lane End Detection and UTurn Implementation

We can add a white strip to the lane end so that we will start getting higher intensity values when we reach to the end of the lane, we can store the intensity values in a different vector and for each frame we will also calculate the intensity value sum, if the car is still in the lane the intensity values will be lesser but when horizontal white strip is encountered at the end of the frame, we will start getting higher intensity values. On hit and trial we found that it was 4000 for the lane end and once we reach that number we can execute our Lane end routine to make a Uturn.

Now for each frame apart from finding the lanes, we will parallely be also checking if the frame contains lane end.
For the lane end part we are going to process our entire frame and divide the frame into 400 matrices of width 1 height 240 and 3 channels. Similar to above we add the intensity values in each of our 1*240 matrix. At the point where our car finds the white strip this value will be the highest. With the hit and trial method we find this value and in the code we add the condition to take UTurn whenever this large value of laneEnd is encountered, once we encounter this large value we take UTurn and then reset the value.

Figure 12: Lane End

5 Object Detection and Machine Learning

Now the next stage of our project would be to identify stop sign, obstacles and traffic signal in our path. This part of our project falls into the category of object detection. We will be using Machine learning to detect objects in the frames.

This task can be done using several techniques, but we will use the haar cascade classifier in OpenCV. OpenCV comes with 2 types of classifier Haar and LBP. Haar is generally more accurate while LBP is faster to train.

The way Haar cascade works is it tries to look for features in our image, and it looks for this features in different layers. So at top layer it will be looking at large features that span nearly the whole image window down to bottom layer where it will be looking at really fine details. This makes the end model fast enough to detect objects in real time because it can quickly reject areas of image that fail to match features in the top most layer and can spend more time in analyzing areas of images that are good candidates by studying those finer details. OpenCV provides the support for Haar Cascade model already where we will be needing just the data for which we are going to train our model on.

The main reason for me to choose Haar Cascade was that the haar-cascade algorithm is light and works in real-time with a perfect frame per second. I tried object detection using YOLO as well but the latency in processing the results increased.

We needed something that would take less computing/processing time as we had to send signals to our RPI in the real time as well and any delay in processing the frames would impact the signals being sent to the slave device Arduino and thereby delaying processing of next frame.

5.1 What are Haar Cascades?

Haar cascade is an algorithm that can detect objects in images, irrespective of their scale in image and location. This algorithm is not so complex and can run in real-time. We can train a haar-cascade detector to detect various objects like cars, bikes, buildings, fruits, etc.

Haar cascade uses the cascading window technique, and it tries to compute features in every window and classify whether it could be an object. Sample haar features traverse in window-sized across the picture to compute and match features. Haar cascade works as a classifier. It classifies positive data points → that are part of our detected object and negative data points → that don't contain our object.

Some advantages of this model are :

- Haar cascades are fast and can work well in real-time.
- Simple to implement, less computing power required.

Some disadvantages of this model are :

- Haar cascade is not as accurate as modern object detection techniques are.
- Haar cascade has a downside. It predicts many false positives.

5.2 Training of the model

We use our Raspberry Pi V2-8 camera to capture the positive and negative images of our training data. I have created a simple program in C++ that helps us capture images on press of a keyboard button. See CaptureImages.cpp.
We used OpenCV's integrated annotation tool to annotate the object to be detected from our positive images. After annotating the images we use opencv_createsamples to create the positive vector which will be needed later by the training model as input for positive images.
opencv_traincascade.exe is the opencv trainer program that needs the path of the folder where we want to save our result. Next it needs the vector file with all the positive sample vectors. Then we give the path of the negative images folder. Next we need to specify the width and height of the detection window. The value is same as we used to create our vector file. Basically this field tells the classifier to not to identify objects with size less than w*h. We then specify the number of positive and negative images. numOfStages tells the classifier the number of stages to train. More stages would more time and over training. The model then can be trained as :

opencv_traincascade.exe -data cascade/ -vec pos.vec -bg neg.txt -precalcValBufSize 6000 -precalcIdxBufSize 6000 -numPos 200 -numNeg 1000 -numStages 12 -w 24 -h 24 -maxFalseAlarmRate 0.4 -minHitRate 0.999

5.3 Loading the model

After training the application created the xml file which cotains the parameters of our trained model. We can use opencv *CascadeClassifier* class to load the XML created in the cascade folder after the training.

5.4 Stop Sign Detection

To train our ML model on stop signals we created a stop sign with 7cm in diameter.

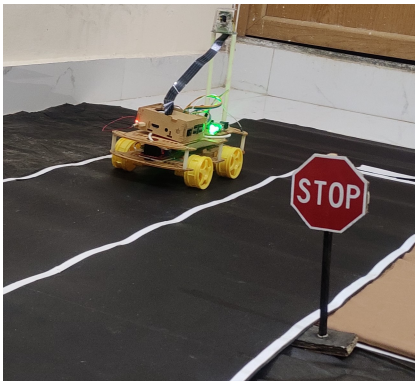

Figure 13: Stop Signal

Positive Samples : Captured around 100images of my stop sign from different angles and distance. Manually annotated all the images.

Figure 14: Positive Sample of Stop Signal

Negative Samples : Captured around 600 images of surrounding without the stop sign.

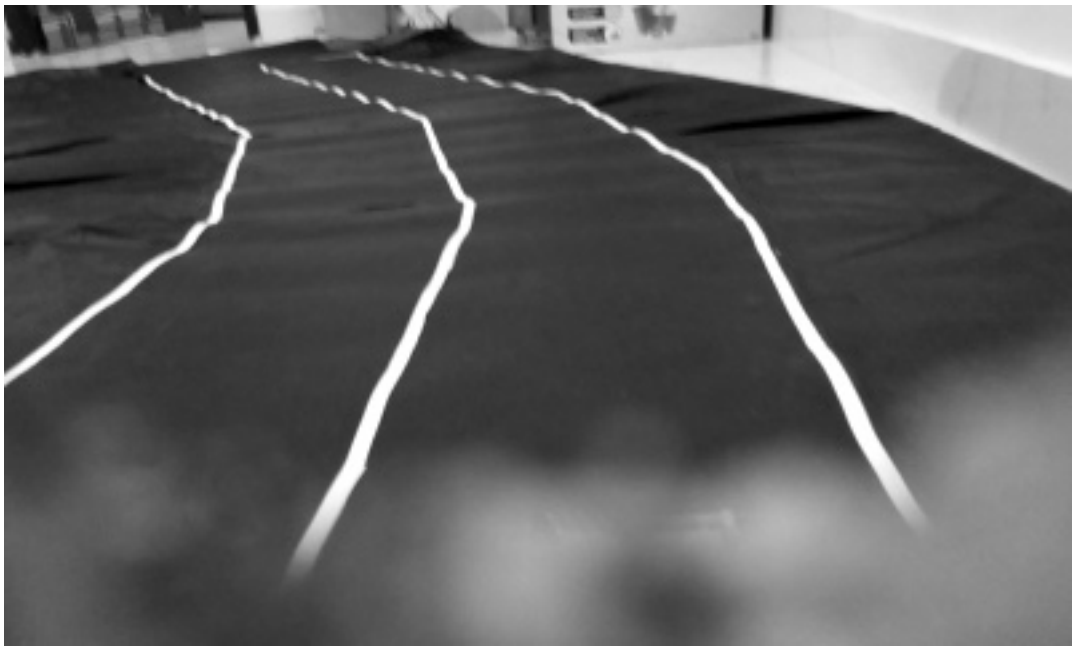

Figure 15: Negative Sample for Stop Signal Detection

To view the captured images, check *CaptureImages* folder.

5.5 Obstacle Detection

To detect obstacle in the path we used another RoboCar as an obstacle.

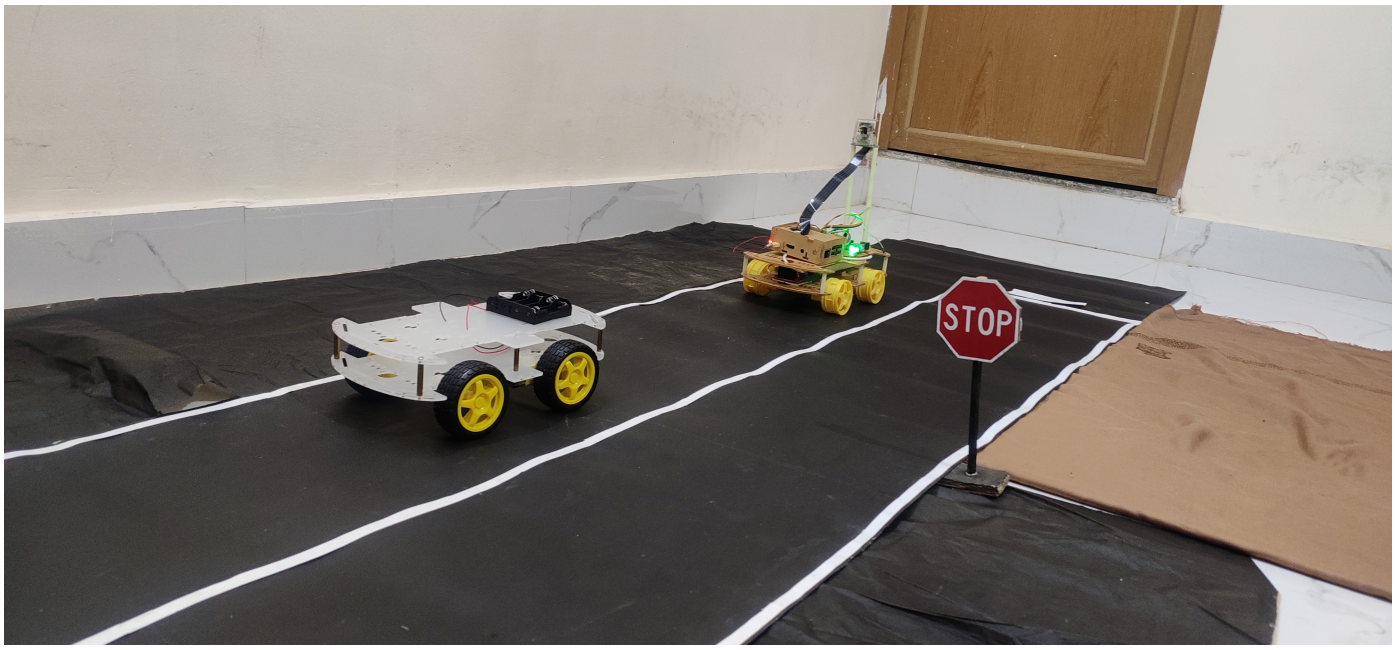

Figure 16: Obstacle in the Path

Positive samples : Captured around 100 images of the object from different angles and distance. Manually annotated all of them.

Figure 17: Positive Images for Obstacle Detection

Negative Samples : Captured around 600 images of surrounding without the obstacle in path.

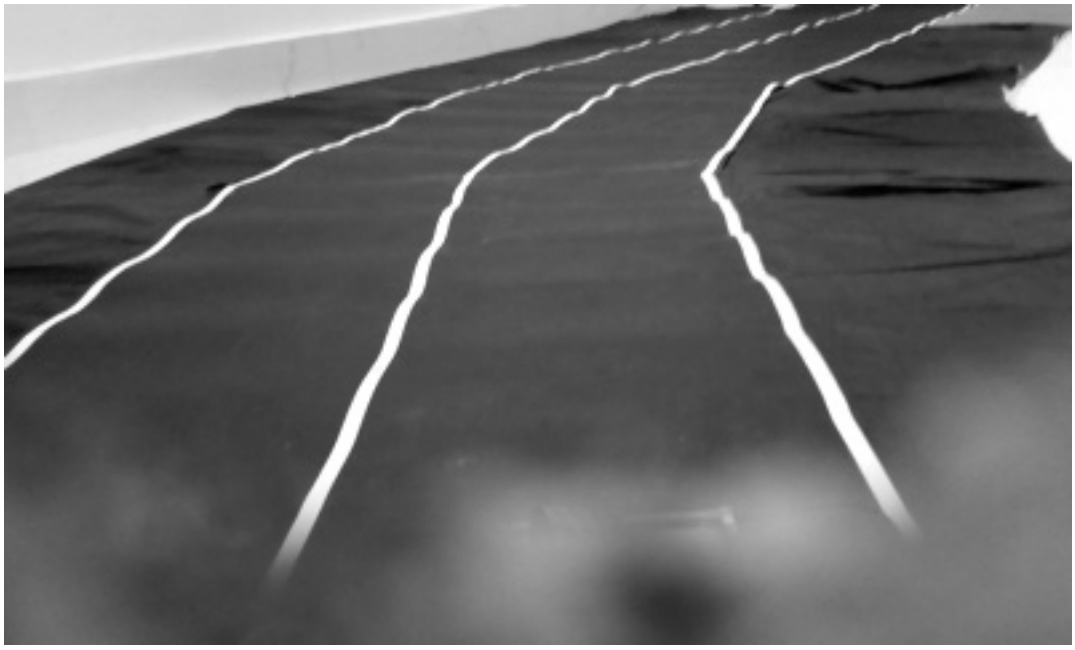

Figure 18: Negative Image for Obstacle Detection

To view the captured images, check "CaptureImages" folder.

5.6 Traffic Signal Detection

Created a simple traffic light model using the LED lights. Trained the model to identify traffic light with red light or no lights as the positive images. A simple white box using white paper was created around the green led light to make our ML model robust since traffic light board was all black and wasn't giving us a better accuracy with just the LED lights.

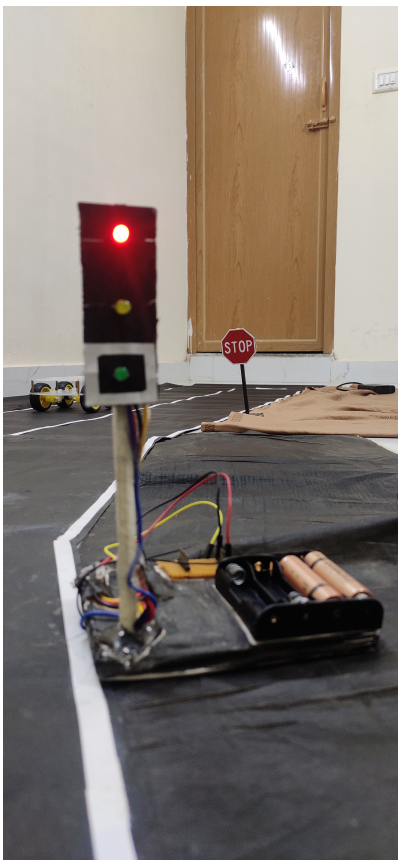

Figure 19: Traffic Signal

Positive Samples : Captured around 50 images of Traffic light with all LEDs off and 50 images with red light on.

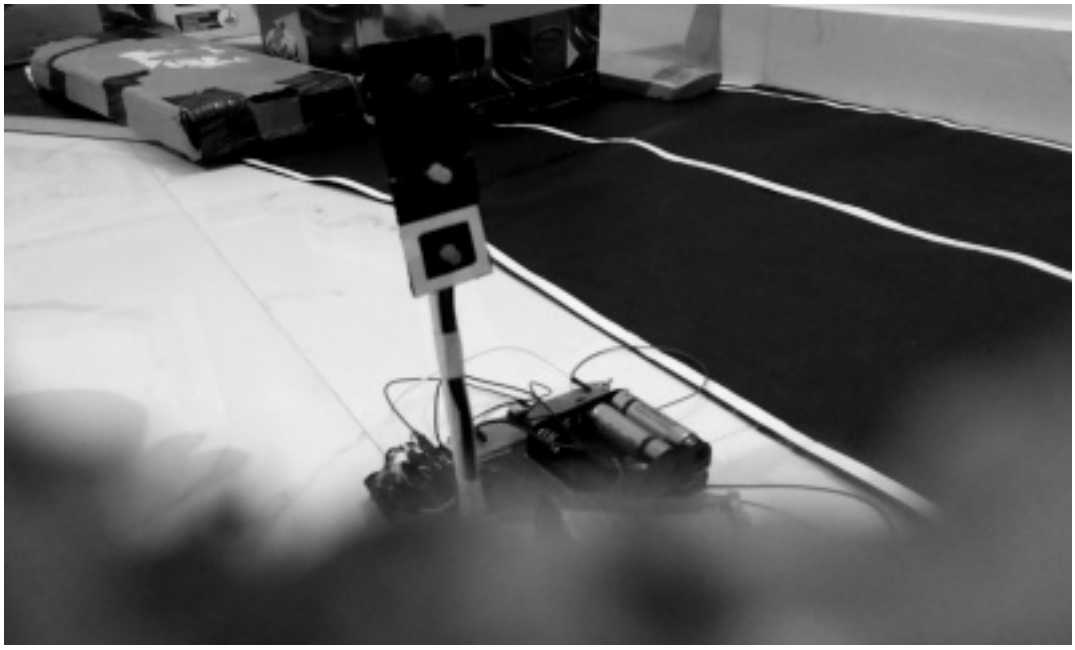

Figure 20: Positive Image of Traffic Signal With No Lights On

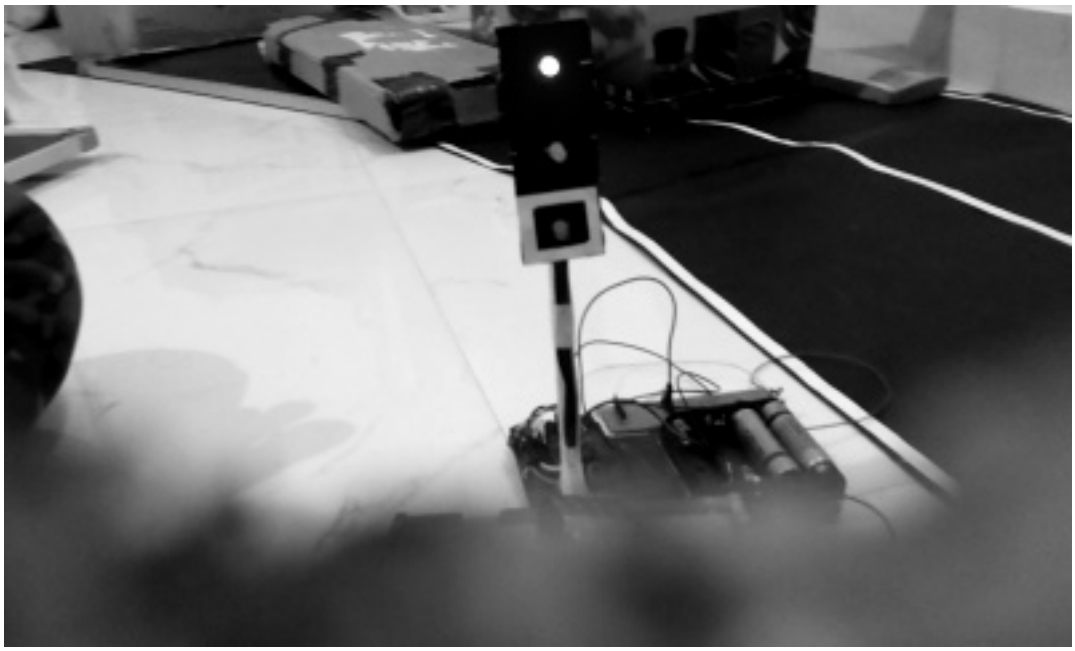

Figure 21: Positive Image of Traffic Signal With Red Light On

Negative Samples : Captured around 100 images of traffic light with the Green LED on and around 600images of surrounding.

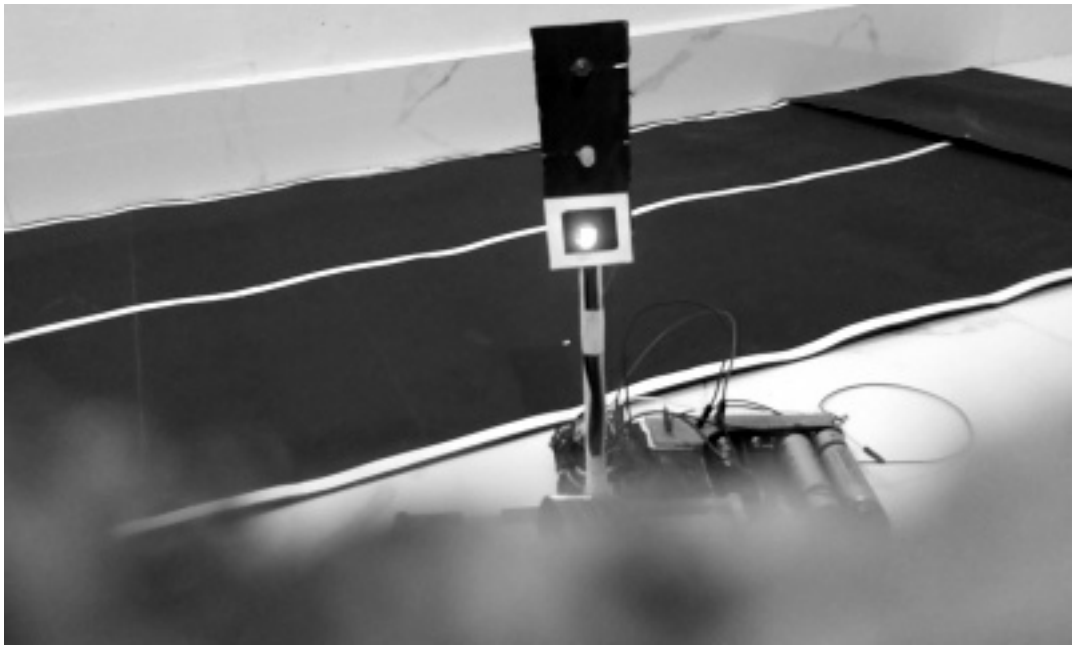

Figure 22: Negative Image for Traffic Signal Detection With Green Light On

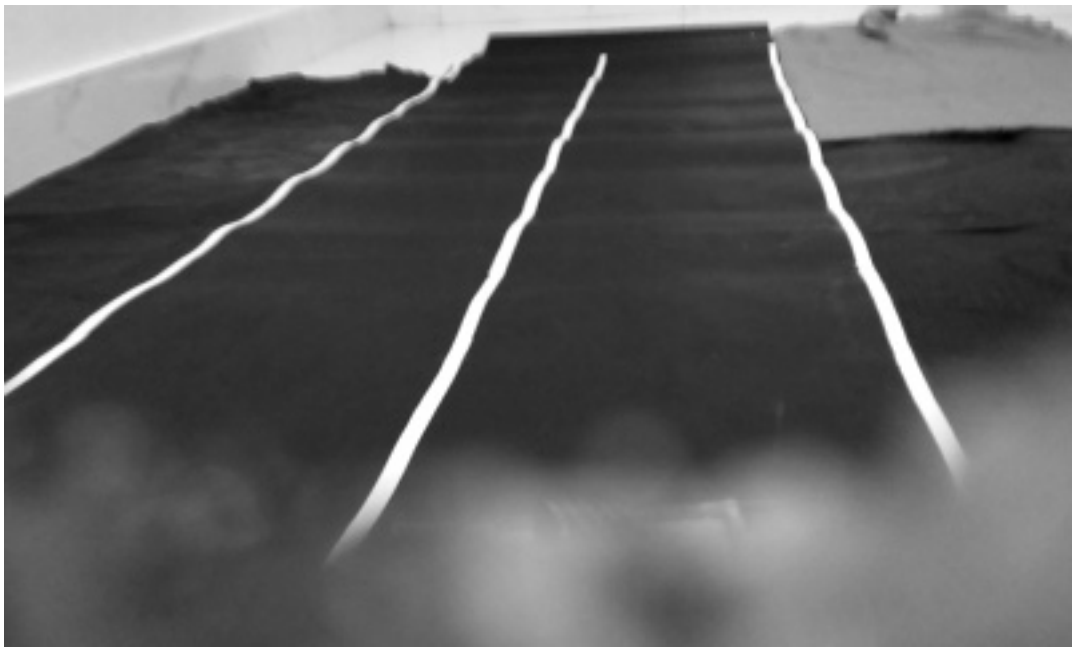

Figure 23: Negative Image of Surrounding for Traffic Signal Detection

To view the captured images, check "CaptureImages" folder.

5.7 Distance Estimation of Objects

There are multiple ways in which distance could be estimated, here we used a simple vanilla approach by plotting the pixel width of the detected area and the actual distance between camera and object measured by a scale.

We take at least 15-20 data points of pixel width value and the corresponding distance value and pass it through a gradient descent algorithm to get the approx value of m and c.

Using sklearn pre-built LinearRegression model to find m and c values.

```
from sklearn.linear_model import LinearRegression

regressor = LinearRegression()

X = np.array([x1, x2, ...])
y = np.array([y1, y2, ...])
# train the model
regressor.fit(X, y)
# print the value of m and c
print(regressor.intercept_)
print(regressor.coef_)
```

Now that we got our intercept and coefficient values, we can put the values to our linear equation formula. This can quite literally be plugged in into our formula from before:

$$distance = width * coef_+intercept \tag{2}$$

To calculate the distance value manually using the python program, we can use

```
def calc(slope, intercept, pixelWidth):
    return slope*pixelWidth+intercept

distance = calc(regressor.coef_, regressor.intercept_, pixelWidth)
print(distance)
```

6 Results

Following the hardware and software specifications as mentioned in the system design, the car was ready for a test run. The camera module was fully functional and the 'result' variable was being used to keep track of the car's line of sight and hence would act as a measure of how accurately the car was able to traverse the central line of the lane. Few images taken during the test run of the car have been embedded here. Further, images showing the car during its test run have also been added, as the car was successfully able to traverse the lane without colliding with any obstacles. It was successfully able to detect the stop signal and stop for few seconds. When the car found the traffic signal to be red it stopped until the traffic signal turned green again. The car was able to take UTurn on lane end and was following the lane in a circular manner.

Full video of the working of the car with behind the scenes working can be found in the youtube playlist

Figure 24: Car Moving along the Track

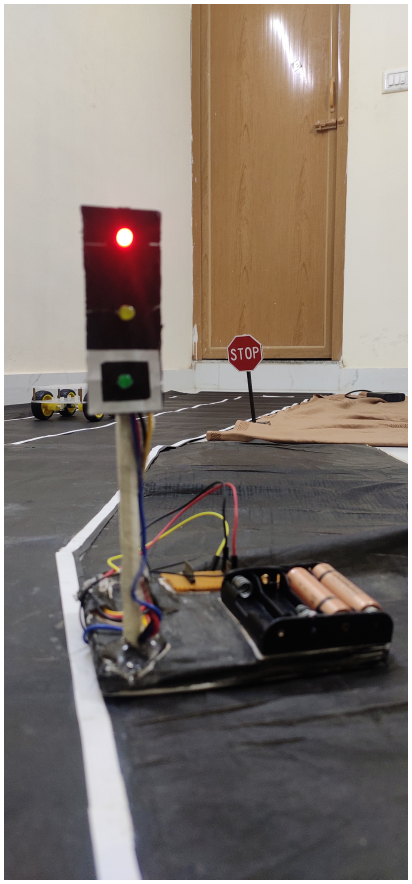

Figure 25: Track with Stop Signal and Traffic Light

References

[1] OpenCV CascadeClassifier
https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html

[2] Training a Cascade Classifier
https://learncodebygaming.com/blog/training-a-cascade-classifier
https://www.youtube.com/watch?v=XrCAvs9AePM&t=1352s

[3] Tesla and AVs
mindy-support.com/news-post/how-machine-learning-in-automotive-makes-self-driving-cars-a-reality

[4] Line following Robot
https://circuitdigest.com/microcontroller-projects/arduino-uno-line-follower-robot

[5] Lane Detection using OpenCV
https://www.analyticsvidhya.com/blog/2020/05/tutorial-real-time-lane-detection