MANCHESTER
1824
The University of Manchester

Artificial Intelligence for Autonomous vehicle navigation in Mini Town with Web Server UI

Third Year Individual Project – Final Report

April 2023

MICHALIS IAKOVIDES

1062383

Supervisor:

Dr Alexandru Stancu

Contents

Word count: 8116

Abstract

One of the main reasons for developing self-driving vehicles is to offer safer transportation and decrease car accident rates. This project explores the algorithms used in modern autonomous vehicles and covers numerous concepts including computer vision and machine learning.

This project focuses on designing algorithms that can assist a mini vehicle to navigate without human intervention. The approach involves testing navigation and recognition algorithms in a 3D robotics simulation and then transferring and fine-tuning the algorithms on a real mini-vehicle. Using a simulation allows for quicker development and various scenarios can be tested. The results of the project prove that the selected algorithms can navigate a vehicle in a town. However, building such vehicles requires hardware that can handle the computational power needed.

Declaration of originality

I hereby confirm that this dissertation is my own original work unless referenced clearly to the contrary, and that no portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

1 Introduction

1.1 Background and motivation

The term self-driving car or autonomous vehicle has increasingly appeared on tech news websites, with leading companies participating in a race to build the perfect intelligent vehicle. It is a fact that smart vehicles with different autonomy levels [1] are becoming more commonly seen on the streets these days. During the past decade, the concept of self-driving or autonomous vehicles was a far-future idea, but due to technological evolution like machine learning, it came to life in recent years. This kind of vehicle requires travelling many miles to train algorithms before being able to offer a safe drive. The construction of an autonomous vehicle requires an effective combination of software and hardware that can work together and process data in real-time. The main operation of a self-driving car is being able to map its surroundings using different types of sensors like cameras, lidars, and radars and identifying the best path while avoiding obstacles. The industry is working on making those vehicles smarter by integrating different algorithms and safety systems to prevent accidents and offer safer transportation. In 2019, there were over 1400 self-driving vehicles in test by over 80 companies across the US [2].

Most companies are facing a common challenge when working on autonomous vehicles: building the perfect navigation system. Many factors are holding back, such as weather conditions, sensor limitations, poor street conditions, or traffic regulations. The automotive industry is taking advantage of the computational resources available and tests their vehicles' behaviour in simulated worlds. Simulators are preferred because they offer safe testing without risking accidents, reduce equipment costs, allow for data collection that can improve the algorithm's performance, and enable scalability by testing a vast number of scenarios.

The motivation behind this project is to build a self-driving vehicle with a web app that allows data visualisation. This can be commercialised since automotive companies will be able to track and collect data or interact with the vehicle remotely. The web app could also be used for remote vehicle inspections by the manufacturer and to assess the vehicle's condition. The project approach was to program and experiment with a simulated vehicle using a 3D robotics simulator and transfer and test the algorithms on a real mini vehicle. The navigation algorithm follows the same principle that a real self-driving vehicle use in a real-world environment. In terms of performance improvements, simulated worlds can accelerate the machine learning process and improve the vehicle’s navigation performance, by testing different algorithms and exploring their limitations.

1.2 Aims and objectives

This project aims to build a fully functional mini self-driving vehicle capable of navigating in a mini town while sending data to a web server for user observation. The project includes the following objectives:

- Gain familiarity with installing and using UbuntuOS and explore the capabilities of ROS and Python in a simulated and real-world environment.
- Understand how computer vision works and implement navigation, detection, and recognition algorithms.
- Understand how convolutional neural networks work with images and ways to increase their accuracy.
- Understand the importance of using high-quality data for training an AI model and how to validate its accuracy.
- Gain experience working in Gazebo Simulator and explore its capabilities.
- Gain experience working on a simulated robot and replicate its functionality on a real robot.
- Identify suitable control algorithms that provide smooth navigation and fast response.
- Build a web app with GUI that can update its data in real-time.
- Identify ways to send data to a web app without compromising the robot’s performance.
- Gain experience using web development programming languages/frameworks and designing a user-friendly interface.
- Gain hands-on experience with the Nvidia Jetson Nano board and explore its capabilities in embedded systems.
- Explore the limitations of software and hardware when building an autonomous vehicle and the alternative methods used based on these limitations.

1.3 Report structure

The report consists of five chapters:

- Chapter 1: Outline the challenges of building a self-driving car and description of the aims and objectives of the project.
- Chapter 2: Review of past research papers on self-driving cars and techniques that have been used.

- Chapter 3: Describe the approach and algorithms used to design, build, and test the mini vehicle.
- Chapter 4: Discuss and assess the performance of the robot in a simulated and real-life environment.
- Chapter 5: Summarise the achievements and challenges of the project and propose possible features to be implemented in future work.

2 Literature review

2.1 Introduction

Artificial Intelligence is a rapidly growing field that has been applied to various domains, including autonomous vehicle navigation. AI techniques such as machine learning, computer vision, and deep learning have been widely used to improve the perception, decision-making, and control capabilities of vehicles. Below, the research papers propose various algorithms, which are commonly used in commercial autonomous vehicles.

2.2 Lane Detection Techniques

Vision Based Lane Detection for Self-Driving Car [3] applies basic image processing on real car footage, where there are both yellow and white lines. In this paper, HSV and grayscale were applied to the original image to extract the lane lines. Then both extracted images are combined, and the canny edge detection algorithm is applied followed by a region of interest operation. This paper's approach to lane line fitting relies on inverse perspective transformation to achieve a top-down view. Windows are made around detected pixels and based on their x coordinate, lanes lines are defined using polynomial fitting. As a result, this method had increased accuracy in detecting lanes in complex environments.

Real-Time Self-Driving Car Navigation Using Deep Neural Network [4] used a manually controlled mini car and collected images of the lane using an onboard camera while storing the steering angle of the car. The convolutional neural network (CNN) requires a massive amount of data to have accurate results. To overcome the time spent on collecting data, the authors used data augmentation technique to generate more images like the ones they already collected. The generated and real data were then used to train a convolutional neural network. The results of this approach had a model with high accuracy and the car was able to navigate, but it could use a simulator for testing and train the model using data from a database to increase the performance.

2.3 Traffic Light Recognition Techniques

Traffic Light Detection and Recognition for Self-Driving Cars Using Deep Learning [5] suggests the use of a pre-trained Faster Region-based Convolutional Neural Networks (R-CNN) Inception-V2 model to detect and recognise the traffic lights. The author processed and labelled the images from a traffic light dataset and trained the model. The resultant model took an extensive amount of time to be trained but it had high accuracy.

Traffic Lights Detection and Recognition Method Based on the Improved YOLOv4 Algorithm [6] proposed the use of the YOLO model. The authors trained the model on the LISA traffic light dataset. This paper used the Improved YOLO-v4 model, which has an uncertainty prediction mechanism that is applied to smaller boxes on the image. The resultant model had a higher accuracy compared to other YOLO models.

2.4 Traffic Sign Classification Techniques

Yolo V5 for Traffic Sign Recognition and Detection Using Transfer Learning [7] paper is based on a YOLO v5 model. The dataset used to train the model is the GTSRB (the dataset used for this project). As a result, the model was able to detect and classify the multiple traffic sign in the same frame in real-time with high accuracy.

2.5 Summary

The above research papers were examined, and some techniques were adopted for this project. The purpose of the research was to get a broader view of what had been tried and the limitations of each approach.

3 Methods

This section of the report provides a comprehensive overview of the project approach and outlines the techniques, software, and hardware tools used to achieve the project's objectives.

3.1 Introduction

The project consists of a variety of advanced and complex concepts in the robotics field. This means that Continuing Professional Development (CPD) was crucial in the project's learning curve and progress. Some fields that the project incorporated were the following: programming, computer vision, convolutional neural networks, web development, and control theory. During the early stages of the project, a Gantt chart (Appendix A) was created to list the tasks throughout the project's timeline. Various online resources, including research papers, YouTube videos, GitHub repositories, community platforms, and blogs were researched to understand the project's concepts. The information gathered from these sources was carefully combined to contribute to the development and selection of appropriate software (algorithms, techniques) and hardware components. The objective of this project was to build a real autonomous mini-vehicle that can navigate in a mini-town using detection, recognition, and steering control algorithms and transmit real-time data on a web application. The project approach involved programming and testing algorithms on a digital twin of the real mini vehicle in a 3D robotics simulator. Features like lane detection, traffic sign classification, traffic light recognition, steering control and navigation algorithm were implemented to enable self-driving functionality. Once the simulated vehicle had achieved the intended performance, the assembly of the real mini vehicle followed. A further tuning process was necessary so that the robot could cope with real-life mini-town requirements. The final component of the project was the development of a web application that displayed real-time data. The application was built using a combination of programming languages and frameworks.

3.2 Operating System

For this project, Ubuntu OS (version 20.04.5 LTS) was chosen over Windows OS since the 3D robotics simulation software (Gazebo Simulator) is officially available only on Ubuntu OS [8]. The installation of Ubuntu OS was made as dual booting by installing the OS on a USB stick and following the official Ubuntu installation guide [9]. It is worth noting that running Ubuntu OS through a virtual machine is not recommended due to the high computing power required by the simulator, which can result in significant delays. One advantage of Ubuntu OS is that Python 3.8.10 [10] is pre-installed, which takes away the need for the Python installation process. Additionally, Ubuntu OS comes with a powerful command-line interface, the Terminal, which provides a convenient and efficient way to run software components and install additional libraries and packages.

3.3 Robotics Operation System – ROS

Robotics Operation System (ROS version) is an open-source framework that is widely used in robotics research and development. For this project, ROS Noetic Ninjemys was used due to its compatibility with the Ubuntu version. Since there was no prior experience with ROS, it was essential to learn the basics. The learning process began with reading the official ROS documentation on the ROS website [11] and watching YouTube videos [12] that demonstrated the steps of ROS tutorials. The ROS framework is available in both C++ and Python, but Python was chosen as the main programming language for the project due to its simpler syntax, abstraction, and numerous libraries for computer vision (Section 3.5.1) and machine learning (Section 3.8.2). ROS was used to exchange data between the Python scripts (.py) and the robot in the Gazebo simulator. Some essential ROS examples that utilise basic concepts that have been studied are ROS Catkin Workspace, ROS package, ROS Node, ROS Publisher and Subscriber, and teleops twist keyboard.

The ROS architecture [13] consists of the following components:

- ROS Master: is responsible to keep track of all the nodes in the network, including their published topics and subscribed topics, and establishes the communication between them.
- ROS Nodes: are executable software components that run Python scripts and communicate with other nodes.
- Topics: are named buses that transmit messages using a publish-subscribe model
- Messages: are typed data containers that can hold information of various types, such as integers, floats, strings, arrays, and custom data types. They are transmitted by each node under topics.

Considering the implementation in ROS, each algorithm explained below had its node and was sending messages over topics to other nodes. The Python scripts were compiled and debugged using a code editor (Visual Studio Code or VSCode) [14]. VSCode provides a user-friendly interface with numerous features such as debugging, syntax highlighting, and auto-completion.

3.4 Gazebo Simulator

The Gazebo simulator [15] is an open-source 3D robotics simulation tool that is used to simulate the behaviour of robots and other objects in real-world situations. Gazebo accelerated the project's process because it was used to test different scenarios on the digital twin of the mini vehicle. The Gazebo user interface allows the user to move around the objects in the 3D environment, in this case moving the robot in different starting positions or moving the traffic lights and signs to test and optimise the algorithm's performance. Since the project was focused on the development of navigation algorithms, the supervisor provided the 3D models of the track and the robot. Additional information about the Gazebo files can be found on the Manchester Robotics Github page [16]. To gain more hands-on experience using ROS with the Gazebo Simulator, some basic examples were followed, such as driving the robot in a straight line or a square shape using open-loop and closed-loop feedback control. These examples provided a better understanding of how Topics, Publisher and Subscriber nodes work and how they could be used to control the movement of the robot.

3.5 Lane Detection Algorithm

The road lane detection feature was critical for vehicle's perception of its environment. The algorithm had to be able to detect lanes in straight or curved lanes, as well as in different lighting conditions. Various online resources [17][18] have been researched to develop a robust algorithm.

3.5.1 Reading Camera Frames

The implementation of lane detection required access to the robot's onboard camera, which is the only perception sensor. Reading camera frames was also required for Traffic Sign Classification (Section 3.8) and Traffic Light Recognition (Section 3.7). Reading and transforming the camera's view was achieved with an open-source computer vision package (OpenCV 4.2.0) [19]. The camera frame was resized to 800x800 pixels, which allowed enough width to capture the lines and enough height to see upcoming traffic signs or lights.

In the simulator, the virtual camera of the robot was reading the camera feed as ROS Image Message. This could not be used for image manipulation and thus a ROS library called CV_Bridge had to be used to run OpenCV commands. The same method was used on the real mini vehicle as the image from the camera was sent to the processor as a ROS Image Message.

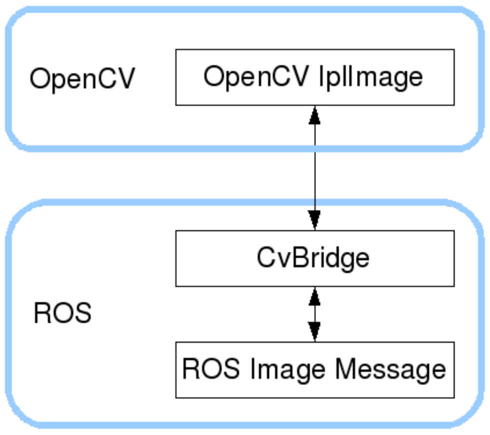


Figure 1: OpenCV interfacing with ROS [20]

3.5.2 Greyscale Conversion

In general, images are represented as matrices, where each element stores each pixel’s values. The image that comes out of the camera is an RGB (red/green/blue) and each pixel is a combination of three intensity values of the three colours. OpenCV converts and stores the image in BGR (blue/green/red) colour space. Working with three colour channels image requires higher computing power and increases complexity as more data is collected and processed. The image has been converted to greyscale which is a single-channel colour image. Greyscale pixels have one intensity value from 0 to 255, which indicates the brightness of each pixel. This reduced the size of the image data and simplified the processing requirements.

3.5.3 Canny Edge Detection Algorithm

Edge detection is the process of identifying sharp changes in the intensity of grey (gradient) between the adjacent pixels. In other words, the gradient is the derivation between the adjacent pixels in the x and y directions of the image. A key point was to reduce the image noise and smooth the image. Image noise can affect the edge detection performance, creating inaccurate edges. The above steps were implemented by the Canny algorithm that was used for edge detection. The algorithm applies a Gaussian blur filter on the greyscale image (Section 3.5.2) with a 5x5 kernel to minimise the image noise. A 5x5 kernel is an array that averages the value of each section of the image array. Once the image is smoothed, the algorithm calculated the gradient to identify edges accurately. An upper and a lower limit on the gradient value was set to extract the edges of the black lane lines.

3.5.4 Region of Interest

This part of the algorithm was aiming to isolate the lane lines from other objects within the frame. This was achieved by defining a region of interest (ROI) in the bottom area of the frame where the lane lines were most likely to appear. The region of interest was formed using a plotting library (Matplotlib 3.1.2) [21] to identify the coordinates of a triangle that bounded the lane lines. All pixels within the region of interest were set to white with values of 255 or 1111 in binary, while the remaining pixels were black with values of 0 or 0000 in binary. The final step was to apply the mask of the region of interest over the canny image (Section 3.5.3) using a bitwise operation. The bitwise operation used was based on the AND logical operation applied to pixel binary values. This operation returned a frame containing only the edges of the lane lines.

3.5.5 Probabilistic Hough Line Transform

The Probabilistic Hough Line Transform technique was used to detect the lines of any orientation using the edge points in the selected area formed (Section 3.6.4). The Probabilistic Hough Line Transform is a modification of the Hough Line Transform that is faster and less sensitive to noise. The main principle of the Hough transform is that all lines are represented using the Cartesian plane, which is given by

$$y = mx + c \tag{1}$$

where m is the gradient of the line and c is the point at which the line crosses the y-axis.

The algorithm starts by randomly selecting points from the image and then applying the Hough Transform. Thus, a new parametric space was considered named Hough Space with c (point of intercept) and m (slope) axes. A line in the cartesian plane is represented as a point in Hough space (Figure 2). Lines with different slopes and axis interception points can form a straight line in Hough Space (Figure 3). Hough Space is divided into a grid containing lines with different slope values, and at every point where there is an intercept, the grid increases a counter (Figure 4). A threshold value that tested the value of the counter was used to adjust the resolution. With the vote-casting method on the grid, the value of m and c were determined, which describes the best-fitting line.

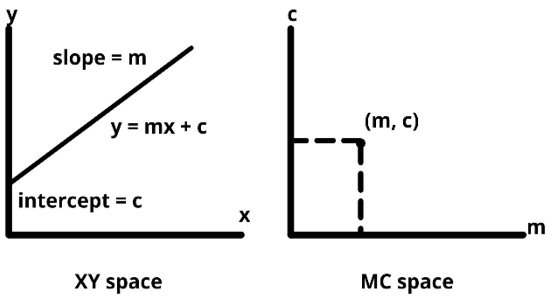
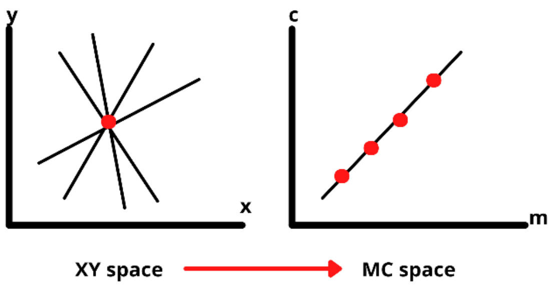


Figure 2: Line represented in XY Cartesian plane (left), Line represented as a point in Hough Space(right) [22]



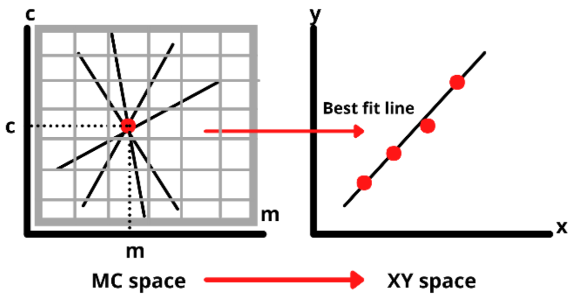


Figure 3: Cartesian plane to Hough Space [22]

Figure 4: Hough Space to Cartesian plane [22]

3.5.6 Lines Classification

The lines detected using the Hough transform have been assigned endpoint coordinates (X1, Y1) and (X2, Y2). These coordinates were used to categorise the lines as either right or left lines. A simple technique was used to classify the lines. Lines with upward slopping that appeared on the left side of 1/3 of the frame width, were classified as left lines. Lines with downward slopping that appeared on the right side of 2/3 of the frame width, were classified as right lines. Moreover, since the lanes on the track consisted of three lines (as shown in Figure 11), the algorithm was ignoring the middle line and focused on the outer lines of the lane that form the lane limits. Once the lines were classified, the average value of both the slope and point of the intercept were calculated to form new lines that are more accurate and avoid wrong line detections. A special case that had to be addressed was dealing with vertical lines, which have an infinite slope. These lines could not be used to calculate the average values, and thus they were excluded.

3.6 Steering Control

The steering control algorithm relies on the efficiency of the lane detection algorithm. The algorithm used the lines detected (Section 3.5.6) and checked the number of lines detected and the value of their slope. As mentioned in section 3.5.6, lines can have either a positive slope (right line) or a negative slope (left line) value. The steering control algorithm was defined using the following equation,

$$angular\ speed = constant * \frac{1}{slope} \tag{2}$$

The value of slope was a key factor in correcting the robot's angle or following a turn because it was used as a divider on the angular speed of the robot.

Based on the number of lines detected, two scenarios have been identified. In the case where only one line has been detected, it meant that the other line went out of the camera's field of view. It was assumed that the robot had an angle offset compared to the middle line or it was approaching a turn. When there is an angle offset, the correction had to be slight to prevent oscillation, but on a turn, it had to increase the rotation rate to allow enough time for the turn. In a case where both lines are detected, it meant that both lines were in the camera field of view, and it was assumed that the robot was in the middle of the lane and can drive straight.

The resulting operation of the steering control algorithm was adjusting the robot's driving angle by turning left or right.

3.7 Traffic Light Recognition

Traffic lights are vital to every city since they ensure traffic control. Building an autonomous car requires a reliable and accurate traffic light recognition feature as it needs to follow traffic rules.

3.7.1 HSV Conversion

The conversion of the BGR colour space to the HSV colour space was necessary for efficient traffic light detection. Both BGR and HSV are colour spaces, but they differ in how they represent and manipulate colours. The initial step was to convert BGR to HSV colour space. In the BGR colour space, a colour is represented by a combination of three colours (Red, Green, Blue). The BGR to HSV conversion was necessary since BGR has limited ability in detecting colours with different brightness and shade levels that can be found in a real-world environment.

HSV was chosen as it allows a straightforward colour selection. The HSV colour space represents colours in terms of hue, saturation, and value. Hue represents the colour tone, saturation refers to the intensity of the colour, and value denotes the brightness of the colour. The hue, saturation and value ranges were adjusted by trial and error to target specific colour shades and hence can cover a wider range of the same colour. The camera frame was captured as a BGR colour space image and converted to an HSV colour space image using OpenCV. In this case, the entire frame was considered since traffic lights could be placed in different positions (middle, right side of the frame etc).

3.7.2 Masking

A crucial aspect that enhanced the algorithm’s reliability was its ability to differentiate between traffic lights and other objects or signs in the camera frame that had the same colour. To solve this challenge, a masking layer was created by setting an upper and a lower limit based on the hue value of the HSV colour space image for each colour (red, yellow, green). These limits were used to generate a mask that was overlaid on the camera's HSV-converted image. The original camera frame and the masking image were then combined using a bitwise operation (as described in section 3.5.4), which highlighted only one colour at a time and ignored the rest of the frame.

3.7.3 Circle Hough Transform

The masking method was able to isolate one out of three colours, but detecting the colour alone is not enough to identify a traffic light. The next step was to detect circular shapes, which are typically used for traffic lights. The detection of the circles was carried out by the Circle Hough Transform operation which is based on basic circle theorems. To begin with, all circles can be represented using the equation in the Cartesian plane, which is given by

$$(x-a)^2 + (y-b)^2 = r^2 \tag{1}$$

Where a and b are the x and y coordinates of the circle's centre and r is the radius.

In this case, the radius of the traffic lights could vary since it depended on the distance between the robot and the traffic light. The range of possible radii was determined by inspecting how the distance affects the detected range. A 2D parametric space, like the one used in Section 3.5.5, was considered named Hough Space with a (x-coordinate of the circle) and b (y-coordinate of the circle). With that in mind, the algorithms applied Circle Hough Transform for each radius in the range using the camera frame as input.

Firstly, it detects the circumference of the circle using the Hough Line Transform. Each point on the circumference is represented as a set of points (circle) in Hough Space. This process is repeated for all the points of the circle and all possible radii. With the vote-casting method on the Hough space, the point with the most circle interception describes the centre and radius of the detected circle.

However, the Circle Hough transform requires high image detail to detect the circumference of the circle accurately and avoid false detections. To overcome this limitation, a median blur was applied to the image to reduce the noise.

From a programming perspective, the Circle Hough transform returns a list containing the circle parameters (centre coordinates and radius) of the traffic light it detects. In this case, three variables (lists) were used corresponding to each colour for storing the circle parameters. Then, a condition statement checks whenever a variable holds a value and returns its colour accordingly.

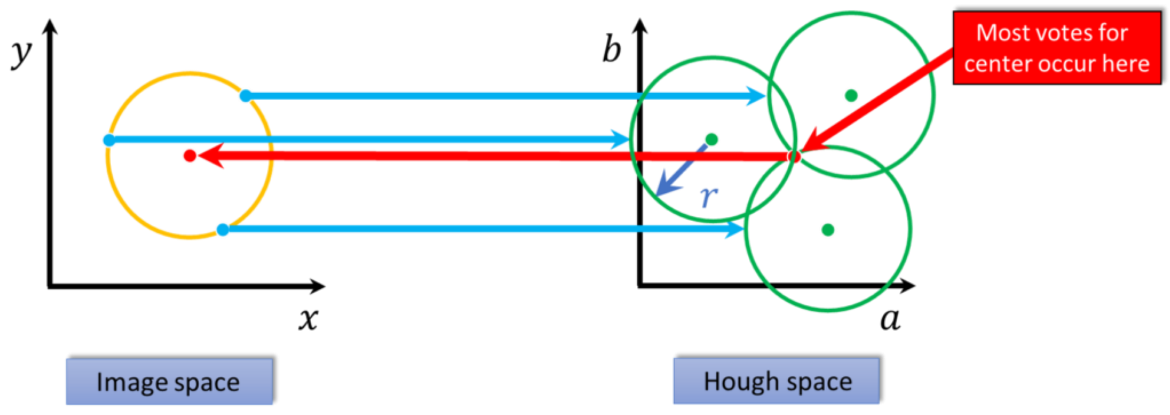


Figure 5: Cartesian to Hough Space mapping of a circle [23]

3.8 Traffic Sign Classification

Traffic signs are essential for traffic control and prevent accidents. A traffic sign recognition feature had to be implemented to mimic the behaviour of commercial autonomous vehicles and obey the traffic rules.

3.8.1 Data Management

Data was an important aspect in developing the traffic sign classification feature. Building an accurate model requires numerous images of traffic signs from different angles and lighting conditions to build an accurate model. The dataset used is called GTSRB (German Traffic Sign Recognition Benchmark) [24] and had been downloaded from an online platform for data scientists (Kaggle). Research had been obtained on various datasets, but the selected dataset has lots of downloads and holds a CC0: Public Domain licence, meaning that it can be used without permission. It contains 50000 labelled images of more than 40 different traffic signs. However, to ensure that the traffic signs could be tested in both the simulator and real-world environment, a data cleaning process was conducted. The dataset also included a labelling data file for the traffic signs in the form of a Microsoft Excel (.csv) file.

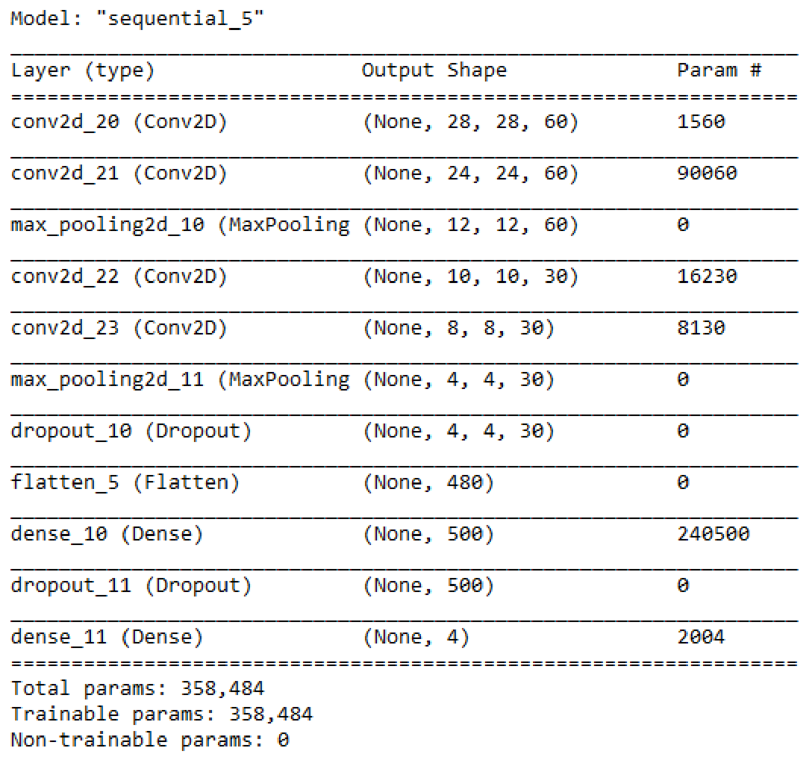
Figure 6: Examples of the traffic sign images from the dataset

3.8.2 Model Training

The traffic sign classification algorithm was trained using a sequential convolutional neural network (CNN). Since there was no background experience in CNNs, an online free course was studied to get the appropriate knowledge [25]. CNN is a deep learning algorithm used in image-related tasks like traffic sign recognition. The basic principle of CNNs is based on human vision, which extracts visual features from an image. The CNN model was built using an open-source machine learning framework (Tensorflow 2.6.0) [26] deep learning framework (Keras 2.6.0) [27]. The images in the dataset were different in size and image processing was made to downscale to 32x32 and apply greyscale conversion using OpenCV. Then they were split into three subsets: training images (60%), validation images (20%) and test images (20%). The CNN was trained using 20 epochs, 445 steps per epoch and a batch size of three. An epoch is a complete pass through the entire training dataset. The steps per epoch refer to the number of batches of sampled images during one epoch. The batch size is the number of samples that are processed together.

The CNN architecture consists of multiple layers. The model starts with two convolutional 2D layers, with 60 filters and 5x5 kernels which performed 2D convolution on the input image and extracted patterns. The number of filters (60 in this case) determines the number of feature maps generated for each layer. The next layer is a Maxpooling 2D layer that downscaled the information extracted by the convolutional layers. It used a 2x2 pool size that was used to pool the maximum value per patch and created a new 2x2 feature map. Two more convolutional 2D layers with 30 filters and 3x3 kernels were added to extract more complex features. Another Maxpooling 2D layer was concatenated. A dropout layer with a rate of 0.5 was added by randomly dropping out (setting to 0) input images to prevent overfitting. A flatten layer was used to reshape the output of the previous layer and convert it into a one-dimensional array that are then passed to the dense layer. The dense layer is a fully connected layer made of 500 nodes. A Rectified Linear Unit (ReLu) activation function is applied to provide non-linearity to the model. Another dropout layer with a rate of 0.5 was added by randomly dropping out nodes from the dense layer to prevent overfitting. The final layer was a dense output layer with softmax activation, which returned a vector of probabilities for each class (sign). The class with the highest probability was considered the predicted output of the neural network.

The CNN algorithm was trained on a laptop computer with a powerful graphics processing unit (Nvidia RTX 2060). A web-based computing platform (Jupyter Notebook) [28] was used, which offers an organised template that can display code, images, and graphs.



```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_20 (Conv2D)           (None, 28, 28, 60)        1560      
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 24, 24, 60)        90060     
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 12, 12, 60)        0         
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 10, 10, 30)        16230     
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 8, 8, 30)          8130      
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 4, 4, 30)          0         
_________________________________________________________________
dropout_10 (Dropout)         (None, 4, 4, 30)          0         
_________________________________________________________________
flatten_5 (Flatten)          (None, 480)               0         
_________________________________________________________________
dense_10 (Dense)             (None, 500)               240500    
_________________________________________________________________
dropout_11 (Dropout)         (None, 500)               0         
_________________________________________________________________
dense_11 (Dense)             (None, 4)                 2004      
=================================================================
Total params: 358,484
Trainable params: 358,484
Non-trainable params: 0
```

Figure 7: CNN architecture

3.9 Navigation

Navigation is the link between all the above sections, it is a combination of lane detection, steering control, traffic light detection and traffic sign classification algorithms that results in autonomous driving. The navigation software constitutes the decision-making feature of the robot as it was responsible for handling the data that was sent from the other algorithms. Considering the implementation in ROS, each algorithm had its node and was sending messages over topics.

The Lane Detection node was sending the slope as a float number (std_msgs/Float32 Message) [29] and was calculated in section 3.5.6. This part of the node contributes to the Steering control algorithm (section 3.6). In addition, this node was sending lane data as a string (std_msgs/String Message) [30] and outputted four possible values: “Go straight”, “Turn Right”, “Turn Left” and “No Lines detected”. These values were based on the value of the slope of the detected line.

The Light Detection node was sending the colour of light detected (Section 3.7.3) as a string (std_msgs/String Message) that returned four possible values: “Red”, “Yellow”, “Green” and “No traffic lights detected”.

The Sign Recognition node was sending the name of the traffic sign detected (Section 4.5) as a string (std_msgs/String Message) and outputted five possible values: “Ahead only”, “Stop”, “Turn Right Ahead” and “End speed + passing limit”.

The messages coming from the above nodes were led to a central node (Navigation node) which was controlling the movement of the robot. The navigation node mainly works on the steering control algorithm (Section 3.6) and made decisions based on traffic signs or lights detected, while keeping the robot within the lane. The navigation node was controlling the value of the linear and angular speed of the left and right motors (geometry_msgs/Twist.msg) [31]. The Twist message type contains the parameters of the linear speed on the x-axis and the angular speed on the z-axis.

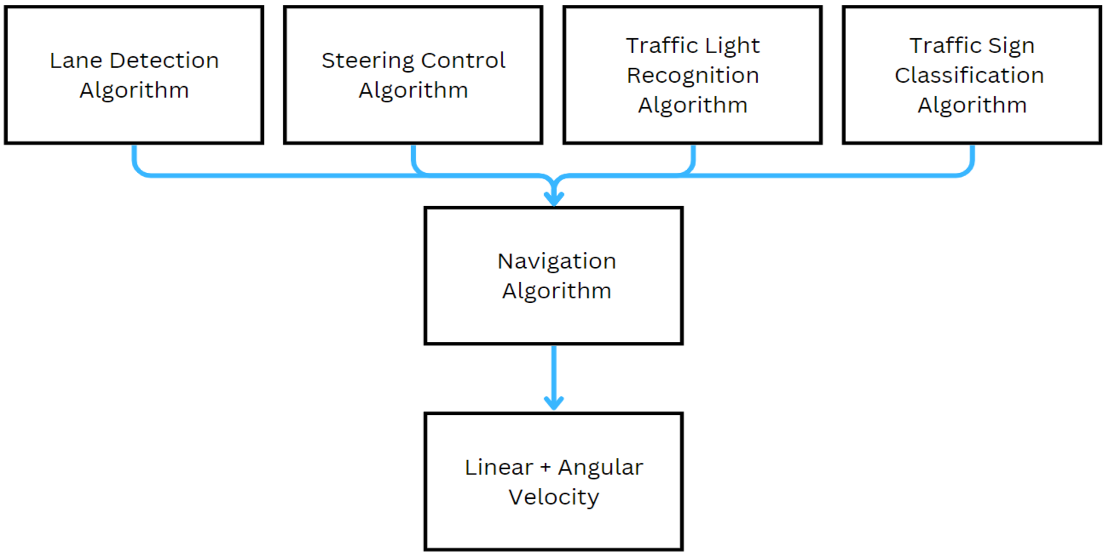

Lane Detection Algorithm
Steering Control Algorithm
Traffic Light Recognition Algorithm
Traffic Sign Classification Algorithm
Navigation Algorithm
Linear + Angular Velocity

Figure 8: Navigation algorithm’s flow chart

3.10 Web Application

A web application can be highly beneficial in a robotics application. In this case, the most important feature of a web app was the real-time data visualisation of the robot. The robot was sending the following data to the web application: robot’s linear speed, robot’s angular speed, left wheel’s angular speed, right wheel’s angular speed, traffic light and traffic sign detected. The web application was built using a web framework written in Python (Flask 2.2.3) [32]. Flask was preferred over other web frameworks since it is lightweight, flexible and allows building scalable web applications quickly and easily. It offers a robust set of features, including template rendering, URL routing, and HTTP request handling. Flask was rendering two folders stored in the Flask application’s directory called “templates” and “static”. It is recommended to name the template folder (“templates”) and static folder (“static”) and store them in the appropriate folder to have a working web app.

The template folder was storing a file with a “.html” extension. HTML (HyperText Markup Language) is a markup language used to create web pages. It was used to define the structure and content of the web page, including headings, images, and the robot’s transmitted data. Flask uses a templating engine (Jinja), which is used to render dynamic HTML pages and thus allows the use of variables (robot’s data).

The static folder was storing the images of traffic signs and traffic lights and a file with the “.css” extension. CSS (Cascading Style Sheets) is a style sheet language used for styling the web page written in HTML. It was used to edit the layout of the page, fonts, and colours of text, and display images or text over the background dashboard image.

In addition, Flask supports asynchronous requests, which allows web pages to update specific parts without refreshing the entire page. This was possible with the use of a web development technique called AJAX (Asynchronous JavaScript and XML). A JavaScript library that simplifies the process of client-side web development (jQuery) was used. This was essential because, without asynchronous requests, the user had to refresh the page to get the updated values sent by the robot. The robot’s data was sent in JSON (JavaScript Object Notation) format and the web app was programmed to refresh its values every second to display live data.

The design of the web-based application was inspired by the Apple CarPlay dashboard which offers a user-friendly interface. The dashboard design was edited and downloaded using a web-based design application (Figma) [33].

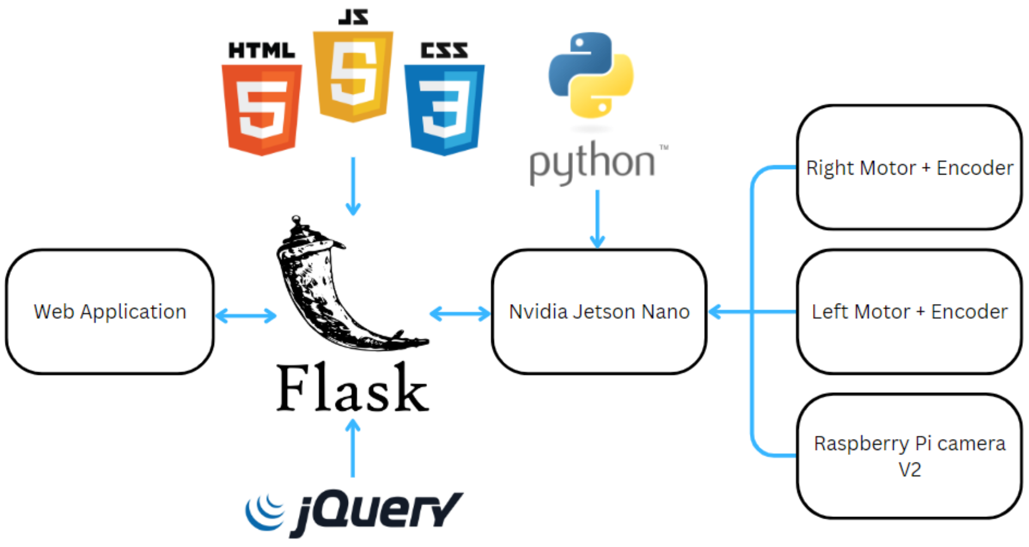


Figure 9: Web app flow chart

3.11 Hardware

The hardware components were another important aspect of the project. The selected hardware had to be able to run the algorithms mentioned in the above sections without delays. Hardware was the last part of the project, and the robot was assembled after the simulated robot was working as intended. The components were mounted on a chassis made of acrylic, which is modular and allows for easy replacement and upgrades of components. The hardware used for this project is the following:

- Nvidia Jetson Nano: This board is a small computer used for embedded AI and machine learning applications. It has a built-in GPU and CPU and can run ai models. It offers higher computational power compared to alternative bords like a Raspberry Pi. A 64GB microSD card which stores the operating system was required for booting Ubuntu (version 18.04.6 LTS) and for storage. The OS was flashed with the help of open-source flashing software (Etcher) [34]. The OS was given by the supervisor and was supporting Python 2.7.17 and 3.6.9. The step-by-step setup guide from the official website was followed [35]. Additionally, Jetson Nano does not have a built-in Wi-Fi module and thus an external Wi-Fi dongle was used for remote access (Secure Shell or SSH). Remote access on the board was made using a remote network tool software (MobaXterm) [36].
- Hacker Board: This board is an ESP32-based microcontroller with built-in Wi-Fi and Bluetooth modules. The ESP32 module was hosting a webpage where the user was able to

change network settings or the motor control mode. The motor control mode profile used was sending the linear and angular velocity of the robot, which was the same approach used in the simulated robot. The board also has a DC-DC converter and a motor driver circuit for providing power to the DC motors. The board runs an internal PID (Proportional-Integral-Derivative) feedback control to set the PWM (Pulse-Width-Modulation) sent to each motor. A display is attached to the board that displays the angular speed of each wheel and network details.

- DC Motors: The motors were mounted on the front side of the chassis forming a front-wheel-drive configuration allowing a more precise handling and stability. Wheels are attached to the motors giving chassis height and the camera is high enough to capture the lane in the frame.
- External Power Supply: Power to the Jetson Nano and the Hacker board was provided using a power bank (5V,3A) via USB cables (USB2.0 to Type C and USB2.0 to Micro-USB).
- Raspberry Pi camera V2: The camera board was mounted on the front side of the chassis since it was the component responsible for the navigation. It is equipped with a wide-angle lens allowing a wider field of view and thus detecting objects that appeared at the edges of the frame (traffic signs or traffic lights).

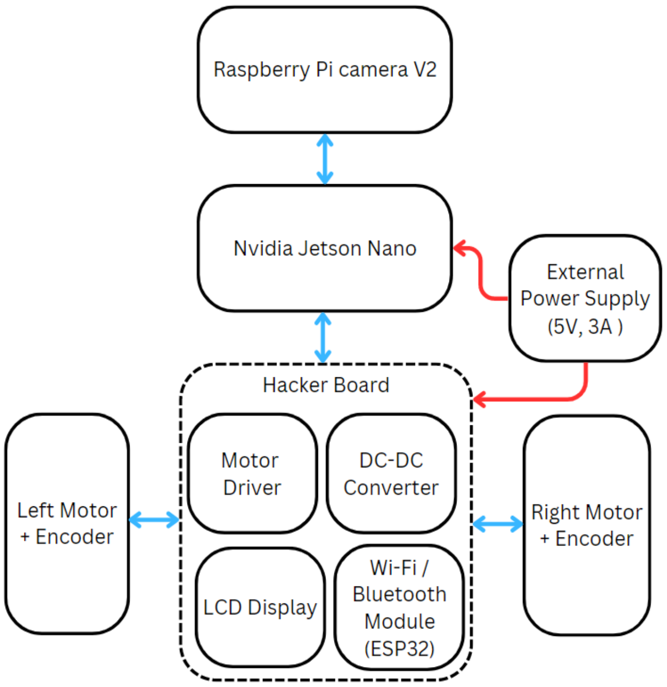


Figure 10: Mini vehicle’s hardware block diagram

4 Results and discussion

4.1 Introduction

This section of the report presents the results obtained by methods discussed in Section 3. The final step of the project was to test and evaluate the performance of the robot in both simulated and real-world environments. The test revealed some minor differences in the robot's performance between the simulated and real scenarios.

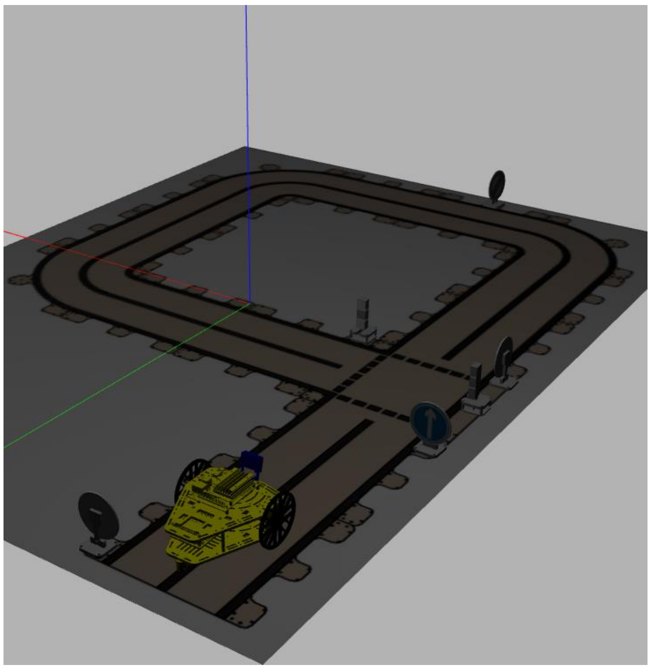


Figure 11: Simulated vehicle in the mini-town

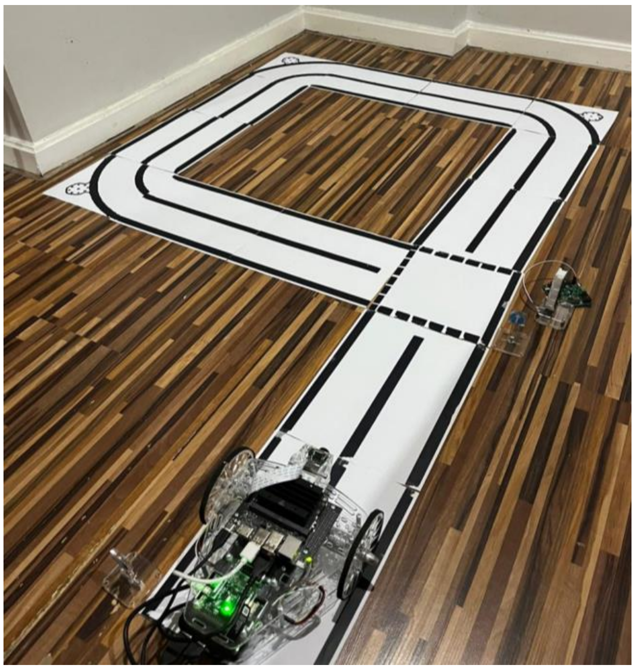


Figure 12: Real vehicle in the mini-town

4.2 Lane Detection Algorithm

In Figure 13, all the intermediate steps of the image processing utilised in the lane detection algorithm are attached. The algorithm in both the simulated (Figure 11) and real robot (Figure 12) were able to detect the lane lines and an approaching turn. The greyscale conversion in combination with Gaussian blur offered real time image processing and the robot was able to receive the coordinates of the detected lines in real-time. The selected camera frame dimensions and region of interest provided the ability to keep lines in the frame on a straight or turned lane. Visualising the lines detected, by displaying an overlay line was useful to fine-tune the algorithm.

The robot was also able to keep itself between the two lines, even in cases where there was a dashed line in the same direction as the regular line or no middle line. When testing on the real robot, a higher camera tilt was attempted to capture the lane from a distance, but this caused the detection of wrong lanes compared to the robot's position. An attempt to use a wide-angle camera view (1520x700) was made but when the robot had an offset from the middle line, the algorithm was detecting the middle line as an outer line.

An important factor that affected the algorithm's performance between the simulation and the real environment was the lighting conditions. Light reflections on the black lines were affecting the camera on what it identifies as a black line. The calibration was achieved using the Canny algorithm, as mentioned in section 3.5.3, by adjusting the gradient's upper and lower limits.

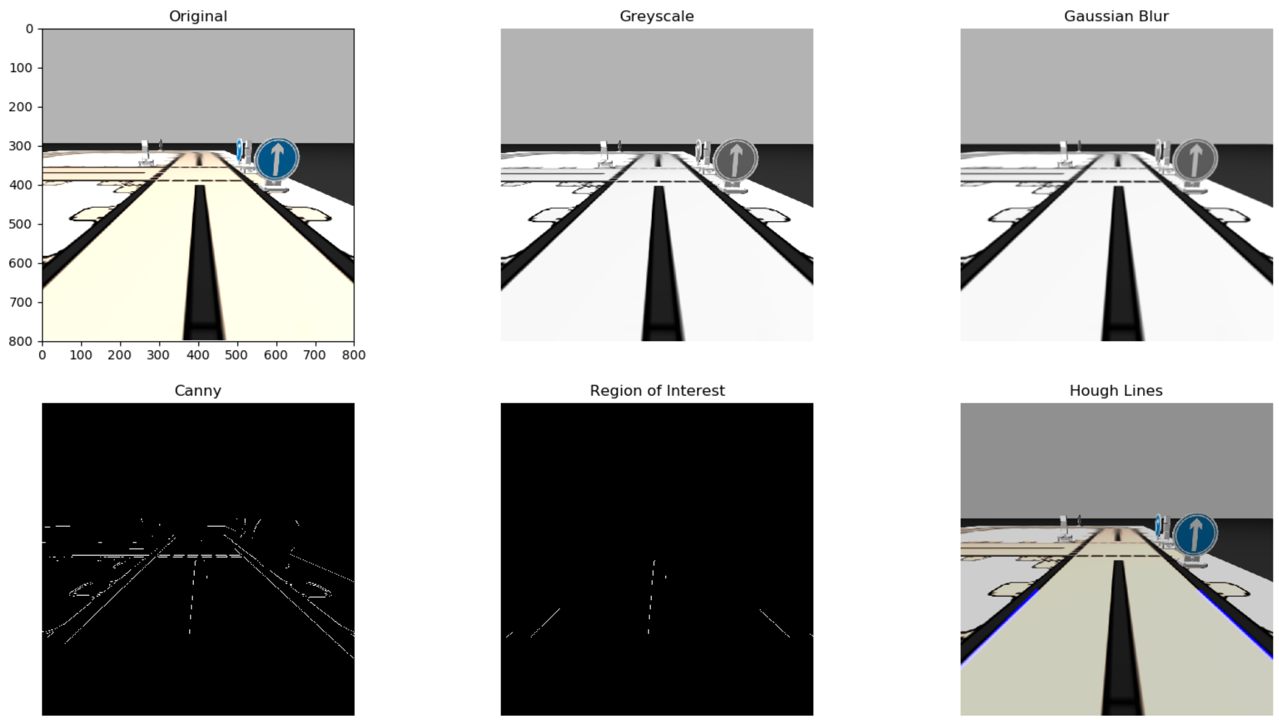


Figure 13: Lane detection image processing steps

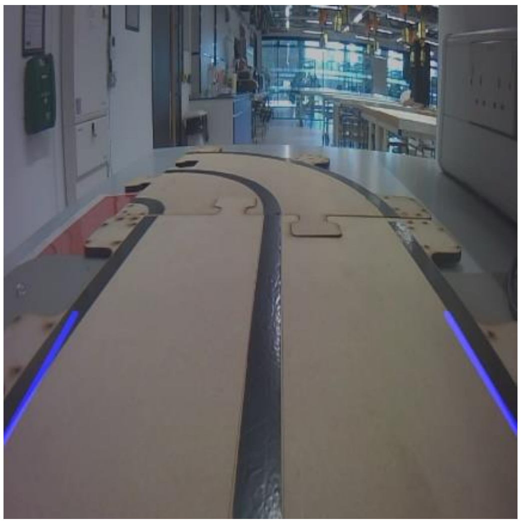


Figure 14**:** Detected lanes from the real mini vehicle

4.3 Steering Control

The robot’s steering control capability was first tested without the traffic signs and lights, and it was able to perform as intended. The algorithm was able to adjust the vehicle’s linear and angular speed in real time based on the number of lines detected and the slope value calculated from the lane detection algorithm (Section 3.5.6). A small change in the multiplier value on the angular speed of the mini vehicle was made to match the behaviour between the simulated and the real robot.

In Figure 15, the scenarios described in section 3.6 are displayed.

The robot was maintaining a constant speed on straight lines. In case where it had an offset from the middle line it corrected itself based on the line slope without oscillations. In a turn, where the line gradient was high it was turning with a higher angular speed to ensure that it will not get out of the track while keeping a constant linear speed.

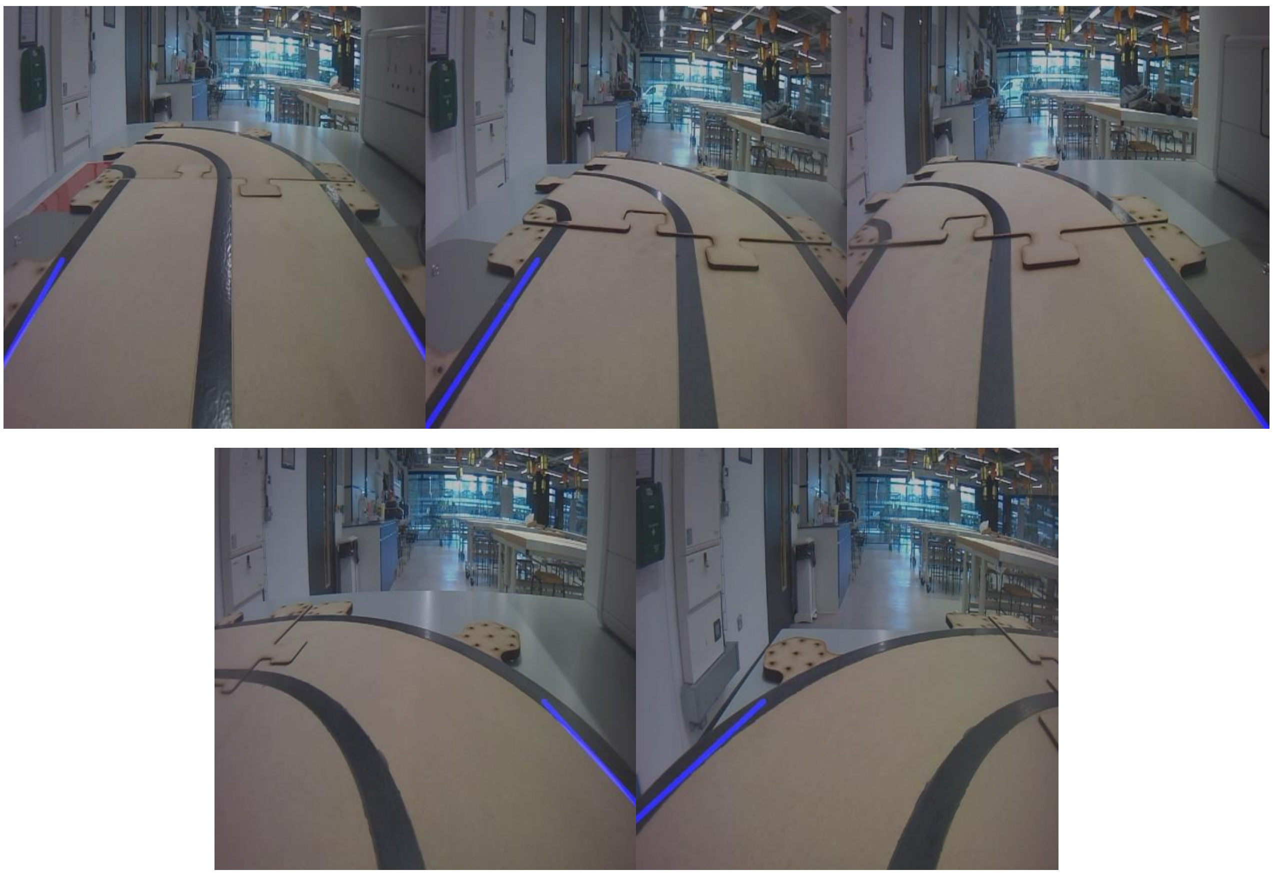


Figure 15: Lane detection scenarios

4.4 Traffic Light Recognition

The traffic light recognition algorithm was able to detect and recognise the colour of the traffic light using the onboard camera. In Figure 16, all the intermediate steps of the image processing used in traffic light recognition are shown.

It can be inspected that the algorithm isolated one colour that falls within the upper and lower HSV masking threshold. However, at the point where the median blur was applied, the algorithm could not detect if the coloured objects were traffic lights. The final step was to use the Hough Circle Transform, which provided high accuracy in detecting coloured pixels that form a circle and thus return the traffic light detected.

One factor that affected the algorithm's performance was the brightness level of the lights. Since the brightness of a simulated and real traffic light are different, a calibration process was needed to detect the colours in a real environment. The calibration was achieved using the masking method mentioned in section 3.7.2, which allows the adjustment of an upper and a lower limit. As a result, the calibration improved the algorithm’s ability to detect traffic lights accurately in different lighting conditions.

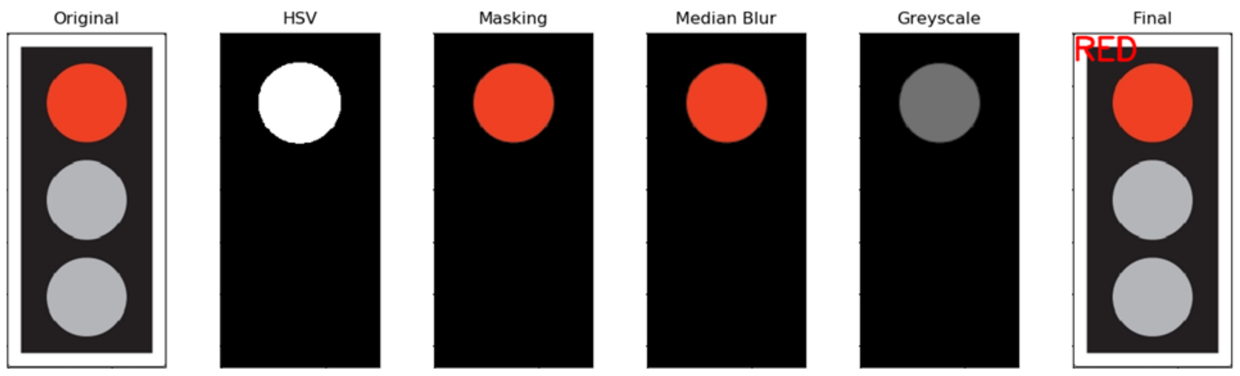


Figure 16: Image processing steps for red colour (the same process used to recognise yellow and green lights)

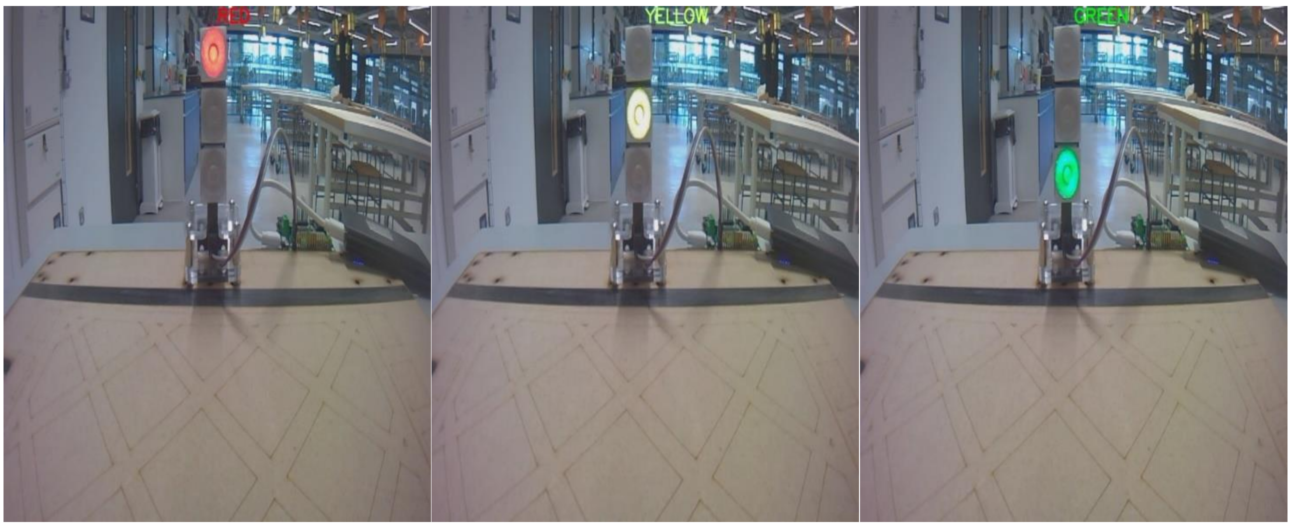


Figure 17: Recognised traffic lights from the real mini vehicle

4.5 Traffic Sign Classification

The first attempt of the traffic sign classification model training was carried out using RBG images. Using this method turned out that the model had less accuracy compared to the training using greyscale converted images [37]. This approach provided a faster response and more accurate predictions on traffic sign detection. Figure 18 shows the accuracy of the model increasing in every epoch. On the other hand, the loss of the model was decreasing in every epoch shown in Figure 19.

The model was able to achieve 98.58% accuracy, which makes the model achieve reliable decisions and be able to detect traffic signs in different lighting conditions.

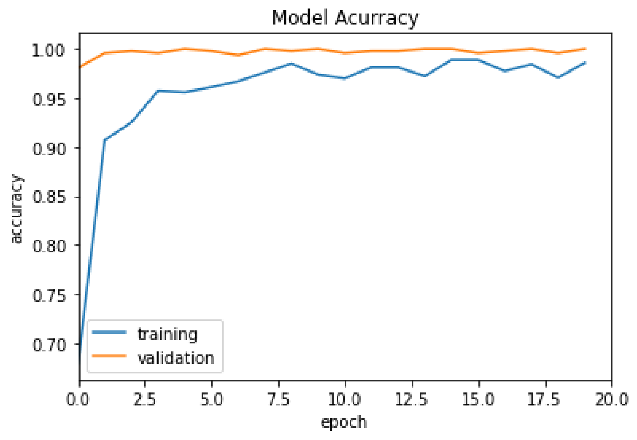


Figure 18: Model accuracy per epoch

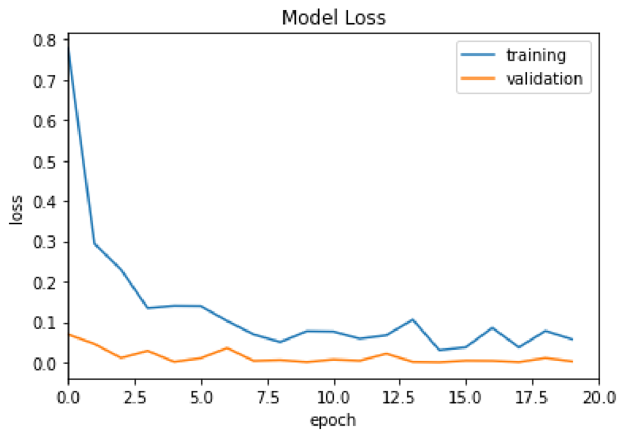


Figure 19: Model loss per epoch

The initial model testing was carried out in the Gazebo simulator. A pre-processing on the camera's frame was performed as the focus was only on the traffic signs. The model was applied on a cropped image that limits the frame and covers the right side of the frame, where the traffic signs are located. In Figure 20, the signs and their corresponding label detected by the robot are shown. The simulated robot was able to run the traffic sign classification algorithm without any processing issues. However, the real mini vehicle was struggling to run the algorithm, because Nvidia Jetson Nano had limited memory and the algorithm was causing delays in the whole system. The solution to the hardware limitation could be the use of a lighter or less complex algorithm.

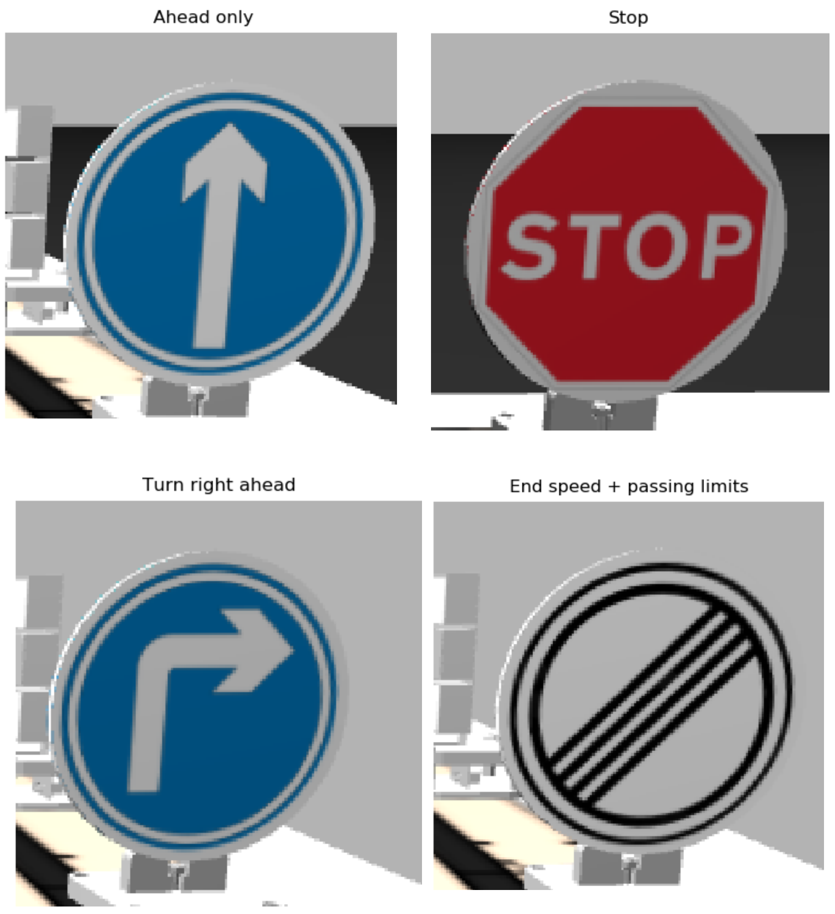


Figure 20: Traffic sign detection with the corresponding label

4.6 Navigation

When testing the robot's navigation capability and considering the robot's response, a decision to slow the rate of some nodes (Light Detection and Traffic Sign Recognition) was made. This didn't affect the overall performance since the robot encountered traffic signs and lights less frequently. This allowed the execution of the lane detection and steering control algorithms at faster rates. The simulated robot could navigate itself by taking into consideration the traffic signs, whereas the real mini vehicle could respect the traffic signs since it was struggling to run the traffic sign classification algorithm.

The robot was driving forward when encountered green traffic light or ahead only sign. In the case of a red traffic light or stop sign, the robot was not moving for 5 seconds. When the robot encountered an end speed and passing limits sign, it was increasing its speed. In case of facing a turn right ahead sign the robot was moving forward in a circular path towards the right side for three seconds and continued the forward movement. The above cases were executed while keeping the robot within the lane limits.

Figure 21, shows all the algorithms running concurrently, represented as nodes and corresponding topic names used to transfer the messages. The diagram clearly shows the source (Publisher) and destination (Subscriber) of the message. The Navigation node was receiving the data sent from the other algorithms from the following topics: “/slope”, “/lane”, “/light” and “/sign”. As a result, the Navigation node was sending the desired values of the linear and angular speed through the “/cmd_vel” topic. This topic was connected to a controller node (/rosserial), which was publishing the required angular speed of each wheel (“wr” and “wl” topics) to achieve the desired speeds.

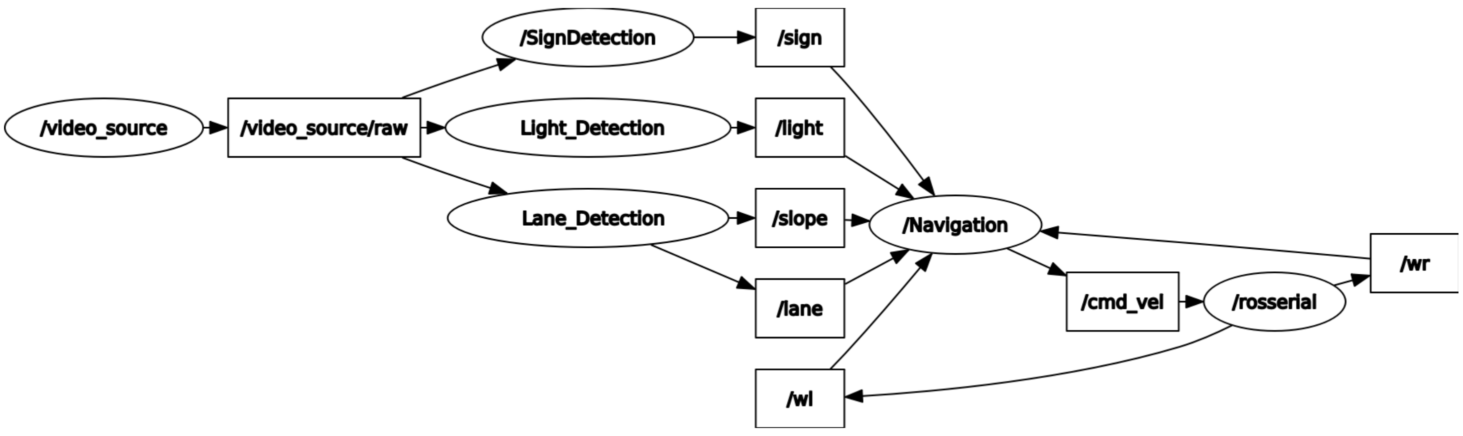


Figure 21: Visualisation of ROS Implementation

4.7 Web Application

The web application was running in parallel with the robot’s navigation node and updating its data without any delays. It was programmed to display the float numbers in two decimals format. The web application was displaying the detected traffic sign and light for two seconds after detection to allow for adequate observation. In cases where the robot did not detect any traffic signs or lights, the corresponding sections on the web app remained empty.

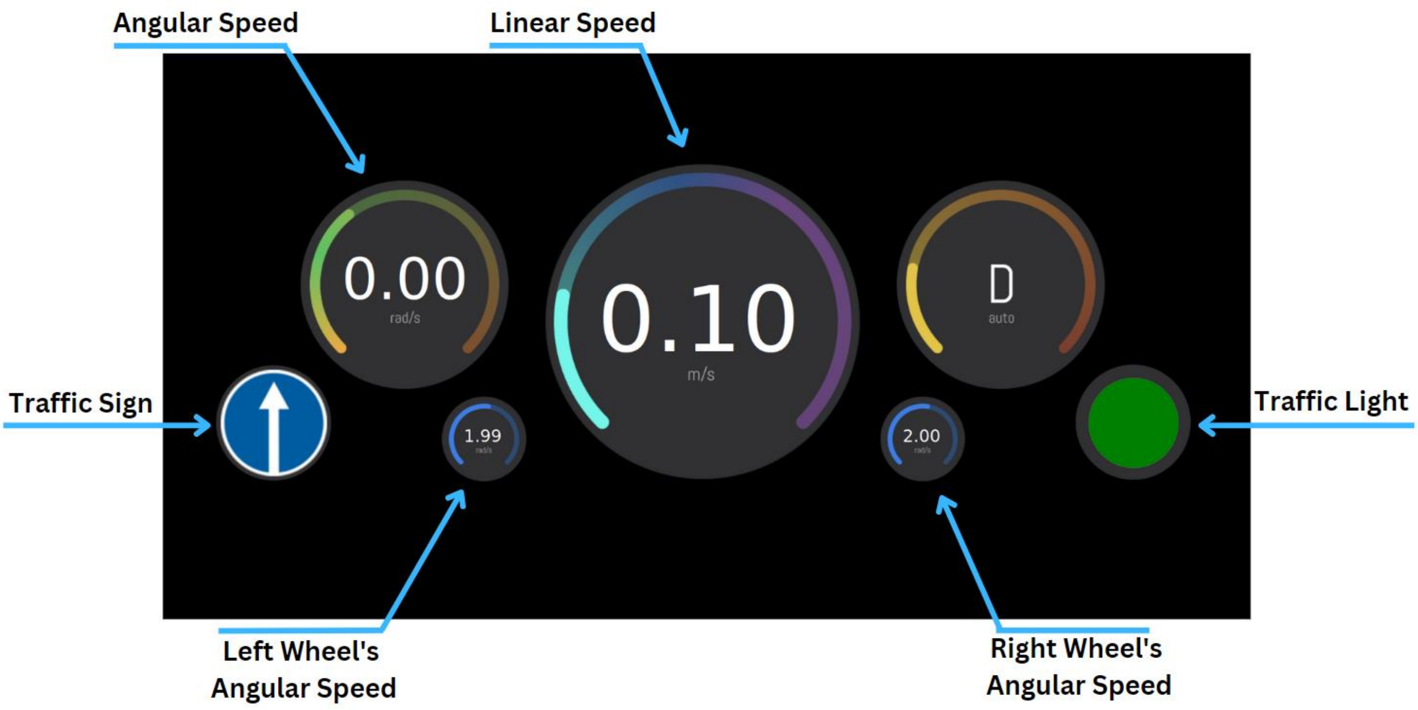


Figure 22: Web application’s front-end view

4.8 Hardware

Figure 23 shows the hardware components mentioned in section 3.5 mounted on the chassis. The Nvidia Jetson Nano board is a reliable component for robotics applications when running light algorithms. The raspberry pi camera V2 with the high image resolution and the wide-angle lens were able to improve the accuracy of the algorithms used. The hackerboard’s built-in PID controller could drive the dc motors precisely and in response to the node inputs without oscillations.

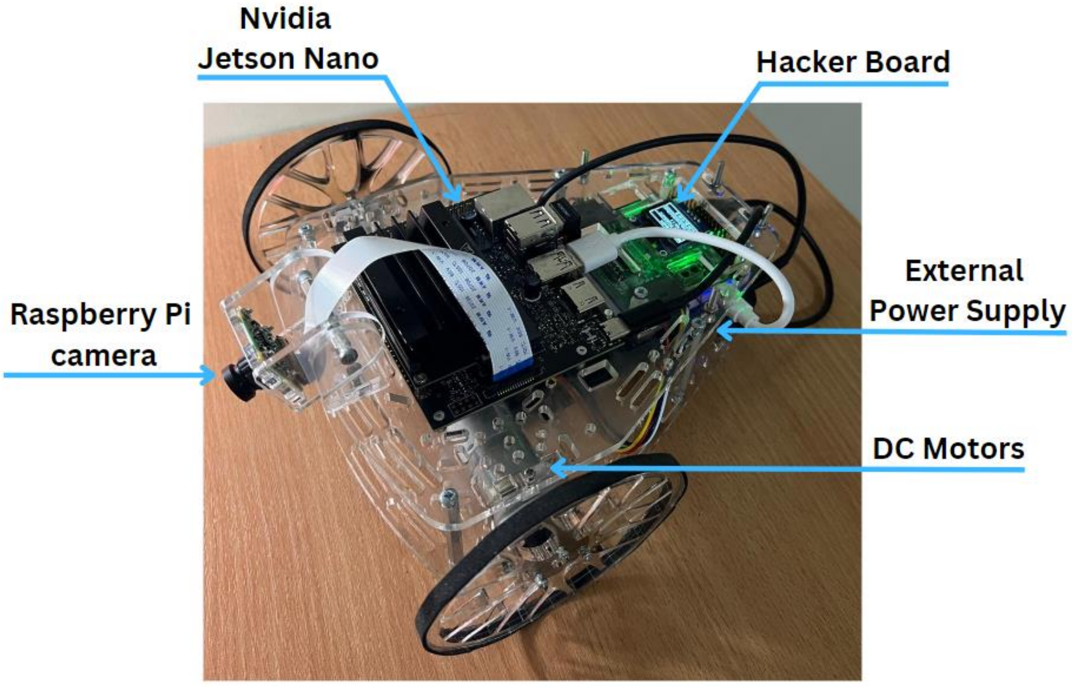


Figure 23: Hardware components of the real mini vehicle

4.9 Summary

In this section, we analysed the result obtained from both simulated and real mini vehicles, that use computer vision techniques and machine learning algorithms to navigate in the mini town. I have presented five algorithms that were able to drive the mini vehicle autonomously in the mini town.

The main difficulty that was encountered was the transition from the simulated mini vehicle to the real mini vehicle. The algorithms were firstly designed based on the simulated vehicle without taking into consideration the limitation that the hardware might face. Based on the results, limitations appeared when testing the traffic sign classification algorithm as the hardware could not keep up with the algorithm’s resource requirements. However, the use of Nvidia Jetson Nano with higher memory or the use of a neural network with different architecture may give higher flexibility.

To conclude, the self-driving feature requires the use of expensive hardware and multiple sensors working together to map its environment and navigate itself. The implementation of the above algorithms was made to balance the accuracy and the required computing power.

5 Conclusions and future work

5.1 Conclusions

In conclusion, the project covered a wide range of fields related to autonomous driving, including software and hardware. Most of the project objectives were successfully achieved, and the algorithms could navigate the autonomous vehicle in the mini town, despite some hardware limitations. Building an autonomous vehicle requires a combination of theoretical and practical knowledge across multiple fields. The project provided a valuable learning experience and demonstrated the process of autonomous vehicle development.

5.2 Future work

The project is scalable since it implements only a few features that real life autonomous vehicles have. The project can be further improved by adding more functionalities and handling real life scenarios. Some additional features can be:

- Obstacle detection and avoidance: Additional perception sensors like lidars and radars can be added and implement simultaneous localization and mapping (SLAM) for 3D mapping of the environment. In parallel with computer vision algorithms, the vehicle will be able to detect other vehicles or pedestrians and avoid collisions. This feature is related to path planning as the vehicle should plan a new path when encountering an obstacle.
- Web application: A better version of the existing web app can be made by adding more features. For example, making the web app interactive or adding remote control capabilities.
- Alternative deep learning object detection algorithms: Other models like Region-Based Convolutional Neural Networks (R-CNN) or You Only Look Once (YOLO) can be tested and compared on their performance. Some comparison metrics can be response time, accuracy, and repeatability.
- Expand the mini town: The mini town can be expanded by adding new traffic signs, obstacles, other cars, and pedestrians. In addition, different road conditions can be made like roundabouts and narrow or winding roads. These will test the vehicle's ability to handle different driving scenarios.

References

[1] M. Krishnakumar, “The 6 Autonomous Driving Levels Explained,” *W&B*, Sep. 03, 2022. https://wandb.ai/mukilan/autonomous-cars/reports/The-6-Autonomous-Driving-Levels-Explained--VmlldzoyNTcwOTQ1#:~:text=Level%201%3A%20Very%20light%20automation (accessed: Nov. 10, 2022).

[2] D. Etherington, "Over 1,400 self-driving vehicles are now in testing by 80 companies across the U.S.," TechCrunch, 11 June 2019. [Online]. Available: https://techcrunch.com/2019/06/11/over-1400-self-driving-vehicles-are-now-in-testing-by-80-companies-across-the-u-s/?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAAHMiIoRfligWA0TjZ7bz4FhOge9T-KImR9a7Fdnr49DvPZrDCxNwMs1erBsWES1jSOg3Pe8T_MusT-xdVL4695y5YTNiIM2zGUWYVF6LtdpnFCFuPBAdiKpDO0NGyuKAKoWXKA1M-WLnc-jCINTWa4uBVNVo6ZXvFOQKjdqwBt2o (accessed: Nov. 10, 2022).

[3] Z. Sun, “Vision Based Lane Detection for Self-Driving Car,” *IEEE Xplore*, Aug. 01, 2020. https://ieeexplore.ieee.org/document/9213624 (accessed Feb. 1, 2023).

[4] T.-D. Do, M.-T. Duong, Q.-V. Dang, and M.-H. Le, “Real-Time Self-Driving Car Navigation Using Deep Neural Network,” *IEEE Xplore*, 2018. https://ieeexplore.ieee.org/abstract/document/8595590 (accessed Feb. 1, 2023).

[5] R. Kulkarni, S. Dhavalikar, and S. Bangar, “Traffic Light Detection and Recognition for Self Driving Cars Using Deep Learning,” *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, Aug. 2018, doi: https://doi.org/10.1109/iccubea.2018.8697819 (accessed Feb. 1, 2023).

[6] Q. Wang, Q. Zhang, X. Liang, Y. Wang, C. Zhou, and V. I. Mikulovich, “Traffic Lights Detection and Recognition Method Based on the Improved YOLOv4 Algorithm,” *Sensors*, vol. 22, no. 1, p. 200, Dec. 2021, doi: https://doi.org/10.3390/s22010200 (accessed Feb. 1, 2023).

[7] O. Nacir, M. Amna, W. Imen, and B. Hamdi, “Yolo V5 for Traffic Sign Recognition and Detection Using Transfer Learning,” IEEE Xplore, Oct. 01, 2022. https://ieeexplore.ieee.org/document/10044022 (accessed Feb. 1, 2023).

[8] “Gazebo : Tutorial : Windows,” *classic.gazebosim.org*. https://classic.gazebosim.org/tutorials?tut=install_on_windows&cat=install (accessed Nov. 30, 2022).

[9] “Install Ubuntu desktop,” *Ubuntu*. https://ubuntu.com/tutorials/install-ubuntu-desktop#1-overview (accessed Oct. 5, 2022).

[10] “Python Release Python 3.8.10,” Python.org. https://www.python.org/downloads/release/python-3810/ (accessed Oct. 2, 2022).

[11] “noetic - ROS Wiki,” wiki.ros.org. http://wiki.ros.org/noetic (accessed Oct. 15, 2022).

[12] “ROS Tutorials - ROS Noetic For Beginners - YouTube,” www.youtube.com. https://youtube.com/playlist?list=PLLSegLrePWgIbIrA4iehUQ-impvIXdd9Q (accessed Oct. 15, 2022).

[13] “ros_comm - ROS Wiki,” wiki.ros.org. http://wiki.ros.org/ros_comm (accessed Oct. 15, 2022).

[14] Microsoft, “Visual Studio Code,” Visualstudio.com, Apr. 14, 2016. https://code.visualstudio.com/ (accessed Nov. 30, 2022).

[15] “Gazebo,” staging.gazebosim.org. https://staging.gazebosim.org/home (accessed Nov. 30, 2022).

[16] “Manchester-Robotics,” GitHub. https://github.com/Manchester-Robotics (accessed Oct. 20, 2022).

[17] “OpenCV Python Tutorial - Find Lanes for Self-Driving Cars (Computer Vision Basics Tutorial),” www.youtube.com. https://www.youtube.com/watch?v=eLTLtUVuuy4&t=4181s (accessed Nov. 5, 2022).

[18] madrasresearchorg, “SELF-DRIVING CARS USING OPEN CV,” MSRF|NGO, Aug. 15, 2021. https://www.madrasresearch.org/post/self-driving-cars-using-open-cv (accessed Nov. 21, 2022).

[19] OpenCV, “OpenCV library,” Opencv.org, 2019. https://opencv.org/ (accessed Nov. 21, 2022).

[20] “cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython - ROS Wiki,” wiki.ros.org. http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython (accessed Nov. 23, 2022).

[21] Matplotlib, “Matplotlib: Python plotting — Matplotlib 3.1.1 documentation,” Matplotlib.org, 2012. https://matplotlib.org/ (accessed Nov. 21, 2022).

[22] “Understanding Hough Transform With A Lane Detection Model,” Paperspace Blog, Jul. 23, 2021. https://blog.paperspace.com/understanding-hough-transform-lane-detection/ (accessed Dec. 17, 2022).

[23] datahacker.rs, “OpenCV #010 Circle Detection Using Hough Transform,” Master Data Science, Jul. 04, 2019. https://datahacker.rs/opencv-circle-detection-hough-transform/ (accessed Dec. 15, 2022).

[24] “GTSRB - German Traffic Sign Recognition Benchmark,” www.kaggle.com. https://www.kaggle.com/datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign (accessed Apr. 28, 2023).

[25] “Traffic Sign Classification,” Computer Vision Zone. https://www.computervision.zone/courses/traffic-sign-classification/ (accessed Feb. 15, 2023).

[26] Google, “TensorFlow,” TensorFlow, 2019. https://www.tensorflow.org/ (accessed Feb. 10, 2023).

[27] Keras, “Home - Keras Documentation,” Keras.io, 2019. https://keras.io/ (accessed Feb. 10, 2023).

[28] Jupyter, “Project Jupyter,” Jupyter.org, 2019. https://jupyter.org/ (accessed Feb. 15, 2023).

[29] “std_msgs/Float32 Documentation,” docs.ros.org. http://docs.ros.org/en/melodic/api/std_msgs/html/msg/Float32.html (accessed Oct. 15, 2022).

[30] “std_msgs/String Documentation,” docs.ros.org. http://docs.ros.org/en/melodic/api/std_msgs/html/msg/String.html (accessed Oct. 15, 2022).

[31] “geometry_msgs/Twist Documentation,” docs.ros.org. http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Twist.html (accessed Oct. 15, 2022).

[32] “Welcome to Flask — Flask Documentation (2.3.x),” flask.palletsprojects.com. https://flask.palletsprojects.com/en/2.3.x/ (accessed Mar. 16, 2023).

[33] “Apple CarPlay Preview WWDC 2022 (Community) | Figma Community,” Figma, 2023. https://www.figma.com/community/file/1119210027945914524 (accessed Mar. 20, 2023).

[34] “balenaEtcher - Flash OS images to SD cards & USB drives,” balenaEtcher. https://www.balena.io/etcher (accessed Oct. 7, 2022).

[35] “Getting Started With Jetson Nano Developer Kit,” NVIDIA Developer, Mar. 05, 2019. https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit (accessed Mar. 20, 2023).

[36] Mobatek, “MobaXterm free Xserver and tabbed SSH client for Windows,” mobaxterm.mobatek.net. https://mobaxterm.mobatek.net/ (accessed Apr. 20, 2023).

[37] H. M. Bui, M. Lech, E. Cheng, K. Neville, and I. S. Burnett, “Using grayscale images for object recognition with convolutional-recursive neural network,” IEEE Xplore, Jul. 01, 2016. https://ieeexplore.ieee.org/document/7562656 (accessed Feb. 20, 2023).

Appendices

A Project outline

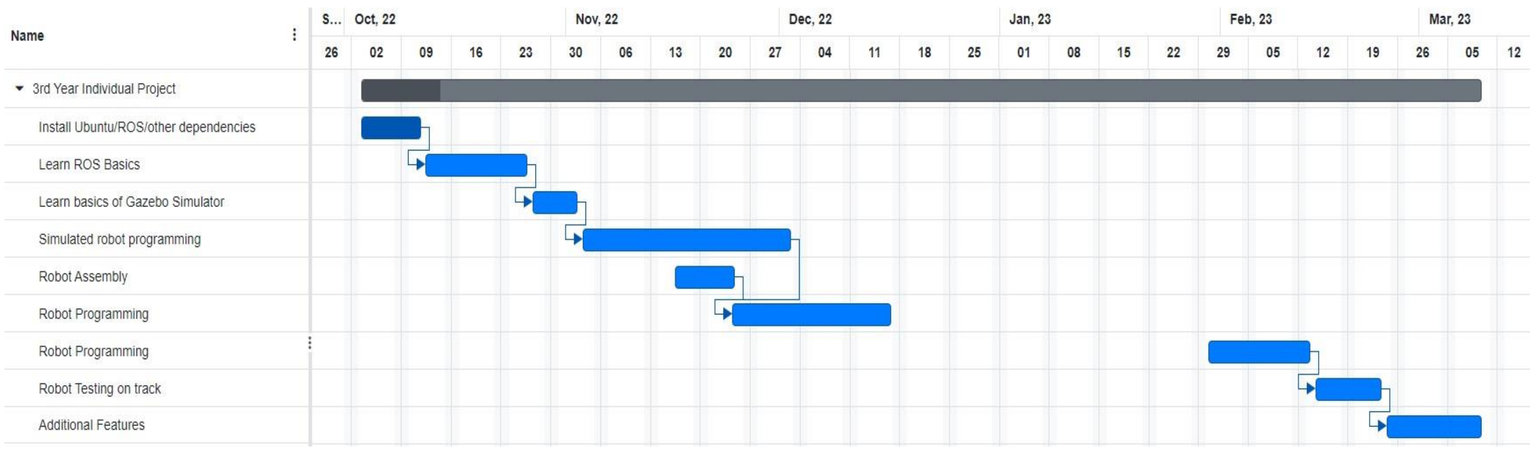
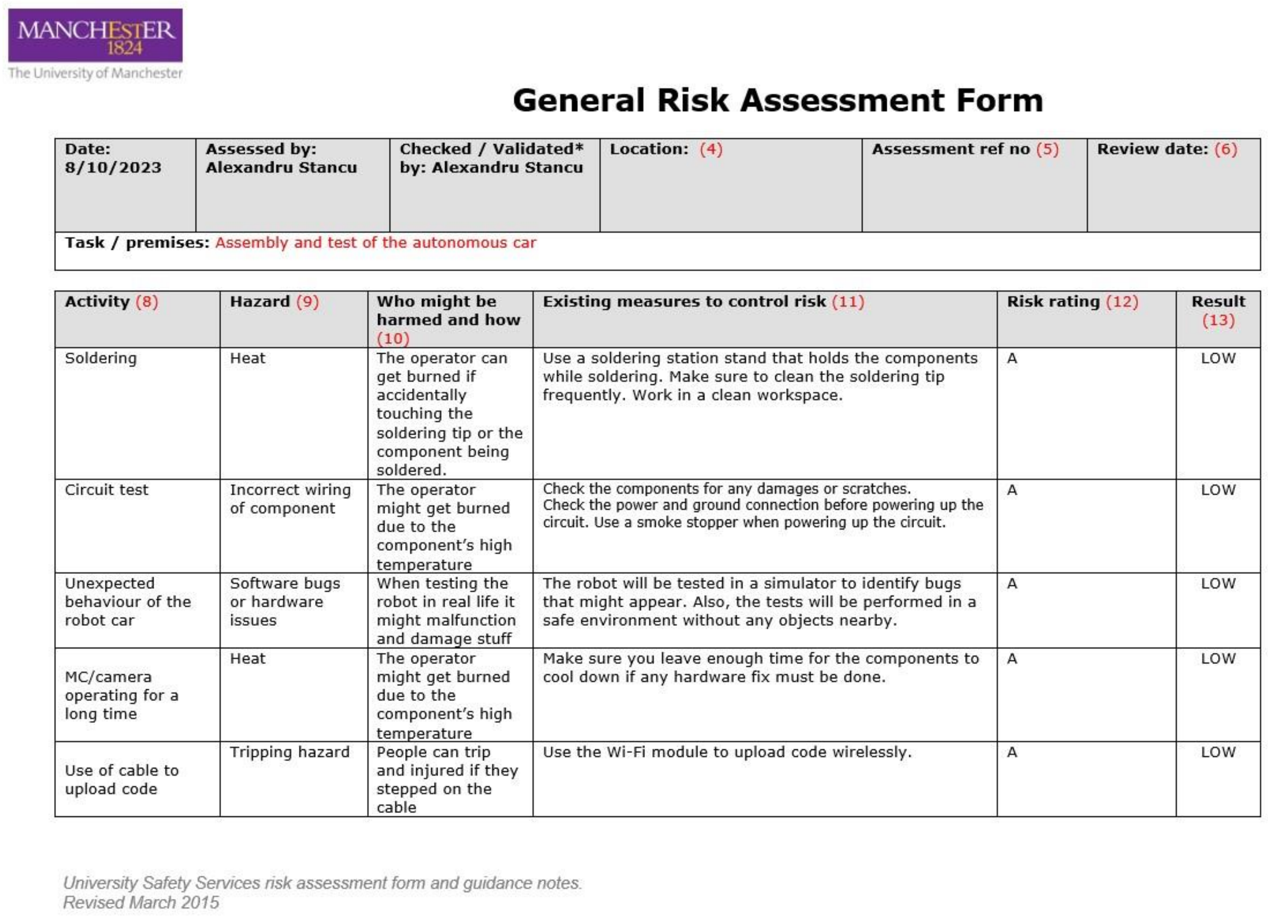


Figure 24: Gantt Chart

B Risk assessment



The University of Manchester

General Risk Assessment Form

Date: 8/10/2023	**Assessed by:** Alexandru Stancu	**Checked / Validated* by:** Alexandru Stancu	**Location:** (4)	**Assessment ref no** (5)	**Review date:** (6)
Task / premises: Assembly and test of the autonomous car					

Activity (8)	**Hazard** (9)	**Who might be harmed and how** (10)	**Existing measures to control risk** (11)	**Risk rating** (12)	**Result** (13)
Soldering	Heat	The operator can get burned if accidentally touching the soldering tip or the component being soldered.	Use a soldering station stand that holds the components while soldering. Make sure to clean the soldering tip frequently. Work in a clean workspace.	A	LOW
Circuit test	Incorrect wiring of component	The operator might get burned due to the component's high temperature	Check the components for any damages or scratches. Check the power and ground connection before powering up the circuit. Use a smoke stopper when powering up the circuit.	A	LOW
Unexpected behaviour of the robot car	Software bugs or hardware issues	When testing the robot in real life it might malfunction and damage stuff	The robot will be tested in a simulator to identify bugs that might appear. Also, the tests will be performed in a safe environment without any objects nearby.	A	LOW
MC/camera operating for a long time	Heat	The operator might get burned due to the component's high temperature	Make sure you leave enough time for the components to cool down if any hardware fix must be done.	A	LOW
Use of cable to upload code	Tripping hazard	People can trip and injured if they stepped on the cable	Use the Wi-Fi module to upload code wirelessly.	A	LOW

University Safety Services risk assessment form and guidance notes. Revised March 2015

Table 1: Risk Assessment Form

C Code

Navigation Code

```
#!/usr/bin/env python

import rospy

import cv2

from sensor_msgs.msg import Image

from std_msgs.msg import String

from cv_bridge import CvBridge, CvBridgeError

from geometry_msgs.msg import Twist

from std_msgs.msg import Float32
```

```
lane = None

slope = None

sign = None

right = None

left = None


def wlcallback(wl):

   global left

   left = wl.data


def wrcallback(wr):

   global right

   right = wr.data


def callbacklane(ln):

   global lane

   lane = ln.data


def callbackslope(slp):

   global slope

   slope = slp.data


def callbacksign(sgn):
   global sign
   sign = sgn.data


def callbacklight(lgt):
   global light
   light = lgt.data


def main():
  try:
   rospy.spin()
  except KeyboardInterrupt:
   rospy.loginfo("Shutting down")


  cv2.destroyAllWindows()
if __name__ == '__main__':
   rospy.init_node('Navigation', anonymous=False)
   rate = rospy.Rate(10)
   pub = rospy.Publisher("/cmd_vel", Twist, queue_size=10)
   sublane = rospy.Subscriber("/lane",String,callback=callbacklane)
   subslope = rospy.Subscriber("/slope",Float32,callback=callbackslope)
   subsing = rospy.Subscriber("/sign",String,callback=callbacksign)
   subwl=rospy.Subscriber("/wl",Float32,callback=wlcallback)
   subwr=rospy.Subscriber("/wr",Float32,callback=wrcallback)
   sublight = rospy.Subscriber("/light",String,callback=callbacklight)
   msg = Twist()

   dt = 0.1

   total = 0.0

   totalth=0.0

   global speed

   speed = 0.1


while not rospy.is_shutdown():

   if sign == 'End speed + passing limits':

     speed = 0.11


   if lane == "Go straight" or sign == "Ahead only":

      msg.linear.x = speed

      msg.angular.z = 0.0

      pub.publish(msg)


   elif lane == "Turn Left" or lane == "Turn Right":

      msg.linear.x = speed

      msg.angular.z = 0.25*(1/slope)

      pub.publish(msg)


   else:

      msg.angular.z = 0.0

      msg.linear.x = 0.0
```

```
        pub.publish(msg)


    if sign == "Stop" or light == "Red":
      msg.linear.x = 0.0
      msg.angular.z = 0.0
      pub.publish(msg)
      rospy.sleep(5)
      continue


    if sign == "Turn right ahead":
      msg.linear.x = 0.1
      msg.angular.z = -0.5
      pub.publish(msg)
      rospy.sleep(3)


    pub.publish(msg)
    rate.sleep()
```

Lane Detection Code

```
#!/usr/bin/env python


import rospy
import cv2
import numpy as np
```

```
from sensor_msgs.msg import Image

from std_msgs.msg import String

from cv_bridge import CvBridge, CvBridgeError

from std_msgs.msg import Float32


def compute_steering_angle(frame, lane_lines):

  msg = String()

  slp = Float32()

  if len(lane_lines) == 0:

     #rospy.loginfo('No lane lines detected, do nothing')

     return "No Lines"


  if len(lane_lines) == 1:

      try:

        slope =(lane_lines[0][3]-lane_lines[0][1])/(lane_lines[0][2]-lane_lines[0][0])

        if slope > 0:

          slp = slope

          msg = "Turn Left"

          pub.publish(msg)

          pub1.publish(slp)

          return msg

        else:

          slp = slope

          msg = "Turn Right"
          pub.publish(msg)

          pub1.publish(slp)

          return msg

        except:

         slp = "None"

         msg = "None"

         pub.publish(msg)

         pub1.publish(slp)

         return msg


  else:

    msg = "Go straight"

    pub.publish(msg)

    return msg

def can(image):

    gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)

    blur = cv2.GaussianBlur(gray,(5,5),0)

    can = cv2.Canny(blur,455,460)

    return can


def display_lines(image,lines):

  line_image = np.zeros_like(image)

  if lines is not None:

    for line in lines:
```

```
    x1,y1,x2,y2 = line.reshape(4)

    cv2.line(line_image,(x1,y1),(x2,y2),(255,0,0),10)

  return line_image


def make_coordinates(image,line_parameters):

  height = image.shape[0]

  width = image.shape[0]

  slope,intercept = line_parameters

  y1 = height  # bottom of the frame

  y2 = int(y1 * (3 / 5))  # make points from middle of the frame down

  x1 = max(-width, min(2 * width, int((y1 - intercept) / slope)))

  x2 = max(-width, min(2 * width, int((y2 - intercept) / slope)))

  return np.array([x1,y1,x2,y2])


def average_slope_intercept(image,lines):

  height = image.shape[0]

  width = image.shape[0]

  lane_lines=[]

  left_fit = []

  right_fit = []

  boundary = 1/3

  right_region_boundary = width * (1 - boundary)  # left lane line segment should be on left 2/3 of
the screen

  left_region_boundary = width * boundary # right lane line segment should be on left 2/3 of the
screen
```

```
if lines is None:
      return lane_lines

for line in lines:
  for x1, y1, x2, y2 in line:
    if x1 == x2:
      continue
    parameters = np.polyfit((x1,x2),(y1,y2),1)
    slope = parameters[0]
    intercept = parameters[1]
    if slope < 0:
      if x1 < left_region_boundary and x2 < left_region_boundary:
        left_fit.append((slope, intercept))
    else:
      if x1 > right_region_boundary and x2 > right_region_boundary:
        right_fit.append((slope, intercept))
left_fit_average = np.average(left_fit,axis=0)
right_fit_average = np.average(right_fit,axis=0)
if len(left_fit) > 0:
  lane_lines.append(make_coordinates(image, left_fit_average))

if len(right_fit) > 0:
  lane_lines.append(make_coordinates(image, right_fit_average))
```

```
 return lane_lines


def region_of_interest(image):
 height = image.shape[0]
 triangle = np.array([
 [(2,height-200),(810,height-200),(400,400)]
 ])
 mask = np.zeros_like(image)
 cv2.fillPoly(mask,triangle,255)
 masked_image = cv2.bitwise_and(image,mask)
 return masked_image


class camera_1:
 def __init__(self):
  self.image_sub = rospy.Subscriber("/camera/image_raw", Image, self.callback)
 def callback(self,data):
  bridge = CvBridge()
  try:
   image = bridge.imgmsg_to_cv2(data, desired_encoding="bgr8")
  except CvBridgeError as e:
   rospy.logerr(e)


  lane_image = np.copy(image)
  canny = can(lane_image)
  cropped_image = region_of_interest(canny)

  lines =
cv2.HoughLinesP(cropped_image,2,np.pi/180,10,np.array([]),minLineLength=30,maxLineGap=10)

  averaged_lines = average_slope_intercept(lane_image,lines)

  line_image = display_lines(lane_image,averaged_lines)

  combo_image = cv2.addWeighted(lane_image,0.8,line_image,1,1)

  steeringangle = compute_steering_angle(combo_image,averaged_lines)


if __name__ == '__main__':

  rospy.init_node('Lane_Detection', anonymous=False)

  pub = rospy.Publisher('lane', String, queue_size=10)

  pub1 = rospy.Publisher('slope', Float32, queue_size=10)

  rate = rospy.Rate(5)

  camera_1()


  try:

    rospy.spin()

  except KeyboardInterrupt:

    rospy.loginfo("Shutting down")


  cv2.destroyAllWindows()
```

Traffic Light Recognition Code

```
#!/usr/bin/env python
```

```
import rospy

from sensor_msgs.msg import Image

from std_msgs.msg import String

from cv_bridge import CvBridge, CvBridgeError

from std_msgs.msg import Float32

import cv2

import numpy as np

import matplotlib.pyplot as plt


def process(image):

    msg = String()

    font = cv2.FONT_HERSHEY_SIMPLEX


    # Converts images from BGR to HSV

    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    lower_red = np.array([0,50,50])

    upper_red = np.array([10,255,255])


    lower_orange = np.array([5,100,100])

    upper_orange = np.array([15,255,255])


    lower_yellow = np.array([20,100,100])

    upper_yellow = np.array([30,255,255])
```

```
lower_green = np.array([50,100,100])

upper_green = np.array([70,255,255])


    # Here we are defining range of bluecolor in HSV

    # This creates a mask of blue coloured

    # objects found in the frame.

red_mask = cv2.inRange(hsv, lower_red, upper_red)

yellow_mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

orange_mask = cv2.inRange(hsv, lower_orange, upper_orange)

green_mask = cv2.inRange(hsv, lower_green, upper_green)


    # The bitwise and of the frame and mask is done so

    # that only the blue coloured objects are highlighted

    # and stored in res

red_res = cv2.bitwise_and(image,image, mask= red_mask)

orange_res = cv2.bitwise_and(image,image, mask= orange_mask)

yellow_res = cv2.bitwise_and(image,image, mask= yellow_mask)

green_res = cv2.bitwise_and(image,image, mask= green_mask)

    # median blur

redb= cv2.medianBlur(red_res,3)

orangeb = cv2.medianBlur(orange_res,3)

yellowb= cv2.medianBlur(yellow_res,3)

greenb = cv2.medianBlur(green_res,3)
```

```
        #RGB to Grey
    red_grey = cv2.cvtColor(redb,cv2.COLOR_BGR2GRAY)
    orange_grey = cv2.cvtColor(orangeb,cv2.COLOR_BGR2GRAY)
    yellow_grey = cv2.cvtColor(yellowb,cv2.COLOR_BGR2GRAY)
    green_grey = cv2.cvtColor(greenb,cv2.COLOR_BGR2GRAY)

        # hough circle detect
    r_circles = cv2.HoughCircles(red_grey,
cv2.HOUGH_GRADIENT,1,20,param1=60,param2=40,minRadius=0,maxRadius=0)
    o_circles = cv2.HoughCircles(orange_grey,
cv2.HOUGH_GRADIENT,1,20,param1=60,param2=40,minRadius=0,maxRadius=0)
    y_circles = cv2.HoughCircles(yellow_grey,
cv2.HOUGH_GRADIENT,1,20,param1=60,param2=40,minRadius=0,maxRadius=0)
    g_circles = cv2.HoughCircles(green_grey,
cv2.HOUGH_GRADIENT,1,20,param1=60,param2=40,minRadius=0,maxRadius=0)

    if r_circles is not None:
        msg = 'Red'
        pub.publish(msg)
        return 'Red'

    if y_circles is not None:
        msg = 'Yellow'
        pub.publish(msg)
        return 'Yellow'
    if g_circles is not None:

        msg = 'Green'

        pub.publish(msg)

        return 'Green'


    else:

        return "No traffic light"




class camera_1:


  def __init__(self):

    self.image_sub = rospy.Subscriber("/camera/image_raw", Image, self.callback)


  def callback(self,data):

    bridge = CvBridge()


    try:

      image = bridge.imgmsg_to_cv2(data, desired_encoding="bgr8")


    except CvBridgeError as e:

      rospy.logerr(e)
    img = np.copy(image)

    light = process(img)

    print(light)


if __name__ == '__main__':

    rospy.init_node('Light_Detection', anonymous=False)

    pub = rospy.Publisher('light', String, queue_size=10)

    rate = rospy.Rate(10)

    camera_1()


    try:

      rospy.spin()

    except KeyboardInterrupt:

      rospy.loginfo("Shutting down")


    cv2.destroyAllWindows()
```

Traffic Sign Classification Training Code

```
import matplotlib.pyplot as plt

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam

from keras.utils.np_utils import to_categorical

from tensorflow.keras.layers import Dropout, Flatten
from keras.layers.convolutional import Conv2D, MaxPooling2D

import cv2

from sklearn.model_selection import train_test_split

import pickle

import os

import pandas as pd

import random

from tensorflow.keras.preprocessing.image import ImageDataGenerator


#Parameters

path = r"H:\\Traffic_Sign_Recognition\\Train" # folder with all the class folders

labelFile = r'H:\\Signdataset\\labels\\labels.csv' # file with all names of classes

batch_size_val= 3   # how many to process together

steps_per_epoch_val=445

epochs_val=20

imageDimesions = (32,32,3)

testRatio = 0.2    # if 1000 images split will 200 for testing

validationRatio = 0.2 # if 1000 images 20% of remaining 800 will be 160 for validation


# Importing of the Images

count = 0

images = []

classNo = []

myList = os.listdir(path)
```

```
print("Total Classes Detected:",len(myList))

noOfClasses=len(myList)

print("Importing Classes.....")

for x in range(0,len(myList)):

    myPicList = os.listdir(path+"/"+str(count))

    for y in myPicList:

        curImg = cv2.imread(path+"/"+str(count)+"/"+y)

        curImg = cv2.resize(curImg, (32,32), interpolation = cv2.INTER_AREA)

        images.append(curImg)

        classNo.append(count)

    print(count, end =" ")

    count +=1

print(" ")

images = np.array(images)

classNo = np.array(classNo)


# Split Data

X_train, X_test, y_train, y_test = train_test_split(images, classNo, test_size=testRatio)

X_train, X_validation, y_train, y_validation = train_test_split(X_train, y_train,
test_size=validationRatio)

# X_train = ARRAY OF IMAGES TO TRAIN

# y_train = CORRESPONDING CLASS ID


# TO CHECK IF NUMBER OF IMAGES MATCHES TO NUMBER OF LABELS FOR EACH DATA SET

print("Data Shapes")
print("Train",end = "");print(X_train.shape,y_train.shape)

print("Validation",end = "");print(X_validation.shape,y_validation.shape)

print("Test",end = "");print(X_test.shape,y_test.shape)

assert(X_train.shape[0]==y_train.shape[0]), "The number of images in not equal to the number of
lables in training set"

assert(X_validation.shape[0]==y_validation.shape[0]), "The number of images in not equal to the
number of lables in validation set"

assert(X_test.shape[0]==y_test.shape[0]), "The number of images in not equal to the number of
lables in test set"

assert(X_train.shape[1:]==(imageDimesions))," The dimesions of the Training images are wrong "

assert(X_validation.shape[1:]==(imageDimesions))," The dimesionas of the Validation images are
wrong "

assert(X_test.shape[1:]==(imageDimesions))," The dimesionas of the Test images are wrong"


# READ CSV FILE

data=pd.read_csv(labelFile)

print("data shape ",data.shape,type(data))


# DISPLAY SOME SAMPLES IMAGES  OF ALL THE CLASSES

num_of_samples = []

cols = 5

num_classes = noOfClasses

fig, axs = plt.subplots(nrows=num_classes, ncols=cols, figsize=(5, 300))

fig.tight_layout()

for i in range(cols):
    for j,row in data.iterrows():

        x_selected = X_train[y_train == j]

        axs[j][i].imshow(x_selected[random.randint(0, len(x_selected)- 1), :, :],
cmap=plt.get_cmap("gray"))

        axs[j][i].axis("off")

        if i == 2:

            axs[j][i].set_title(str(j)+ "-"+row["Name"])

            num_of_samples.append(len(x_selected))

#DISPLAY A BAR CHART SHOWING NO OF SAMPLES FOR EACH CATEGORY

print(num_of_samples)

plt.figure(figsize=(12, 4))

plt.bar(range(0, num_classes), num_of_samples)

plt.title("Distribution of the training dataset")

plt.xlabel("Class number")

plt.ylabel("Number of images")

plt.show()


# PREPROCESSING THE IMAGES

def grayscale(img):

    img = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    return img

def equalize(img):

    img =cv2.equalizeHist(img)

    return img

def preprocessing(img):
    img = grayscale(img)     # CONVERT TO GRAYSCALE
    img = equalize(img)      # STANDARDIZE THE LIGHTING IN AN IMAGE
    img = img/255            # TO NORMALIZE VALUES BETWEEN 0 AND 1 INSTEAD OF 0 TO 255
    return img

X_train=np.array(list(map(preprocessing,X_train)))  # TO IRETATE AND PREPROCESS ALL IMAGES
X_validation=np.array(list(map(preprocessing,X_validation)))
X_test=np.array(list(map(preprocessing,X_test)))
cv2.imshow("GrayScale Images",X_train[random.randint(0,len(X_train)-1)]) # TO CHECK IF THE
TRAINING IS DONE PROPERLY

# ADD A DEPTH OF 1
X_train=X_train.reshape(X_train.shape[0],X_train.shape[1],X_train.shape[2],1)
X_validation=X_validation.reshape(X_validation.shape[0],X_validation.shape[1],X_validation.shap
e[2],1)
X_test=X_test.reshape(X_test.shape[0],X_test.shape[1],X_test.shape[2],1)

# AUGMENTATAION OF IMAGES: TO MAKE IT MORE GENERIC
dataGen= ImageDataGenerator(width_shift_range=0.1,   # 0.1 = 10%     IF MORE THAN 1 E.G 10
THEN IT REFFERS TO NO. OF  PIXELS EG 10 PIXELS
                            height_shift_range=0.1,
                            zoom_range=0.2,  # 0.2 MEANS CAN GO FROM 0.8 TO 1.2
                            shear_range=0.1,  # MAGNITUDE OF SHEAR ANGLE
                            rotation_range=10)  # DEGREES
dataGen.fit(X_train)
```

```
batches= dataGen.flow(X_train,y_train,batch_size=20)  # REQUESTING DATA GENRATOR TO
GENERATE IMAGES  BATCH SIZE = NO. OF IMAGES CREAED EACH TIME ITS CALLED
X_batch,y_batch = next(batches)


# TO SHOW AGMENTED IMAGE SAMPLES
fig,axs=plt.subplots(1,15,figsize=(20,5))
fig.tight_layout()


for i in range(15):
    axs[i].imshow(X_batch[i].reshape(imageDimesions[0],imageDimesions[1]))
    axs[i].axis('off')
plt.show()
y_train = to_categorical(y_train,noOfClasses)
y_validation = to_categorical(y_validation,noOfClasses)
y_test = to_categorical(y_test,noOfClasses)


#CONVOLUTION NEURAL NETWORK MODEL
def myModel():
    no_Of_Filters=60
    size_of_Filter=(5,5) # THIS IS THE KERNEL THAT MOVE AROUND THE IMAGE TO GET THE
FEATURES.
                         # THIS WOULD REMOVE 2 PIXELS FROM EACH BORDER WHEN USING 32 32 IMAGE
    size_of_Filter2=(3,3)
    size_of_pool=(2,2)  # SCALE DOWN ALL FEATURE MAP TO GERNALIZE MORE, TO REDUCE
OVERFITTING
```

```
    no_Of_Nodes = 500   # NO. OF NODES IN HIDDEN LAYERS
    model= Sequential()

model.add((Conv2D(no_Of_Filters,size_of_Filter,input_shape=(imageDimesions[0],imageDimesion
s[1],1),activation='relu')))  # ADDING MORE CONVOLUTION LAYERS = LESS FEATURES BUT CAN
CAUSE ACCURACY TO INCREASE
    model.add((Conv2D(no_Of_Filters, size_of_Filter, activation='relu')))
    model.add(MaxPooling2D(pool_size=size_of_pool)) # DOES NOT EFFECT THE DEPTH/NO OF
FILTERS


    model.add((Conv2D(no_Of_Filters//2, size_of_Filter2,activation='relu')))
    model.add((Conv2D(no_Of_Filters // 2, size_of_Filter2, activation='relu')))
    model.add(MaxPooling2D(pool_size=size_of_pool))
    model.add(Dropout(0.5))


    model.add(Flatten())
    model.add(Dense(no_Of_Nodes,activation='relu'))
    model.add(Dropout(0.5)) # INPUTS NODES TO DROP WITH EACH UPDATE 1 ALL 0 NONE
    model.add(Dense(noOfClasses,activation='softmax')) # OUTPUT LAYER
    # COMPILE MODEL
    model.compile(Adam(lr=0.001),loss='categorical_crossentropy',metrics=['accuracy'])
    return model


# TRAIN
model = myModel()
```

```
print(model.summary())

history=model.fit_generator(dataGen.flow(X_train,y_train,batch_size=batch_size_val),steps_per_
epoch=steps_per_epoch_val,epochs=epochs_val,validation_data=(X_validation,y_validation),shuff
le=1)


# PLOT

plt.figure(1)

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])

plt.legend(['training','validation'])

plt.title('Model loss')

plt.xlabel('epoch')

plt.ylabel(loss’)

plt.figure(2)

plt.plot(history.history['accuracy'])

plt.plot(history.history['val_accuracy'])

plt.legend(['training','validation'])

plt.title('Model Acurracy')

plt.xlabel('epoch')

plt.ylabel(‘acurracy')

plt.show()

score =model.evaluate(X_test,y_test,verbose=0)

print('Test Score:',score[0])

print('Test Accuracy:',score[1])
```

```
# STORE THE MODEL

model.save(r'Downloads/TSRv3.h5’)

cv2.waitKey(0)
```

Traffic Sign Classification Code

```
#!/usr/bin/env python


import os

import matplotlib.pyplot as plt

import cv2

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

import rospy

import tensorflow as tf

from cv_bridge import CvBridge, CvBridgeError

#load the trained model to classify sign

from keras.models import load_model

from PIL import Image

from sensor_msgs.msg import Image

from sklearn.model_selection import train_test_split

from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPool2D

from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical

from sklearn.metrics import accuracy_score

from std_msgs.msg import String


model = load_model(r"/home/michalis111/catkin_ws/src/puzzlebotcode/scripts/TSRv2.h5")

#dictionary to label all traffic signs class.

classes = { 0:'Stop',

            1: 'End speed + passing limits',

            2: 'Turn right ahead',

            3: 'Ahead only' }


def grayscale(img):

  img = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

  return img


def equalize(img):

  img =cv2.equalizeHist(img)

  return img


def preprocessing(img):

  img = grayscale(img)

  img = equalize(img)

  img = img/255

  return img
```

```
class camera_1:

 def __init__(self):

  self.image_sub = rospy.Subscriber("/camera/image_raw", Image, self.callback)

 def callback(self,data):

  bridge = CvBridge()

  try:

   image = bridge.imgmsg_to_cv2(data, desired_encoding="bgr8")

  except CvBridgeError as e:

   rospy.logerr(e)

  font = cv2.FONT_HERSHEY_SIMPLEX

  threshold = 0.95

  msg = String()

  sign_image = np.copy(image)

  cropped_image = sign_image[260:410, 520:670] # Slicing to crop the image

  ncropped_image = np.asarray(cropped_image)

  ncropped_image = cv2.resize(ncropped_image, (32, 32))

  ncropped_image = preprocessing(ncropped_image)

  ncropped_image = ncropped_image.reshape(1, 32, 32, 1)

  cv2.putText(sign_image, "CLASS: " , (20, 35), font, 0.75, (0, 0, 255), 2, cv2.LINE_AA)

  cv2.putText(sign_image, "PROBABILITY: ", (20, 75), font, 0.75, (0, 0, 255), 2, cv2.LINE_AA)


  # PREDICT IMAGE

  predictions = model.predict(ncropped_image)
```

```
    classIndex = model.predict(ncropped_image)
    probabilityValue =np.amax(predictions)
    index = np.argmax(predictions)
    if probabilityValue > threshold:
      print("Predicted traffic sign is: ", classes[index])
      msg = classes[index]
      pub.publish(msg)
      return msg
    else:
      print("Predicted traffic sign is: No Sign")
      msg = "No Sign"
      print(msg)
      pub.publish(msg)
      return msg


if __name__ == '__main__':
    rospy.init_node('Sign Detection', anonymous=False)
    pub = rospy.Publisher('sign', String, queue_size=10)
    rate = rospy.Rate(10)
    camera_1()
    try:
      rospy.spin()
    except KeyboardInterrupt:
      rospy.loginfo("Shutting down")
    cv2.destroyAllWindows()
```

Web Application Code

```
#!/usr/bin/env python


import rospy
from sensor_msgs.msg import Image
from std_msgs.msg import String
from cv_bridge import CvBridge, CvBridgeError
from geometry_msgs.msg import Twist
from std_msgs.msg import Float32
from flask import Flask, render_template, Response,jsonify, send_file


app = Flask(__name__)
@app.route('/')
def index():
    return render_template('home.html',static_folder='static')


@app.route('/speed')
def motor_speed():
    # Get the motor speed value from the encoder
    wr = "{:.2f}".format(float(right))

    wl = "{:.2f}".format(float(left))

    linear = "{:.2f}".format(float(vel))

    angular = "{:.2f}".format(float(ang))

    # Return the speed value as JSON data

    return jsonify(rspeed=wr,lspeed=wl,linspeed = linear, angspeed = angular)


@app.route('/signimage')

def signimage():

    # Return the image file as a response

    return send_file('/home/michalis111/catkin_ws/src/puzzlebotcode/scripts/static/'+img,
mimetype='image/jpg')


@app.route('/lightimage')

def lightimage():

    # Return the image file as a response

    return send_file('/home/michalis111/catkin_ws/src/puzzlebotcode/scripts/static/'+limg,
mimetype='image/jpg')


def callbackvel(data):

   global vel

   global ang

   vel = data.linear.x

   ang = data.angular.z
```

```
def wlcallback(smsgl:Float32):

    global left

    left = round(smsgl.data,2)


def wrcallback(smsgr:Float32):

    global right

    right = round(smsgr.data,2)


def callbacksign(data):

    global sign

    global img

    sign = data.data

    img = sign+".png"


def callbacklight(data):

    global light

    global limg

    light = data.data

    limg = light+".png"


if __name__ == '__main__':

    rospy.init_node('web_interface', anonymous=False)

    sub = rospy.Subscriber("/cmd_vel",Twist,callback=callbackvel)

    subwl=rospy.Subscriber("/wl",Float32,callback=wlcallback)
    subwr=rospy.Subscriber("/wr",Float32,callback=wrcallback)

    subsing = rospy.Subscriber("/sign",String,callback=callbacksign)

    sublight = rospy.Subscriber("/light",String,callback=callbacklight)


    app.run(debug=True)
```

HTML Code (Web Application)

```
<!doctype html>

<html>

    <head>

        <title>WebUI</title>

        <link rel="stylesheet" href='/static/style.css' />

        <meta charset="UTF-8">

    </head>

    <body>

        <div class="dashboard">

             <p id="right-speed"></p>

              <p id="left-speed"></p>

             <p id="linear-speed"></p>

             <p id="angular-speed"></p>

             <img class="background" src="{{url_for('static', filename='dashboard.jpeg')}}" >

        </div>

              <img id="websign" src="{{ url_for('signimage') }}" >

              <img id="weblight" src="{{ url_for('lightimage') }}" >
```

```
          <!-- jQuery -->
          <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
          <!-- Bootstrap JS -->
          <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"></script>
          <script>
                    function updateSpeed() {
                       // Make an AJAX request to the Flask route
                       $.getJSON('/speed', function(data) {
                          // Update the content of the container element with the motor speed value
                          $('#right-speed').text(data.rspeed);
            $('#left-speed').text(data.lspeed);
            $('#linear-speed').text(data.linspeed);
            $('#angular-speed').text(data.angspeed);
                       });
                    }
      function updatesignImage() {
      // Get a new image URL from the Flask route
      var imageUrl = "{{ url_for('signimage') }}?t=" + new Date().getTime();
         $('#websign').attr('src', imageUrl);
            // Delay updating the 'alt' attribute for 2 seconds
      setTimeout(function() {
         $('#websign').attr('alt', ' ');
      }, 2000);
}
```

```
function updatelightImage() {
        // Get a new image URL from the Flask route
        var limageUrl = "{{ url_for('lightimage') }}?t=" + new Date().getTime();
            $('#weblight').attr('src', limageUrl);
                // Delay updating the 'alt' attribute for 2 seconds
        setTimeout(function() {
            $('#weblight').attr('alt', ' ');
        }, 2000);
}


        // Periodically update the motor image every 0.5 seconds
        setInterval(updatesignImage, 500);
        // Periodically update the motor image every 0.5 seconds
        setInterval(updatelightImage, 500);


            // Call the updateMotorSpeed function every 1 second to update the motor speed value
                    setInterval(updateSpeed, 1000);
            </script>
    </body>
</html>
```

CSS Code (Web Application)

```
.dashboard {
    position: relative;
```

```
  top: 0px;

  left: 0px;

  border: 1px solid #000000;

}


#websign {

  position: absolute;

  top: 504px;

  left: 288px;

  width:150px;

  height:150px;

}


#weblight {

  position: absolute;

  top: 511px;

  left: 1507px;

  width:135px;

  height:135px;

}

#right-speed{

  position: absolute;

  font-family: -apple-system, BlinkMacSystemFont, sans-serif;

  font-size: 1.5em;
  top: 550px;

  left: 1240px;

  color: white;

}

#left-speed{

  position: absolute;

  font-family: -apple-system, BlinkMacSystemFont, sans-serif;

  font-size: 1.5em;

  top: 550px;

  left: 622px;

  color: white;

}

#linear-speed{

  position: absolute;

  font-family: -apple-system, BlinkMacSystemFont, sans-serif;

  font-size: 8em;

  top: 220px;

  left: 810px;

  color: white;

}

#angular-speed{

  position: absolute;

  font-family: -apple-system, BlinkMacSystemFont, sans-serif;

  font-size: 5em;
  top: 240px;

  left: 450px;

  color: white;

}
```