MACHINE LEARNING ALGORITHMS FOR AUTONOMOUS DRIVING

Submitted in partial fulfillment of the requirements for the course

Department of Electrical and Computer Engineering
University of Windsor

ELEC-4000: Capstone Design Project

August 2, 2023

MACHINE LEARNING ALGORITHMS FOR AUTONOMOUS DRIVING

By

SHABOW, Mario
BYRA, Patrick
AKRAM, Peter
YOUSSIF, Youssif

Team 13

Faculty Advisor: Dr. Alirezaee

Department of Electrical and Computer Engineering University of Windsor

August 2, 2023

MACHINE LEARNING ALGORITHMS FOR AUTONOMOUS DRIVING

“No action by any design team member contravened the provisions of the Code of Ethics and we hereby reaffirm that the work presented in this report is solely the effort of the team members and that any work of others that was used during the execution of the design project or is included in the report has been suitably acknowledged through the standard practice of citing references and stating appropriate acknowledgments.”

The presence of the authors’ signatures on the signature page means that they are affirming this statement.

SHABOW, Mario	[illegible]	August 2, 2023
BYRA, Patrick	[illegible]	August 2, 2023
AKRAM, Peter	Peter Akram	August 2, 2023
YOUSSIF, Youssif	[illegible]	August 2, 2023

ABSTRACT

(Mario)

Machine learning algorithms have become a pivotal role in autonomous navigation in the context of self-driving vehicles. These algorithms play a key role in allowing vehicles to process information from their surrounding environment, which facilitates crucial factors to allow for safe autonomous movement. Team 13 investigated various factors and conducted research on certain topics that are essential for autonomous navigation. These topics include mapping (SLAM), object detection, lane detection, and navigation. The primary goal of Team 13's research was to develop a working and effective system that performed autonomous navigation. To achieve this goal, a ROS-based mobile robot equipped with a LiDAR sensor [12] and a camera module [11] was utilized. These two components allowed the robot to perceive and interpret its surrounding environment accurately. With the robot capturing data from its environment, it was able to construct a virtual map by using simultaneous localization and mapping techniques. This map was portrayed as a reference which aided the robot to understand its location relative to the objects surrounding it. To make the robot facilitate autonomous movement, the implementation of a network of nodes was used which coordinated the robot's actions. These nodes established communication between the sensors, which resulted in the robot navigating in its environment efficiently while also adapting to the changing conditions in real-time. In addition, Team 13 covered the critical characteristics of object detection. Machine learning techniques were employed for static and real-time object detection (see Appendix A, 1.2, 1.3). With the help of algorithms, the system was trained to recognize and categorize various objects in an environment such as stop signs, traffic lights, cars, trucks, motorcycles, and more. Similarly, machine learning techniques were used for lane detection (see Appendix A, 1.1), a crucial factor in autonomous navigation. This was done using algorithms such as the Hough transform, Canny edge detection, and other imaging techniques that identify lanes on the road. With the integration of machine learning algorithms for object and lane recognition, along with the LiDAR sensor and camera components being able to create a virtual map enabling the ROS-based mobile robot to navigate autonomously, Team 13 successfully developed an understanding for autonomous driving.

TABLE OF CONTENT

INTRODUCTION

(Youssif)

Background:

The primary aim of this project is to attain a thorough comprehension of the operational principles employed in autonomous navigation, leveraging machine learning techniques and other software applications, including RViz and Gazebo, on a mobile robot integrated with the Robot Operating System (ROS). Autonomous navigation can be simply defined as the process by which any mobile unit is able to accurately maneuver from its initial position to its desired destination, within an environment. For humans, we have our senses, such as our eyes, to see the environment we are in, process a route and navigate through it using instructions our organic processing unit, the brain, has formed. For vehicles, it is more demanding. The first challenge for autonomous navigation is for the mobile unit to understand the geographical makeup of its environment, as well as its location within it. This vital information is needed to provide detail of where the mobile unit is currently located, as well as where it would like to navigate to. The mobile unit must determine an optimal solution to navigate through the environment while avoiding static and dynamic obstacles. The second challenge is that the mobile unit must also be able to abide by traffic laws, and therefore must not only be able to detect and recognize regulatory traffic signs and lights, but act upon them as well. For example, while in navigation, the vehicle must be able to recognize a stop sign and come to a complete stop before moving when it is safe to. This process can be summed up as a 4-step plan: 1) sense, to collect real time data, 2) perceive, interpret the data, 3) plan, create the route, and 4) act, to allow the vehicle to navigate safely through the route.

The implementation of autonomous navigation through the design of the project will impact several groups of people. After completing the project and gaining familiarity from the experience made, the work done will be scaled up and implemented onto a Fiat 500 that is provided by the Mechatronics lab and approved by Dr. Alirezaee. This will require coordination with the sensors team, as our design will need different sensors that will be installed onto specific locations of the vehicle. While the primary focus of this project is not consumer-centric, the design can be modeled by ADAS (advanced drive assisted systems) teams, aiming to address road safety in assisted driving, and therefore enhancing public safety.

Objective:

This project’s main goal is to investigate, understand, and apply the various working principles and operations of autonomous driving capabilities onto a mobile robot embedded with ROS. This can be done by implementing the 4 main processing steps of autonomous driving discussed previously, which are sensing, perception, planning, and acting. The mobile unit must be composed of state-of-the-art sensing capabilities, and therefore will be equipped with several key sensors. As these sensors collect data from the physical world, the specified data must be interpreted. This would aid the system in understanding its environment, localization, and the ability to recognize or track objects. The third step of the design will be planning a route for the mobile unit to follow. This will be composed of a known starting position within the environments, as well as a specified destination for it to navigate to. Finally, the last step of the design will be a summation of the previous steps to display the mobile unit’s ability to physically move from one location to another, while detecting and avoiding obstacles.

Scope:

To understand the scope of this project, we must understand its goal, which is to investigate, understand, and apply the various working principles and operations of autonomous driving capabilities. To achieve our design goals, the team worked together and in parallel to investigate different aspects of the project. The first step included the extraction of data from the sensors, which consist of a single camera and a LiDAR system. Using LiDAR, the team was able to create a virtual environment and identify both static and dynamic obstacles. Within this environment, the mobile unit was also localized, a destination was selected, the mobile unit calculated the shortest path to get there, and avoided obstacles detected by the LiDAR system. Additionally, machine learning algorithms, specifically computer vision techniques, were applied for object detection and recognition, and lane detection. Input data was acquired from the camera lens, while the OpenCV based codes were implemented. The design of the OpenCV algorithms were designed in house, while the SLAM, mapping and localization capabilities of the robot were developed using RViz and Gazebo.

BENCHMARKING

(Mario)

Over the years, technology has gone through constant evolutions, which catalyzed a revolution that has transformed our world. It has forced the shift from non-renewable energy sources such as fossil fuels, to sustainable alternatives such as solar panels and wind turbines. Similarly, the conventional gas-based engines have begun to be replaced by electric batteries, which shows how technology constantly adapts to enhance its impact on the environment. In the area of autonomous navigation, cutting-edge technology is maintained to achieve the goal of safe and non-restricted navigation. The state-of-the-art technology and techniques that are implemented in autonomous vehicles are composed of critical components such as:

Deep Learning Models:

A key challenge in autonomous driving lies with how accurate an environment is perceived. Deep learning algorithms play a vital role in achieving this by the utilization of object detection and lane segmentation. Convolutional Neural Network (CNN), and its variants, Faster R-CNN and YOLO (You Only Look Once), apply these algorithms to achieve accurate results in mapping [29].

Mapping Models:

Simultaneous Localization and Mapping (SLAM) is a critical aspect of autonomous driving because it allows the vehicle to generate a map based on its surroundings while simultaneously localizing itself inside on the map. High-end advanced SLAM algorithms such as IMLS-SLAM [29], utilize machine learning techniques that optimize the mapping accuracy.

Prediction Models:

Safety is a concern when it comes to autonomous driving as it is essential to maintain the safety of pedestrians and others on the road. Due to this, future behaviour of these groups of people must be sought out for. That is why Recurrent Neural Networks (RNN), and Long Short-Term Memory (LSTM) networks [30], are incorporated for predicting tasks that anticipate future trajectory of surrounding objects and obstacles.

Hardware:

Autonomous vehicles rely on different hardware components that allow them to operate successfully. Some key components are LiDAR, which has a high 3D laser scanner that can accurately provide precise information about its environment. Cameras, more specifically stereo cameras that are essential for object, lane, and traffic sign detection. Radar which utilizes radio detection and ranging sensors that monitor the velocity and distance of objects. Ultrasonic sensors which are used for parking and low speed maneuvering, and GPS, which is used in localization [31].

Many companies across the world are currently investing their time and money into autonomous navigation, whether it be manufacturing the vehicles or working on software to make the vehicle autonomous. Some of the major companies in the autonomous market are Waymo, Tesla, NVIDIA, and Baidu [32]. Waymo, which was formally known as the Google Self-Driving Car Project, has been a pioneer in autonomous navigation since 2009 and is still currently developing new technologies to help improve the autonomous industry [33]. Tesla, which has manufactured autonomous vehicles since 2014 [34], has paved the way for autonomous driving with its technology that allows the vehicle to be fully self driving. This enables autopilot which can allow the car to navigate through traffic and adjust the speed of the car if needed. Also braking in certain situations when needed. Although NVIDIA is not necessarily a manufacturing company, they are still supplying the resources needed in autonomous navigation. Many companies use NVIDIA's technology whether it's AI or their GPU's. These products are used in companies such as Tesla, Waymo, Audi, Mercedes-Benz, BMW, Toyota, and Baidu. Some assistance NVIDIA products provide to these companies are, hardware for autopilot and self driving features and enables advanced driver assistance systems [35]. Lastly Baidu, they are a leading Chinese technology company that have great interest in the autonomous field and are actively simulating tests with self-driving technology [36].

DESIGN CRITERIA, CONSTRAINTS, AND DELIVERABLES

(Patrick)

The design of the project was based on a small-scale model of potential sensors and features of which the final vehicle would need. The ROS based mobile robot was equipped with a lidar sensor, a front Raspberry Pi V2 camera, an Arduino to control motor functions, and a Raspberry Pi [3]. With the final objective of an autonomous vehicle in mind, the LiDAR sensor was used for 360-degree object detection as well as mapping of the surrounding environment of which the robot would be operating within. The front camera would be used for object detection and lane detection. In a real-world environment, many variables of the road are placed in front of the driver, such as road lines, street signs and traffic lights. Due to the nature of the variables in front of the vehicle the camera did not need to be high resolution, as modern object detection algorithms operate efficiently in low resolution settings. The raspberry pi would house the operating system of the ROS based mobile robot. The raspberry pi would be used to calculate the shorter routes from each waypoint, communicate with the roscore server, and house all modules to ensure proper functionality of the robot. The Arduino within the ROS based mobile robot was used to motor control, and synchronising of sensors.

A major constraint within this project was the overall efficiency of the codebase to ensure proper functionality of the robot. The robot would need to make real-time calculations to ensure a safe driving environment. The robot would recalculate the best path 3 times a second to ensure this. For a small-scale model of the system this will suffice as the environment the robot would be operating would be a closed and controlled system to ensure variables would be limited for testing purposes. When projecting this system onto a car the system would need to implement a stronger computing system as the variables of which a car would be driving are of much higher volume. A popular constraint would be cost for a system as many different sensors and computing systems need to be in place to mimic a real driver and operate at a higher efficiency than one. With regards to our group's issues with cost constraints, we were not faced with this issue. Our group operated under the guidance of the mechatronics department within the university, and they were gracious enough to offer equipment already within the lab for the group to use over the course of the project to mitigate any potential roadblocks the group would be faced with.

The final project deliverables were to produce a self-driving vehicle that would safely transverse a created and closed track. The ROS based mobile robot, after testing, was able to perform this such function. The ROS based mobile robot would first be manually driven first to create a two-dimensional landscape of the environment of which it will be operating within. Then the robot would have the map reloaded and its starting point within the map would be configured. After the ROS based mobile robot is configured within the map, the user can set a waypoint and orientation for the ROS based mobile robot to move to RViz software.

(Youssif)

To identify lanes accurately and effectively during real time navigation, a few important factors needed to be considered when designing and optimizing the algorithm. To simulate the conditions of an on-road vehicle, the perception of real-world data was gathered using a recording from a camera's POV. This hardware criteria would be acceptable since a camera was also implemented onto the robot's architecture. Using the camera's input, the lane detecting algorithm, should meet the following criteria: find the set of longest lines in the image, identify, and highlight both straight and curved lanes, differentiate the left lane from the right lane, and overlay highlighted areas of interest over the lane to test the quality of the algorithm.

Despite the algorithm's design, there are a few constraints that prevent full lane detection capability due to the hardware constraints of the camera. The first constraint is that the camera's range would pick up more data than needed, since the focus is only on the lane ahead, according to the design of the project. The more irrelevant data captured the more computational power and processing time it would take for the algorithm to run. Second, the camera's ability to adequately capture images is limited due to low light environments, after the sun sets, and weather infractions such as rain or fog. This would limit the camera's visibility, and reduce the image quality, causing some inaccuracy in the lane detection algorithm. Therefore, after considering the availability of resources, criteria, and constraints of the project, the lane detection algorithm will be designed to primarily work in clear weather during the daytime.

(Mario)

Another fundamental design criteria of computer vision is object detection. Being able to locate and identify objects in data sets that contain key figures of interest within an image or in real-time is achieved by utilizing an efficient object detection algorithm. The main goal for any object detection algorithm is being able to achieve high accuracy in locating and identifying objects.

Real time detection requires fast processing so the algorithm should be designed to operate in a way to have a high detection speed. Also, the algorithm should be robust, meaning it should be able to detect variations of object's appearance, operating in different lighting conditions, and background displays. By having the algorithm being robust, it can mimic a real-world scenario and perform in any environment. Not every image or real-time detection will have the best quality of resolution, that is why it is important for the algorithm to perform scaling. That way it can effectively process the image to different resolutions and still be able to detect an object.

During the process of designing an object detection algorithm, you need to factor real-world constraints to ensure the practicality and overall effectiveness of the algorithm. The algorithm must be designed to work within specific computational applications while also being capable of real-time processing. Additionally, the diversity of the training dataset can cause limitations on the algorithm's generalization capabilities. Energy consumption needs to be prioritized on battery powered systems because energy efficiency is important [37]. If a model requires a lot of energy, it can drain the battery rapidly causing the system to potentially power off. This is why algorithms need to be designed such that they prioritize lightweight models and have efficient operations to minimize energy consumption. Object detection requires a variety of sensors and cameras, and this can cause limitations. If a camera is operating at a low resolution, it might not detect all objects or fine detail. The algorithm must be designed in a way to prevent this from happening because it is a safety concern. If something is not being detected when it is supposed to, pedestrians or road users would be at risk [38]. By balancing all these constraints, an effective model can be designed that can cover many concerns one might have.

When taking all the constraints into consideration, the end deliverable is a trained object detection model that is capable of accurately detecting and classifying objects into different classes. The overall performance of the algorithm can be determined by evaluation metrics from the obtained results. Benchmarking the obtained results is important because the designed model can be compared with already existing models to see how similar or different the results are. With these deliverables, the results can be researched to advance the state-of-the-art tools in object detection in order to potentially improve them.

DESIGN METHODOLOGY

(Peter)

In order to accomplish the objective that was set out for this project, a very meticulous design methodology was to be completed. There were three main tools throughout the project that allowed for a demonstrative knowledge of using machine algorithms for autonomous driving. These tools were the ROS-based mobile robot, shown in Figure 1, the objection detection algorithm, and the lane detection algorithm.

The process in which the robot's final design was achieved started with having to decide which features were to be exhibited during the Capstone presentation. The features that were ultimately chosen were simultaneous localization and mapping (SLAM) and autonomous navigation. The hardware aspect of the robot was started by taking those features into account in order to implement accessories that we deemed were necessary to showcase those features during the Capstone presentation. Since the robot was provided to us by our academic advisor and the AI Robotic Center, many of these accessories had come preinstalled onto the machine. These accessories included the LiDAR sensor, the Raspberry Pi 4 Model B system, the OpenCR1.0 device, two DYNAMIXEL actuator systems, and two wheels. The additions and alterations made to the robot were reconfiguring how the LiDAR sensor was placed on the machine, connecting a Raspberry Pi Camera Module 2 to the Raspberry Pi and securing it onto the chassis of the robot [1], as well as making small improvements to the cable management and sturdiness of the components of the robot.

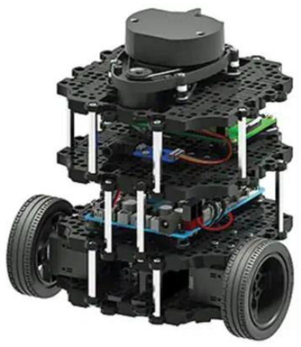


Figure 1: The ROS-based mobile robot used throughout the project

To begin working on the software aspect of the project, Ubuntu was required to be installed onto personal devices that were going to work on the development of the robot [2]. Once completed, Robot Operating System (ROS) was installed and updated on the same device to allow for access to software libraries and tools that would assist throughout the development and testing of the machine [3]. Within ROS, an extensive number of packages were installed to access and connect various physical components of the robot to the computer in use. The device was then connected to a cellphone's cellular hotspot as issues with the University of Windsor's network security would not allow for the processes that were required to exhibit autonomous driving of the device. In order for the onboard Raspberry Pi to control the rest of the physical components on the robot, such as the Arduino, the LiDAR sensor, and the actuator system, it also required its own installation of ROS. This was completed by burning a ROS image onto an SD card using the Raspberry Pi Imager software, which is seen in Figure 2, inserting the SD card into the Raspberry Pi, connecting the computer to a monitor, and launching it [4].

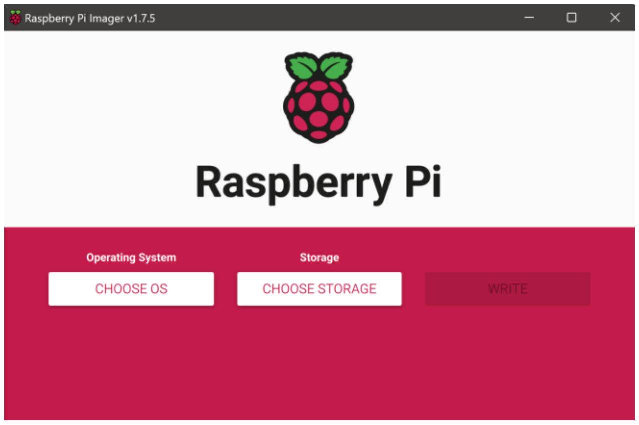
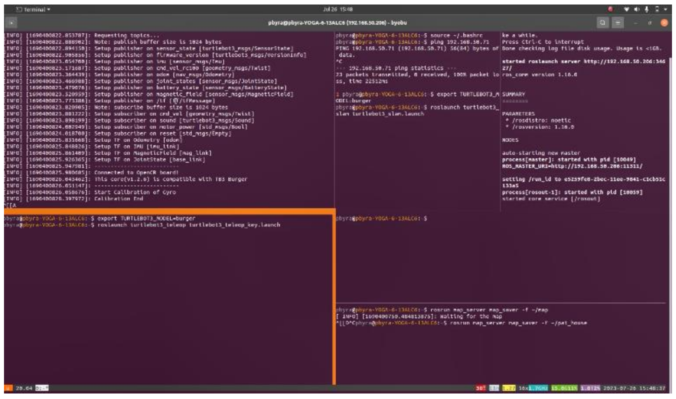


Figure 2: Raspberry Pi Imager used to burn a ROS image onto the SD card [5]

Once launched, the Raspberry Pi was connected to the cellphone's cellular network in order to be hosted by the same network as the device that will be connecting to it and configuring it. The roscore command, which allows for ROS-based nodes to communicate with one another, is run to allow for the PC and the Raspberry Pi to communicate. It should be noted that the nature of roscore features many terminals to be run simultaneously, which is shown in Figure 3. Due to this, byobu was installed onto the remote PC to allow for terminal splitting, increasing ease of use [6]. This communication authorizes control of the robot from the PC using Secure Shell (SSH). As the robot can now be controlled using an external device, a teleoperation node can be run in order to remotely control the movement of the machine. A SLAM node is also run in parallel to the teleoperation node that allows for the robot to be able to map its environment in a software called RViz [3]. Another software that is used by the robot is Gazebo. To start using Gazebo, its simulation packages were installed and launched using scripts within ROS [3]. RViz and Gazebo are extensively discussed in the Simulation Model Development of this report. This concludes the process in which the final design was achieved in referring to the ROS-based mobile machine.



```
[INFO] [1690400822.853787]: Requesting topics...
[INFO] [1690400822.888982]: Note: publish buffer size is 1024 bytes
[INFO] [1690400822.894150]: Setup publisher on sensor_state [turtlebot3_msgs/SensorState]
[INFO] [1690400822.985856]: Setup publisher on firmware_version [turtlebot3_msgs/VersionInfo]
[INFO] [1690400823.054700]: Setup publisher on imu [sensor_msgs/Imu]
[INFO] [1690400823.171687]: Setup publisher on cmd_vel_rc100 [geometry_msgs/Twist]
[INFO] [1690400823.364439]: Setup publisher on odom [nav_msgs/Odometry]
[INFO] [1690400823.466988]: Setup publisher on joint_states [sensor_msgs/JointState]
[INFO] [1690400823.479676]: Setup publisher on battery_state [sensor_msgs/BatteryState]
[INFO] [1690400823.520959]: Setup publisher on magnetic_field [sensor_msgs/MagneticField]
[INFO] [1690400823.773386]: Setup publisher on /tf [tf2_msgs/TFMessage]
[INFO] [1690400823.820905]: Note: subscribe buffer size is 1024 bytes
[INFO] [1690400823.881222]: Setup subscriber on cmd_vel [geometry_msgs/Twist]
[INFO] [1690400823.898199]: Setup subscriber on sound [turtlebot3_msgs/Sound]
[INFO] [1690400824.002949]: Setup subscriber on motor_power [std_msgs/Bool]
[INFO] [1690400824.018708]: Setup subscriber on reset [std_msgs/Empty]
[INFO] [1690400825.831668]: Setup TF on Odometry [odom]
[INFO] [1690400825.848826]: Setup TF on IMU [imu_link]
[INFO] [1690400825.861489]: Setup TF on MagneticField [mag_link]
[INFO] [1690400825.926365]: Setup TF on JointState [base_link]
[INFO] [1690400825.947981]: -----------------------------
[INFO] [1690400825.980685]: Connected to OpenCR board!
[INFO] [1690400826.043462]: This core(v1.2.6) is compatible with TB3 Burger
[INFO] [1690400826.051147]: -----------------------------
[INFO] [1690400826.058676]: Start Calibration of Gyro
[INFO] [1690400828.397972]: Calibration End
^[[A
```

```
pbyra@pbyra-YOGA-6-13ALC6:~$ export TURTLEBOT3_MODEL=burger
pbyra@pbyra-YOGA-6-13ALC6:~$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

```
pbyra@pbyra-YOGA-6-13ALC6:~$ source ~/.bashrc
pbyra@pbyra-YOGA-6-13ALC6:~$ ping 192.168.50.71
PING 192.168.50.71 (192.168.50.71) 56(84) bytes of
 data.
^C
--- 192.168.50.71 ping statistics ---
23 packets transmitted, 0 received, 100% packet lo
ss, time 22512ms

1 pbyra@pbyra-YOGA-6-13ALC6:~$ export TURTLEBOT3_M
ODEL=burger
pbyra@pbyra-YOGA-6-13ALC6:~$ roslaunch turtlebot3_
slam turtlebot3_slam.launch
```

```
ke a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.50.206:346
27/
ros_comm version 1.16.0


SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.16.0

NODES

auto-starting new master
process[master]: started with pid [10049]
ROS_MASTER_URI=http://192.168.50.206:11311/

setting /run_id to e5259fe8-2bec-11ee-9841-c1cb51c
133a5
process[rosout-1]: started with pid [10059]
started core service [/rosout]
```

```
pbyra@pbyra-YOGA-6-13ALC6:~$
```

```
pbyra@pbyra-YOGA-6-13ALC6:~$ rosrun map_server map_saver -f ~/map
[ INFO] [1690400750.484813875]: Waiting for the map
^[[D^Cpbyra@pbyra-YOGA-6-13ALC6:~$ rosrun map_server map_saver -f ~/pat_house
```

Figure 3: Several Ubuntu terminals showing the calibration of the robot, Secure Shell, and commands such as roslaunch and rosrun

(Youssif)

The final design of the lane detection algorithm was achieved through a systematic step-by-step process of researching, understanding, and implementing computer vision techniques. To understand the basics, a literature review was conducted, leveraging computer vision methodologies from Dr. Jonathan Wu's ELEC 4490 Sensor and Vision Systems [23], [24], [25].

To successfully identify road lanes, it was decided that the applications of a Canny edge detector and a Hough Transform were essential. The canny edge detector would be applied to the input images greyscale equivalent, and used to detect the strong edges present in the image. This model was built from scratch as opposed to a built-in function [24]. The Hough Transform algorithm was subsequently employed to detect the two longest lines on the images outputted from the canny algorithm, effectively determining the lane boundaries [25].

After careful algorithm selection, it was decided that the software development would be done using Python, due to its extensive libraries, simplicity, and its growing popularity within the industry. Key libraries such as OpenCV, NumPy, and matplotlib were harnessed to assess certain aspects of the algorithm. The OpenCV library would be used for its vast range of computer vision tools, including image filtering, built in kernel operators, colour detection, while NumPy would be used for array or matrix-based operations. Additionally, matplotlib was used for its image processing and visualization-based tasks. Following the algorithm design, it would then be tested on images first, followed by video-based application for real time simulation. Due to the constraint of using a camera during weather or low light imitation, as well as criteria enforcing camera POVs from the front and center of the robot, videos were accumulated and tested upon, accordingly. The results would then be overlaid upon the input image or video, enabling an assessment of the algorithm's ability to identify lanes accurately.

(Mario)

The process of developing a working object detection model involves several key steps. As stated previously, understanding, and doing research on computer vision techniques is done in order to implement the knowledge into the algorithm. The model was also developed using Python on the PyCharm IDE because of its compatibility with the Python language and ability to easily create virtual machines for version control. The OpenCV library was utilized in the making of the object detection model. As mentioned previously, this library has a vast range of computer vision tools that allow for object detection. Due to this, incorporating this library would be beneficial for the model.

To detect objects, an important factor is a Convolutional Neural Network (CNN). A CNN is a deep learning model that is used for image recognition. It is highly efficient in detecting objects and its primary purpose is to extract features from input data sets. With that in mind, the creation of a CNN would be time consuming and with this project having a time constraint, a pre-trained model was imported[27,28]. With the pre-trained model being imported and the use of OpenCV's libraries, testing on various images and in real-time can begin. At first, the model was trained with static images datasets to see if any key features would be detected.

Some of the objects the model detected were cars, trucks, motorcycles, stop signs, and traffic lights. Once the model successfully detected key features on static images (see Appendix A ,1.2), testing began using the real-time model. With the implementation of a webcam, the model was able to detect objects in a real-time setting. The model was able to detect the same key features stated above but this time, an accuracy score was added to see how accurate it was (see Appendix A, 1.3).

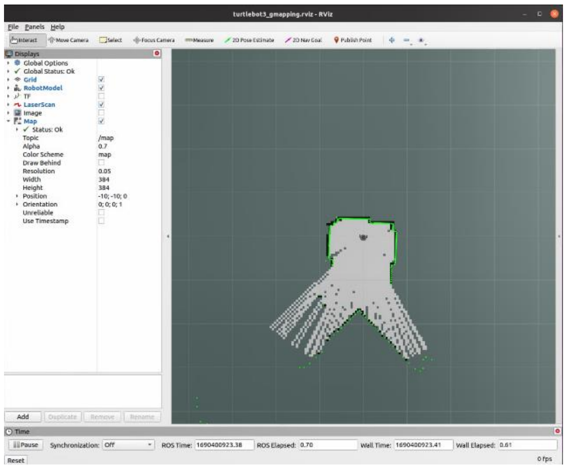
SIMULATION MODEL DEVELOPMENT

(Peter)

The development of simulation models for the ROS-based mobile robot, the object detection algorithm, and the lane detection algorithm was key in determining how the project would shape out to be, and if the objective of the project would be achieved. This development required the final design of each of the models, hardware and software alike, to be completed in order to proceed with the implementation and testing of these simulation models.

The ROS-based mobile robot had numerous simulation models that could have been chosen to exhibit a working knowledge of autonomous driving and to complete the objective of the project. The ones that were ultimately decided upon were to have simulation models for extrinsic mapping, intrinsic mapping, and autonomous navigation, the last of which is shown in Figure 5. As mentioned in the Design Methodology section of this report, the final design was achieved and two softwares, RViz and Gazebo, could be launched using the roscore command, due to them being ported to roscore [3]. These applications were instrumental to the simulation model development of this project. RViz is a software that can take in inputs from the LiDAR sensor, process them, and render a two-dimensional model, or map, of the dynamic and static surroundings of the robot. Figure 4 of the report shows RViz mapping out an environment that the robot is in. It can also use navigational nodes to process this same information in real time in order to navigate autonomously between two points in the shortest path required. On the other hand, Gazebo is a software that acts as a sandbox to erect walls and structures that can be intrinsically mapped by the same parameters that are set for the sensors on the robot in order to test and develop the accuracy and efficiency of these sensors.

The development of the simulation model for the extrinsic mapping began by running the roscore command on the computer to be used to control the robot. After doing so, the computer used Secure Shell to communicate through the network of choice (a cellular hotspot network due to its WPA2-Personal security protocol not found in all networks available) with the onboard Raspberry Pi. Once completed, RViz was then launched in a terminal using the rosrun command and the SLAM node was run using the roscore command [3]. Afterwards, the teleoperation node was run, and the computer was now able to control the movement of the robot using the cursor movement (WASD) keys. Using teleoperation, the robot was controlled remotely and was manually moved around its surrounding environment. The environment that was exhibited during the Capstone presentation was a hallway in a house. The doors into rooms were closed in order to not have the LiDAR sensor pick up and plot those as inputs onto the map. The robot was moved around the hallway while the LiDAR sensor was rotating and RViz used the inputs from this sensor to render a two-dimensional model of the hallway. A rosrun command was then run to save the map as PGM and YAML files [3]. These file types show a greyscale image of the map and node configuration parameters, respectively. The validation of the model and adjustments made to the parameters of the sensor are discussed in the Model Validation section of this report.



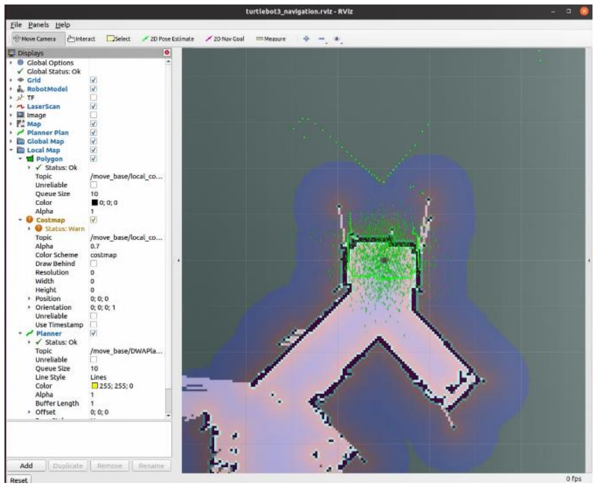
```
Global Options
Global Status: Ok
Grid            ✓
RobotModel      ✓
TF
LaserScan       ✓
Image
Map             ✓
  Status: Ok
  Topic           /map
  Alpha           0.7
  Color Scheme    map
  Draw Behind
  Resolution      0.05
  Width           384
  Height          384
  Position        -10; -10; 0
  Orientation     0; 0; 0; 1
  Unreliable
  Use Timestamp
```

```
Synchronization: Off   ROS Time: 1690400923.38   ROS Elapsed: 0.70   Wall Time: 1690400923.41   Wall Elapsed: 0.61
0 fps
```

Figure 4: The robot mapping out its environment within RViz

(Patrick)

With regards to intrinsic mapping, the software Gazebo was used. Gazebo is an environment sandbox software where users can create structures for simulation of operation of the robot. The application is shown in Figure 6. Within this software an enclosed room was created with many walls in order for simulation of the operation of the ROS-based mobile robot. The robot would run as it was within the environment created and the user would change very little to the operation of the system. The roscore server would be run from the remote laptop, the LiDAR sensor would be calibrated, and the room created would be mapped intrinsically through the lens of the software of which the robot believes itself to be within. The parameters that the LiDAR sensor was tuned to would be loaded within to create an experimental representation of what the system believes itself to be operating within. The environment would be mapped, and autonomous driving would be run in the same manner as if it was operating in the real plane.



```
Global Options
Global Status: Ok
Grid                ✓
RobotModel          ✓
TF
LaserScan           ✓
Image
Map                 ✓
Planner Plan        ✓
Global Map          ✓
Local Map           ✓
  Polygon           ✓
    Status: Ok
    Topic           /move_base/local_co...
    Unreliable
    Queue Size      10
    Color           0; 0; 0
    Alpha           1
  Costmap           ✓
    Status: Warn
    Topic           /move_base/local_co...
    Alpha           0.7
    Color Scheme    costmap
    Draw Behind
    Resolution      0
    Width           0
    Height          0
    Position        0; 0; 0
    Orientation     0; 0; 0; 1
    Unreliable
    Use Timestamp
  Planner           ✓
    Status: Ok
    Topic           /move_base/DWAPla...
    Unreliable
    Queue Size      10
    Line Style      Lines
    Color           255; 255; 0
    Alpha           1
    Buffer Length   1
    Offset          0; 0; 0
```

Figure 5: The robot navigating its environment within RViz

From Gazebo, we were able to perform simulations of what the live environment limitations were. For instance, in large environments, the LiDAR sensor would fail. The walls needed to be within 3 meters of the robots peripherals in order to create a clean map of the system. The ROS-based mobile robot also was quite safe in terms of its comfortability in its proximity to an obstacle, the robot refused to enter spaces it deemed too tight even if it was an area the robot could fit anyway. This radius was found to be approximately one-third of a meter in its surroundings.

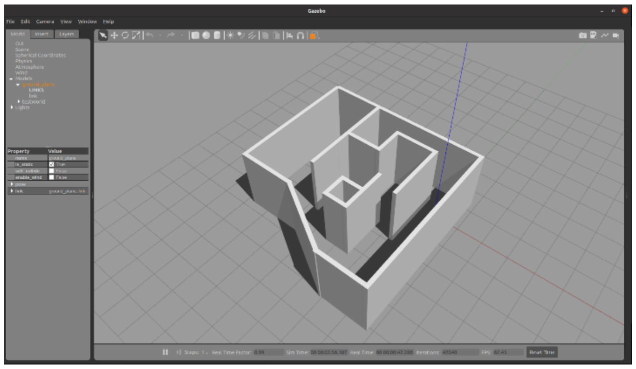


Figure 6: An intrinsic environment created with Gazebo

(Youssif)

The development of the lane detection algorithm was implemented using Python 3.10 on the PyCharm IDE. As mentioned in the design methodology of this report, the OpenCV, NumPy, and matplotlib libraries were imported for a wide range of image and computer vision processing tools. As per criteria, the lane algorithm would be designed to work on well lit and clear-weathered environments, for real time applications, to identify and high road lanes. Likewise, the criteria also stated that the development of the code should emulate the camera's POV on the robot, which is directly at the front and center. Before moving on to real time or video-based compatibility, the algorithm was developed using a single image for testing, as this would aid the development in terms of understanding the process and fundamental principles. In addition, this would allow refinement and optimization of the code more efficiently, as it would serve as a baseline performance metric, and a reference point for transitioning to video compatibility.

The first step of the algorithm was capturing or importing the image that would be used as a reference. However, the captured image displays a coloured portrait, which is made up of many pixels with different RGB scheme. The RGB scheme is used to depict the colour intensity of a pixel, based on how much red, green, and blue is used. The intensity varies on a scale from 0 to 255, since each of the RGB channels is 8 bit, the maximum value of a channel is 2^8=256, or 0 to 255. Since each pixel RGB scheme varies greatly, it is a common theme to convert the input image to its greyscale equivalent. This allows for much simpler and faster processing. The next step involves convoluting the grayscale image with gaussian white noise, to reduce noise and smoothen out the pixel intensity values. Noise can be described as a disturbance to the pattern of the pixel, and generated by many sources, such as thermal noise from the camera or flickering illuminations [23]. These types of noise, from hardware defects can be corrected, by applying the gaussian noise filter. It is like the averaging filter, except it adds more weight to the center pixel. The gaussian filter works by assigning the weighted value to the neighboring pixels with a mxm mask and computes the pixel intensity value in the middle of the filter as the average of those around it, and therefore removing noise defects. The equation below is used to approximate the weighted value of the filter, where $K = \frac{1}{2\pi}$, and the standard deviation, σ, determines how fast the weights decay, based on distance away from the center pixel [23]. For the algorithm deployed in the code, the mask size was set to 7x7, ensuring random pixel intensities caused by noise would have little effect on the effect central pixel of the mask, and a standard deviation of 0.2, which was based off trial and error and comparing different outputs based on how much irregularities, as such spikes, were present. Figure 7 below compares the effects of the Gray-scaling and gaussian filter to the input image.

$$g(x,y) = ke^{-\frac{(x+y)^2}{2\sigma^2}} \quad [23]$$

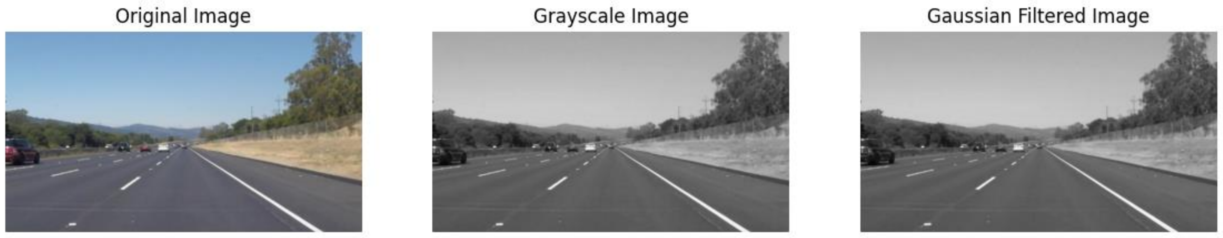


Figure 7: Comparing the original input with the grey scaled and gaussian filtered applications [21]

After the initial acquisition and enhancement of the captured image, the next step was to apply the canny edge detection algorithm. The purpose of this is to be able to detect image features, which in these cases can be classified as edges. These edges can be simplified as the pixel location of where the intensity starts to change, Figure 8. This can be easily visualized using the image below acquired from Dr. Wu's lectures. Any pixel which is sitting at the boundary of these two regions is considered an edge, due to the sharp change in intensity from one side to the other. This is easily identified due to the grey scaling that was applied and the noise filtering.

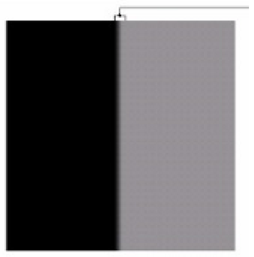


Image 8: Illustrating edges as pixels found between two boundaries [24]

The Canny based algorithm applied in the code was constructed based on two unique techniques, non maxima suppression, and double thresholding [24]. First, a 3x3 Sobel operator was applied on the gaussian filtered image, using the OpenCV library, and this would calculate the gradients magnitude and direction of the image. Furthermore, this would be applied horizontally and vertically, to find the change in pixel intensity in both directions. After acquiring and storing the location of the edge strength (magnitude) and direction, the pixels would be processed through the non-local maxima suppression function. The purpose of this technique is to thin out the edges, by checking if the center pixel is the local maxima across the gradient. If the center pixel is the largest, its intensity is kept unchanged, if it is not, it is discarded by being set to 0. This can be seen in the Figure 9 below, where the center pixel (q) of a 3x3 kernel is compared to the pixel intensities q and r, which are along its gradient [24].

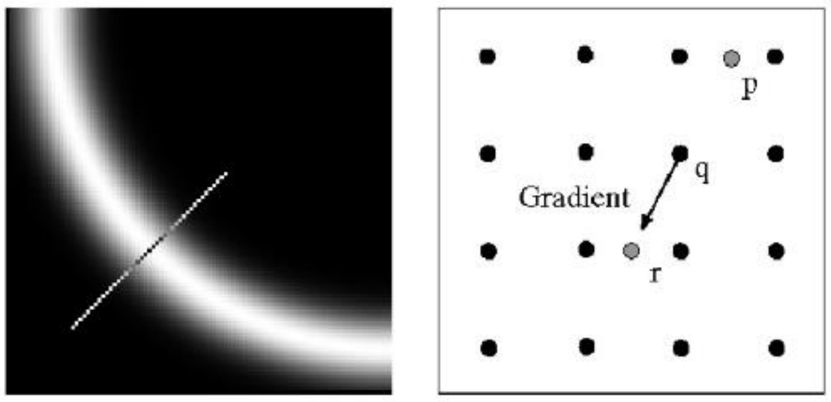


Figure 9: Non-maxima Suppression [24]

After applying the non-maxima suppression algorithm, the edges were now identified, but it would be time to separate strong edges from weak edges. This would be done by implementing a simple, yet effective technique known as double thresholding. This is done by applying two thresholds, a low threshold and a high threshold [24]. This algorithm would then intake the edge strength found in the non-maxima suppression and would place them in one of three cases. Case 1) edge strength under low threshold and is set to an intensity of 0 (black pixel), Case 2) edge strength above high threshold and is set to an intensity of 255 (white pixel), case 3) edge strength between low and high threshold, and therefore it would retained if and only if it is connected directly to a pixel above the high threshold, otherwise it is discarded [24].

In the algorithm developed by the team, the low and high thresholds were chosen to be 180 and 240, respectively, due to its effective filtering results, seen in the Figure 10 below.

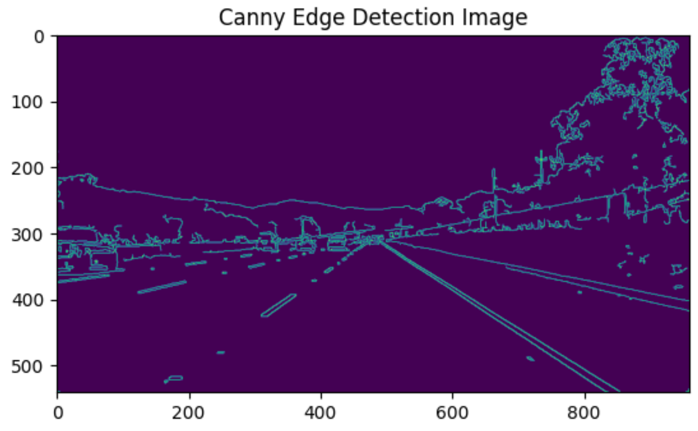


Figure 10: Result of the Canny Edge Detector

As seen by the image above, the canny edge detector picks up all the string and connected edges in the frame, from the POV of a camera placed at the front and center of a vehicle. The problem with this is that a lot of the information or edges perceived are irrelevant, since the focus of this project is on lane detection. A solution to isolate for the relevant area is to create a 4-side figure, resembling the lane in front of the vehicle, that would inform the algorithm where it should focus. This 4-sided figure will be constructed by highlighting the entire bottom width of the image (15% from the bottom left, to 95% for a video) and top width as a fraction of the bottom width, while the side lengths stay constant at around 60% from the bottom. Not only does it highlight the area of focus, but it also reduces the information intake of the system, therefore allowing for fast and efficient processing.

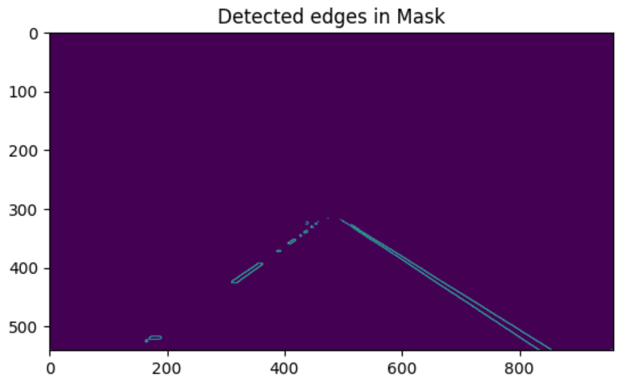


Figure 11: Relevant Area Mask

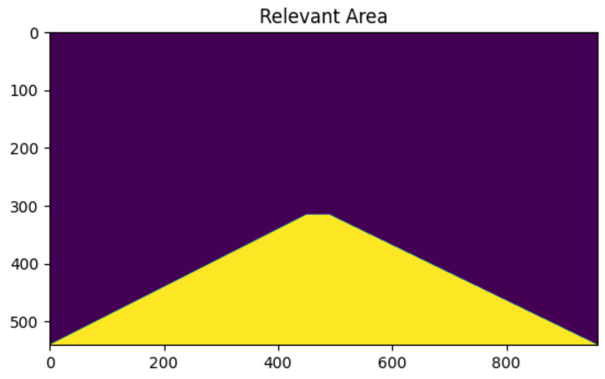


Figure 12: Detected Edges in Mask

As seen with Figure 11, all the background edges seen in Figure 10 have been removed, and the focus of the algorithm is only on the lane ahead. Using this new information, the Hough Transform will be applied. The purpose of this technique is to find and highlight geometric shapes, but in this specific project, it would be used to identify straight lines, corresponding to the lanes in the image. The working principles of a Hough transform are quite complex, so the function was implemented using OpenCV. This technique works by converting the image space (in polar coordinates using rho,ρ, and theta,θ rather than x and y) to the parameter space, where a line in image space is made up of many points, and each of those points correspond to a line in the parameter space. Therefore, each edge detected by canny is a point in image space and a sinusoid curve in parameter space [25]. An accumulator space is set up, where each edge detected votes for the longest line across the gradient direction [25]. The accumulator position with the local maxima is voted for as the longest line in the image. In the algorithm's design, key variables were selected for the Hough function. The ρ was set to 1, which means the distance resolution is set to 1 pixel, while the threshold was set to 3. This would ensure that multiple lines are selected, due to there being two lanes, as well as the broken up left lane, which is made up of multiple equally separated lines. As seen in the previous figure. Other parameters included setting the minimum number of pixels for a line to be valid was 10, while the maximum allowed gap between two lines was set to 10 pixels. This was done to ignore potential lines that were not lanes in the image. For each of the identified line segments, the function would return their end points (x_1,y_1,x_2,y_2), and a line of best fit would be drawn connecting these two points and overlaying them on the original image. The effects of the Hough Transform can be seen in Figure 13, below, and overlaid in Figure 14.

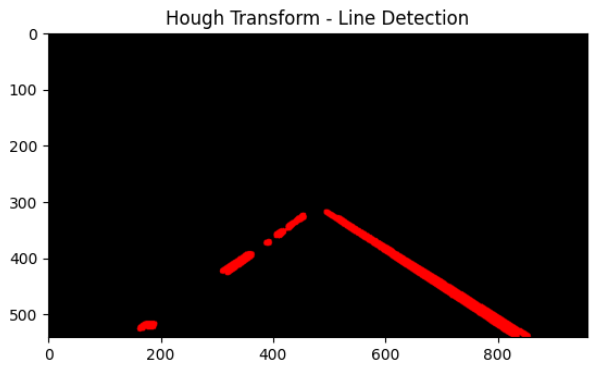


Figure 13: Line Detection using Hough Transform

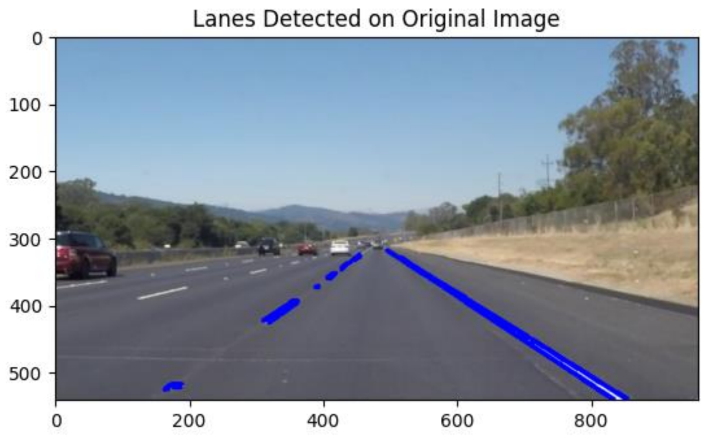


Figure 14: Final result of the image-based algorithm

Now that the process for lane detecting on a single static image has been completed, the algorithm must be readjusted for real time or video-based capability. This would require creating a function for each of the previous steps, while introducing a few new steps, and streamlining the processes by using a main function to tie it all together. This needs to be done since each frame of the video must be updated according to the vehicle's new position, as well as the new frame of data the camera had captured. Starting from the first step, this time, the function for the 4-sided relevant area mask is created, followed by the edge detection function, which converts the image to greyscale, applies Gaussian noise, and obtains the edges using the built in canny function. The Hough transform is also applied after identifying the edges within the relevant area, but this time, rather than identifying every single line, the left lane, which is composed of multiple broken lines, is overlaid with one single lines, matching the right line, but with a different slope [25]. This is done by obtaining the x_1,y_1,x_2,y_2 of both the left line and right line, via Hough transform, and identified whether it's the left lane or right lane based on the slope, and then superimposed onto the original image using the "Line_Overlay" function. The "main" function in the code then calls for all the discussed step of the lane detection algorithm to be applied to each frame of the video, and then superimpose the detected lanes upon the original input video, showcasing enhanced lane detection visualization [22]. Figure 15 below is an end product showcasing the ability of the full completed lane detection algorithm (See Appendix A, 1.1).

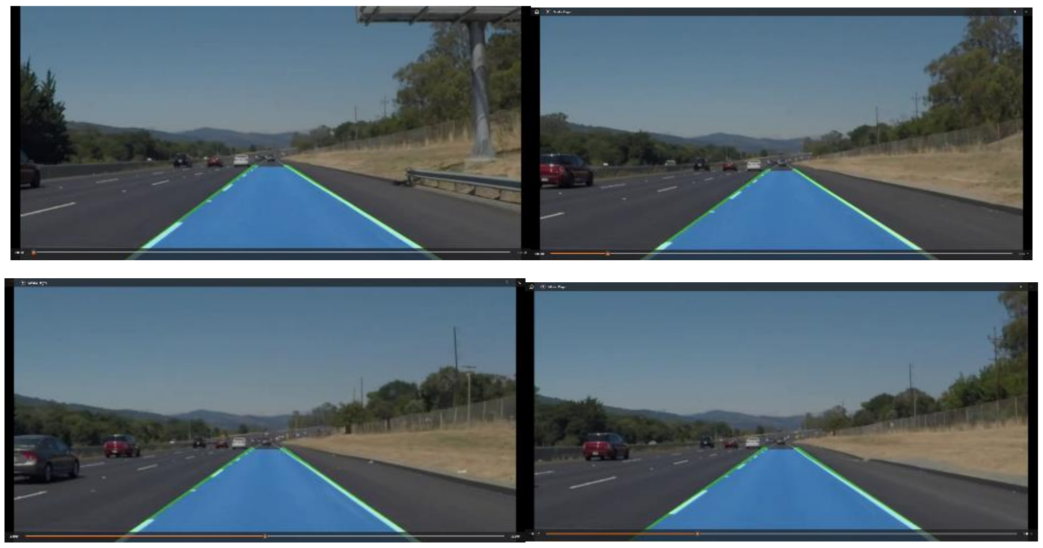


Figure 15: Showcasing the final result on different frames through a video

(Mario)

As mentioned above in the lane detection algorithm portion of the report, the object detection algorithm was also implemented using Python on the PyCharm IDE. The library that was imported to help aid the detection process was the OpenCV library due to its tools in computer vision. The object detection algorithm was designed to detect various key features from imported datasets that contained different object types. While being able to detect objects from static images, the model is also capable of detecting objects in real-time. This is achieved with the implementation of a webcam. Testing began first with an immense number of datasets that contained static images. This was because the model had to see different variations of objects while still being able to accurately detect the object. Once that was completed, the code was altered to fit the requirements for real-time testing. Trials began with having the webcam aimed at objects of interest.

The process in developing the object detection algorithm began with the static model. The first step was to obtain and import datasets that contained key images with a variety of different objects. By having a diverse dataset, the model can be trained with diverse data while still being able to successfully detect objects. The next step would be to import the OpenCV library to PyCharm. By downloading and importing this library, it enables you to use all its commands, specifically its computer vision ones. After the library is imported, the dataset containing the input image is read from the disk. The way the algorithm classifies each object into different classes is by creating a class name list [26]. In this class list, a variety of basic objects are included such as a person, car, truck, stop sign, traffic light, etc. For this model, an empty list is created that stores the class names and the file that contains the class names must be specified. When the model reads the class names from the file, it will store them in the empty list and will print the class name onto the console which will identify the detected object.

As mentioned in the design methodology, the CNN portion of the model was a pre-trained model that was imported. This was due to the complexity of training a model from scratch. The type of model that was imported was a MobileNet SSD [27,28]. This model was chosen over other pre-trained models such as the You Only Look Once (YOLO) model, due to its balance of accuracy, detection, and speed, as well as its compatibility with the Raspberry Pi. A path was specified for the imported files and a detection model that incorporated these paths. The detection model was made with the help of the OpenCV library, more specifically its Detection Neural network command.

Some factors within the model need to be accounted for. The input size for the model was set to 220x220 pixels. By having a smaller input size, it will have a faster inference time. It will require less computations and less memory, which in turn allows the model to process the frames faster, suitable for real-time detection. Also, with smaller input sizes, they typically tend to lose accuracy in detection, however, this size maintains a balance of accuracy and speed and works reasonably well in real-time applications. Another important factor to consider is the scaling factor. It is used in the normalization process of the data sets input image before sending it to the object detection model. Normalization is a step in deep learning algorithms which helps the model's performance. For this model, the scaling factor was set to 1/127.5. The value of 127.5 was chosen specifically because it is the midpoint of the 8-bit color range, 0 to 255 and by dividing the pixel by this scaling factor scales them in a range of -1 to 1. Setting the mean values for the input data to a value close to the scaling factor is important because it can help optimize the model during training.

After the factors mentioned above have been dealt with, the flag will be set to swap the red and blue channels for the input data. The reason behind this is that the model uses the swap command to preprocess that image data and converts it to a format compatible with the pre-trained model. In most models, including this one, the swapping is done to switch the order of the color channel. The normal color channel order that is used in deep learning algorithms is set to BGR, blue, green, and red. However, when using the OpenCV library's computer vision tools, it reads the color channel as RGB, red, green, and blue. To make sure the model reads it the correct way, which is BGR, you will set the flag to swap the red and blue color channels.

Once the color channels have been swapped to the correct channel, object detection on the input image using the pre-trained model can be performed. By using variables that store class identification of the object, store confidence score of the object, store bounding boxes highlighting the detected object, and setting the threshold value to a reasonable value such as 0.5, the model can scan the image and can identify if any objects are present. It will assign the object with a class identification labeling the object with a class name. Finally, a loop is created which detects how many objects are being detected and for however many objects there are, it will encircle them with a bounding box and give it a class identification. Static object detection is shown below in Figure 16.

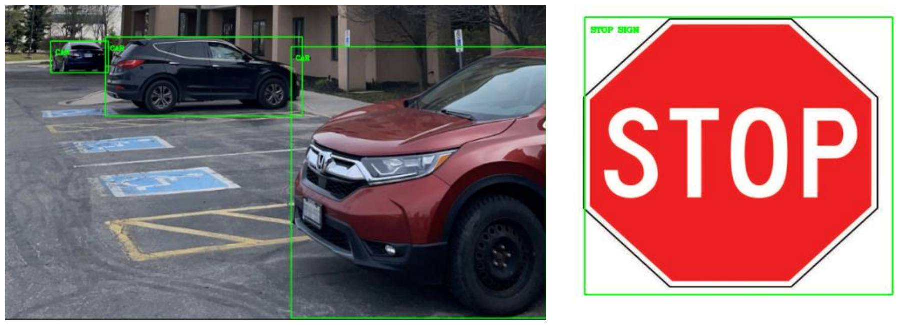


Figure 16: Static object detection performed on an image of multiple cars and a stop sign

With the model performing static object detection (see Appendix A, 1.2), real-time object detection can be done with the implementation of a webcam. With the use of OpenCV's library, the model can initialize a video capture that allows the camera to operate. For this model, the camera's property index was set to 3, which corresponds to the width of the video frame and 1280, which is the desired pixel width. The other property index was set to 4, which corresponds to the height of the video frame and 720, which is the desired height of the pixels. Ultimately, the resolution of the video was set to 1280x720. For the real-time object detection, a confidence score is implemented to see how accurate and precise the model is, (see Appendix A, 1.3). Real-time object detection is shown below in Figure 17.

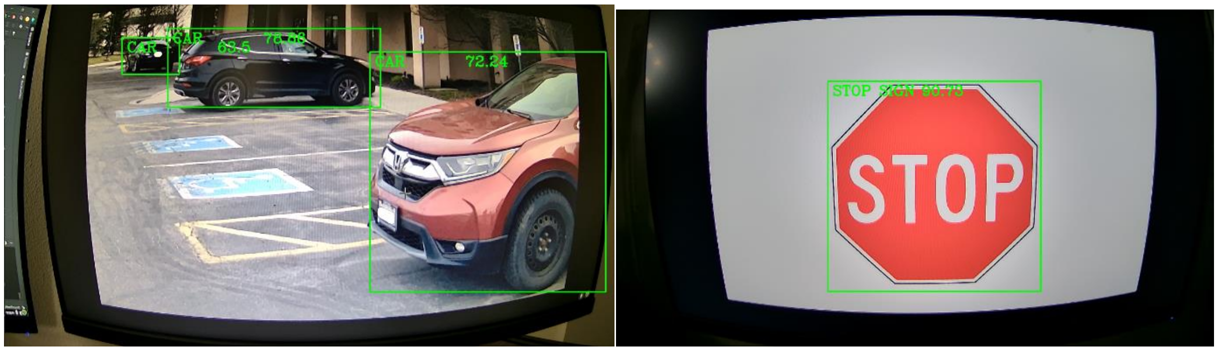


Figure 17: Real-time object detection performed on multiple cars and a stop sign with a confidence score

EXPERIMENTAL METHODS AND MODEL VALIDATION

(Peter)

A crucial step to creating designs and developing simulation models is to validate the models using experimental methods in order to refine the final design or model as much as possible. This refinement allows for a more efficient outcome in which the performance of the design or model can be measured based on its improvements in different metrics.

During the development of the simulation model for the extrinsic mapping created by the ROS-based mobile robot, the model required validation in order to test its accuracy on how well it can detect objects and accurately depict walls and obstacles in its two-dimensional renders in RViz. The first test of accuracy was made after completing the map of an environment. Once the map is saved, it can be viewed as a PGM file, and it can be analyzed on how accurate it is by seeing how the map was rendered compared to what the environment should look like based on the shapes and sizes of walls in contrast to one another, as well as obstacles that were or were not picked up. An example of a PGM file is Figure 18 below. In the YAML file that is also saved alongside the map, the mapping parameters can be adjusted in order to adjust how RViz processes the LiDAR sensor's inputs. These parameters are set within variables that include maxUrange and map update interval [3]. The variable maxUrange is set to the maximum distance in which the LiDAR sensor can detect an object, which is 8 metres [7]. Testing had to be done on the optimal map update interval value depending on the environment that was used. This variable parameterized the frequency of the rendering of the map, with a small value updating the map more frequently, and vice versa. In larger environments with a few or no obstacles, a larger value is more efficient to use, while a smaller value should be used in environments that are smaller and contain numerous obstacles. Additionally, the LiDAR sensor was tested amongst different surfaces to view if RViz would pick up on their presence. As expected, glass and other transparent or translucent surfaces were unable to be detected by the LiDAR sensor since the near-infrared lasers that are shot out of the sensor do not reflect and return to the sensor to calculate the distance away from the object, but instead, the laser passes through these types of objects and go undetected by the robot [8].

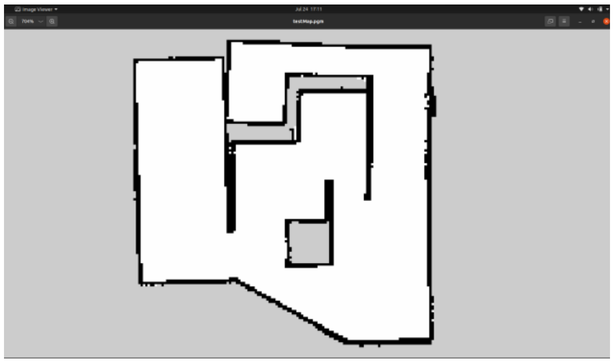


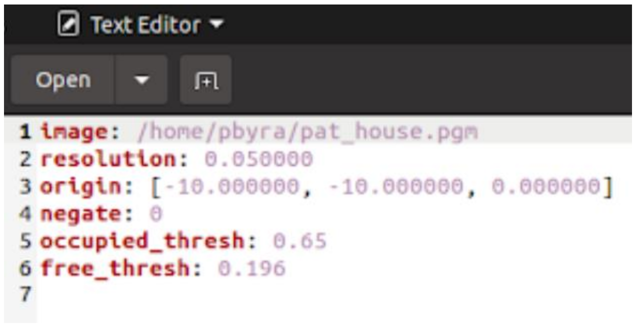
Figure 18: A PGM file that reveals a mapped environment from Gazebo

(Patrick Byra)

The ROS-based mobile robot featured an ability to create artificial environments using a software known as gazebo. Within the gazebo the user can erect structures such as a closed room with obstacles \ for the ROS-based mobile robot to run simulations within. The parameters of the LiDAR sensor would be loaded within the software to create a very similar output as if it were to be run within a real environment. The user would then need to adjust the YAML script that is in accordance with the world created to create a start point of the ROS-based mobile robot within the world, and then the user would control the robot in the same manner as if the robot was operating in real life. The user would start the roscore server, connect to the robot with SSH, run SLAM and teleoperation nodes, and control the robot within the environment. The information would be passed to RViz software, and a two-dimensional map would be created. The user would then be able to set way points within RViz and the simulated robot would calculate the shortest path and move between the way points.

The camera on the robot was used as a front sensor during operation of the robot. The user would first need to calibrate the camera using a checkerboard. The checkerboard was used due the sharp contrast in colours. The calibration process used a GUI, and the checkerboard would be held in front of the sensor adjusting slightly over time until the camera returns a calibration validation. The user from there can download the calibration to reuse later or can commit the calibration to be used as a default from that moment on. The Gazebo maps could then also be loaded within the camera. The camera would output information of the artificially created world instead of the environment it is truly in, allowing for further simulation. This is an important feature for the autonomous driving of the robot as when training the robot to track street lanes and traffic signs, the user could simply create these road features within the software and test the performance of the robot within this artificial environment rather than needing to dedicate real life resources to creating a proper test environment.

Using the autonomous driving nodes required the user to first create a two-dimensional map of the environment of which the ROS-based mobile robot would be operating in. The robot's autonomous driving was greatly influenced by the LiDAR sensor outputs as this sensor would be mainly used to detect obstacles within its surroundings. First the user would create the map based on the LiDAR sensor outputs and then reload this saved map within RViz. The user would then set the start point of the robot. The robot automatically assumes its start position as the position it was in when first mapping. The user would simply manually control the robot, see where the robot believes itself to be within the map, and adjust its start position and orientation. The user would then set the way point of the robot. The user would also set in when rotation the robot would end in. The ROS-based mobile robot would begin the drive to the selected waypoint, recalculating the shortest path three times a second. Successful autonomous driving was dictated by the ROS-based mobile robot's ability to avoid obstacles and reach the end of the set waypoint in an efficient manner, that is not constantly stopping to recalculate the path.



```
image: /home/pbyra/pat_house.pgm
resolution: 0.050000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Figure 19: A YAML file that shows the parameters of the LiDAR sensor based on a given PGM file

(Youssif)

The validation of the lane detection algorithm was quite simple, as it had been based on visualization. The purpose of this model was to detect and identify lanes during real time or video-based inputs effectively and efficiently. As seen by figure…it can be concluded that the algorithm successfully achieved its intended objective, thereby demonstrating its effectiveness and reliability in its performance.

(Mario)

The process of developing an object detection algorithm was completed with the use of experimental methods and model testing methods which helped the performance of the model. The first step for model testing was selecting a dataset that had a diverse variation of images that contained objects of interest. Preprocessing the data was performed in order to make the model robust. This was done by applying normalization and resizing to the model. Once the pre-trained model was imported [27,28], testing was done on static and real-time applications. With the model having a class list [26], that categorizes the objects into different class names, the model was trained with the dataset and labeled each object with a class name. Additionally, the model was trained to have a confidence score during real-time viewing, which tested the model's accuracy and precision. By experimenting with different images in the dataset, the model was able to provide an output when detecting objects.

DESIGN SPECIFICATIONS AND EVALUATION MATRIX

(Patrick)

Table 1: The design specifications of the ROS-based mobile robot

Parameter	**Value**
Maximum Translational Velocity	0.22 m/s
Maximum Rotational Velocity	2.84 rad/s
Maximum Payload	15kg
Size (L x W x H)	138mm x 178mm x 192mm
Weight	1kg
Threshold of Climbing	10mm or lower
Expected Operating Time	2h 30m
Expected Charging Time	2h 30m
SBC (Single Board Computer)	Raspberry Pi
MCU	32-bit ARM Cortex-M7 with FPU (216Mhz, 462 DMIPS)
Actuator	XL 430-W250
LiDAR	LDS-02
IMU	Gyroscope 3 Axis Accelerometer 3 Axis
Power Connections	3.3V / 800mA 5V/4A 12V/1A
Expansion Pins	GPIO 18 pins, Arduino 32 pin
Peripheral	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
DYNAMIXEL ports	RS485 x3, TTL x3
Programmable LEDs	User LED x4
Status LEDs	Board status LED x1 Arduino LED x1 Power LED x1
Buttons and Switches	Push Buttons x2, Reset Button x1, Dip Switch x2
Battery	Lithium polymer 11.1V 1800mAh/19.98Wh 5C
PC Connection	USB
Power Adapter	Input : 100-240V, AC 50/60Hz, 1.5A @max, Output: 12V DC, 5A

Table 2: The evaluation matrix for the project pertaining to the robot

Value	Description	Assessment
ROS-Based Mobile Robot		
LiDAR Mapping	Using the LiDAR sensor within the robot, how well does the robot create two dimensional maps of the area	The ROS-Based mobile robot was able to create accurate maps of the environment of which it operated within. The Robot was able to differentiate from moving obstacles and stationary ones to a high degree of efficiency
Route Calculations	The robots ability to calculate and continually update its proposed route during the operation of the machine	The robot was able to continually recalculate its best path at a rate of three times a second, rarely needing to halt operation to calculate a new path
RViz Accuracy	How well the software was able to communicate with the LiDAR sensor to produce the needed two-dimensional maps for autonomous driving	RViz communicated effectively with the ROS-Based Mobile Robot's LiDAR sensor to produce clean mappings of the area
New Obstacle Detection	How well the robot could update its proposed map in the case of a new obstacle found during operation that was not a part of the original map	The ROS-Based Mobile Robot accurately updated its map during the autonomous driving portion of its operation, being able to map new stationary and moving obstacles
Network Configuration	How effectively the communication of WIFI was during operation	The robot was able to quickly connect to the network it was configured on assuming it was a WPA2-Personal network security
Operating Time	How long the robot could be in operation	The robot maintained a 2h 30min operating time before the single battery within the machine needed to be recharged. The robot would beep at increasing intervals once the robot was below a threshold of 15% battery life

Table 3: The evaluation matrix for the project pertaining to the machine learning algorithms

Value	Description	Assessment
Machine Learning - Computer Vision		
Lane Detection Algorithm	How well the algorithm can detect and identify lanes on the road	The lane detection algorithm, is able to successfully identify lanes on the road and highlight the current lane for video based or real time navigation
Object Detection Algorithm	How well the algorithm can detect and identify objects statically and in real-time applications	The object detection algorithm was successful in identifying objects in static images and in real-time applications

BUDGET

(Peter)

Table 4: The initial budget of the project

Equipment Name	**Amount**	**Cost (CAD)**	**Amount Spent (CAD)**
GNSS Antenna [9]	3	449.97	449.97
GNSS Receiver Module [9]	2	439.98	439.98
Antenna Cable (N-Male to N-Male) [9]	3	29.97	29.97
Raspberry Pi 4 Model B [10]	2	153.90	0
15W Power Supply [11]	2	21.90	0
Total (including Tax)	12	1238.16	1039.51

Table 5: The true budget of the project

Equipment Name	**Amount**	**Cost (CAD)**	**Amount Spent (CAD)**
LDS-02 LiDAR Sensor [12]	1	260.27	0
Raspberry Pi 4 Model B [10]	1	76.95	0
15W Power Supply [11]	1	10.95	0
OpenCR1.0 [13]	1	285.21	0
USB2LDS Board [14]	2	79.63	0
DYNAMIXEL Actuator and Cable [15]	2	132.46	0
Wheel and Tire [16]	2	12.48	0
LIPO Battery and Charger [17]	1	79.10	0
USB Cable [18]	2	12.99	0
Micro HDMI to HDMI Adapter [19]	1	10.99	10.99
Power Cord [16]	1	28.40	0
32GB SD Card [20]	1	19.99	19.99
Chassis [16]	1	110.15	0
Total (including Tax)	16	1265.11	35.00

The budget of our project drastically changed as the scope of our project did. Initially, our project entailed for the purchases of GNSS antennas and receiver modules, as well as the cables needed for the devices to function. Two Raspberry Pi devices were originally going to be provided to us by Dr. Alirezaee as well. As the focus shifted towards the machine learning aspect of autonomous driving and the physical sensors component of the project was delegated to another Capstone team (Team 19), the initial equipment that was budgeted was no longer needed. Due to this, Dr. Alirezaee and the AI Robotics Center offered up the ROS-based mobile robot to complete the project and the only two required equipment that was needed to purchase were an SD card, as well as a cable that connects the robot to a monitor in order to be able to use the onboard computer. Therefore, the initial projection that the project would value at $1238.16 and would cost the group $1039.51 was inaccurate as the true project value is $1265.11 and only cost $35. Although there was a drastic difference with the budget, the $1200 supplied to us by the University would have covered all costs regardless of if the initial budget was put to use.

CONCLUSIONS

(Youssif)

In conclusion, the main objective of this project was to achieve a comprehensive and thorough understanding and application of key underlying principles in autonomous navigation. The goal of the autonomous driving aspect of the ROS-based mobile robot was to traverse a set track, based on manually created maps, with no collisions and efficient calculations of shortest path to set waypoint. The ROS-based mobile robot would need to be able to create accurate maps using its LiDAR sensor before completing the autonomous driving aspect of its operation. It was determined that the software RViz would be used in conjunction with the ROS-based mobile robot, due to its compatibility with ROS. The robot would need to be configured onto the same Wi-Fi network as the remote PC for communication with the server of which is ran from the remote PC. This was done using a simple network script that passed the credentials of the network to the robot. The only stipulation of the network connection was the need for it to be WPA2-personal connection as a domain authentication was impossible with this operating system. The remote PC would then need to SSH into the SBC within the ROS-based mobile robot to start the calibration of the LiDAR sensor. From the remote PC the user could then launch the SLAM, teleoperation, and autonomous driving nodes to begin the autonomous driving aspect of its operation. When launching the autonomous driving node, the user would need to set the path to the PGM file of which they would like to operate within. The ROS-based mobile robot would then need to be set to a default start point and a waypoint to where the user would like it to drive to would be set. The robot performed in an efficient manner during this autonomous driving stage. The robot would recalculate the journey three times a second in case of new obstacles within its path. At times the robot would halt its driving in the case of too many differences within the created map and the map driving within at this moment. This was a rare occurrence and only occurred if the robot's start point was incorrectly configured, user error, or if the number of new obstacles that were introduced into the environment was too vast, in the case of testing how well the robot can detect new obstacles. In the case of moving this technology into a real car, a much stronger CPU would need to be used to ensure no lagging within the operation of the vehicle as the stakes of real-world driving is of a much higher concern.

The goal of the lane detection ability of this project was to successfully identify lanes on the road during real time navigation. As stated in the Design Criteria, Constraints, and Deliverables, a few key factors and underlying principles would need to be established to emulate this task. First, real world data would be gathered, assuming the use of a camera to capture the world parameters and convert them into the image space. Furthermore, the camera would be assumed to be mounted at the front and center of the car, as this is the most effective area to successfully guarantee lane capturing. Next, the camera capabilities would be considered, since it would be ineffective during low light day times, or severe weather conditions, such as rain or fog. By taking all these factors into consideration, the desired performance of the lane detection algorithm was designed to work specifically during the daytime, with clear weather, and non sharp curved lanes. Following these desired performance specifications, the lane detection algorithm was developed using the following key techniques: Gray-scaling, Gaussian white noise filter, Canny edge detector, and the Hough Transform. The algorithm would then be tested on a static image emulating the desired environment and performance requirement, before being optimized for video and real time navigation compatibility. As seen by the final results, in Figure 15, it can be concluded that the measured performance has successfully met the desired performance requirements.

In addition, the objective for the object detection model was successful in this project due to the model being able to identify and classify various objects. The goal for the detection model was to accurately locate and identify various objects within a dataset which would simulate a real-world application such as an autonomous vehicle. By importing various datasets containing diverse images, the model was able to detect objects of different variation allowing it to be familiar within different environments. The development of the model took place with the use of the OpenCV library within the PyCharm IDE. By importing a pre-train model [27,28] and the use of OpenCV's computer vision commands, a working model that can detect static images and real-time applications was developed.

BACK MATTER

References

[1] “Getting started with the Camera Module,” Projects.raspberrypi.org, https://projects.raspberrypi.org/en/projects/getting-started-with-picamera (accessed Mar. 24, 2023).

[2] “Install Ubuntu desktop,” Ubuntu, https://ubuntu.com/tutorials/install-ubuntu-desktop#1-overview (accessed Feb. 9, 2023).

[3] “Wiki,” ros.org, http://wiki.ros.org/ (accessed Feb. 14, 2023).

[4] Ed, “Install ROS Noetic on Raspberry Pi 4,” roboticsbackend.com, https://roboticsbackend.com/install-ros-on-raspberry-pi-3/ (accessed Feb. 16, 2023).

[5] Raspberry Pi, “Raspberry Pi OS,” Raspberry Pi, https://www.raspberrypi.com/software/ (accessed Feb. 16, 2023).

[6] “Download and Install,” Byobu, https://www.byobu.org/downloads (accessed Mar. 2, 2023).

[7] “Robotis e-Manual,” Manual, https://emanual.robotis.com/ (accessed Mar. 28, 2023).

[8] “Radar and LiDAR Comparison,” Ti developer zone, https://dev.ti.com/tirex/explore/ (accessed July. 2, 2023).

[9] SparkFun Electronics, https://www.sparkfun.com/products/ (accessed Apr. 6, 2023).

[10] “Raspberry pi 4 model B/4GB,” PiShop.ca, https://www.pishop.ca/product/raspberry-pi-4-model-b-4gb/ (accessed Apr. 6, 2023).

[11] “Raspberry Pi 15W Power Supply, EU, black,” PiShop.ca, https://www.pishop.ca/product/raspberry-pi-15w-power-supply-eu-black/ (accessed Apr. 6, 2023).

[12] “360 Laser Distance Sensor LDS-01 (LIDAR),” ROBOTIS, https://www.robotis.us/360-laser-distance-sensor-lds-01-lidar/ (accessed Apr. 6, 2023).

[13] “OpenCR1.0,” ROBOTIS, https://www.robotis.us/opencr1-0/ (accessed Apr. 6, 2023).

[14] “Xevelabs product documentation,” Product: USB2LDS, http://www.xevelabs.com/doku.php?id=product%3Ausb2lds%3Ausb2lds (accessed Apr. 7, 2023).

[15] “DYNAMIXEL AX-12W,” ROBOTIS, https://www.robotis.us/dynamixel-ax-12w/ (accessed Apr. 6, 2023).

[16] “TurtleBot,” ROBOTIS, https://www.robotis.us/turtlebot/ (accessed Apr. 9, 2023).

[17] "LIPO Battery 11.1V 1800mAh LB-012," ROBOTIS, https://www.robotis.us/lipo-battery-11-1v-1800mah-lb-012/ (accessed Apr. 9, 2023).

[18] "GOOOA Short SuperSpeed USB 3.0 Extension Cable - A Left & Right Angle - Male to Female - Pack of 2," Amazon.ca: Electronics, https://www.amazon.ca/GOOOA-Short-SuperSpeed-Extension-Cable/dp/B083Q38K2N/ref=sr_1_6?crid=1GY8O68B4WBVZ&keywords=usb%2Bto%2Busb%2Bcables%2Btwo%2Bpack&qid=1691020349&sprefix=usb%2Bto%2Busb%2Bcables%2Btwo%2Bpack%2Caps%2C118&sr=8-6 (accessed Apr. 1, 2023).

[19] "Micro HDMI to HDMI Adapter 4K 60Hz,Snowkids Micro HDMI to HDMI Converter (Male to Female) for GoPro Hero,Camera with 3D ARC HEC UHD,Bi-Directional,Grey,1 Pack," Amazon.ca: Electronics, https://www.amazon.ca/Micro-Adapter-Snowkids-Female-Speed/dp/B08NVKZT5Y/ref=sr_1_11?crid=2KHP88NHQJ72M&keywords=micro%2Bhdmi%2Bto%2Bhdmi&qid=1691020400&sprefix=micro%2Bhdmi%2Bto%2Bhdmi%2Caps%2C101&sr=8-11 (accessed Apr. 2, 2023).

[20] "SanDisk Ultra 32GB Class 10 SDHC UHS-I Memory Card Up to 80MB, 2 Pack (Card)," Amazon.ca: Electronics, https://www.amazon.ca/SanDisk-Ultra-Class-UHS-I-Memory/dp/B07B8RNZJQ/ref=sr_1_7?crid=2YP6FLROZOSN4&keywords=sd%2Bcard%2B32gb&qid=1691020254&sprefix=sd%2Bcard%2B32gb%2Caps%2C95&sr=8-7 (accessed Apr. 6, 2023).

[21] N.Tanksale, "Finding Lane Lines — Simple Pipeline For Lane Detection.," Medium, Jun. 20, 2019. https://towardsdatascience.com/finding-lane-lines-simple-pipeline-for-lane-detection-d02b62e7572b [accessed Jul. 05, 2023].

[22] "CarND-LaneLines-P1/test_videos/solidWhiteRight.mp4 at master · udacity/CarND-LaneLines-P1," GitHub. https://github.com/udacity/CarND-LaneLines-P1/blob/master/test_videos/solidWhiteRight.mp4 [accessed Jul. 05, 2023].

[23] Dr.J.Wu, Class Lecture, Topic: "Image Enhancement and Acquisition." ELEC 4490, Department of Electrical and Computer Engineering, University of Windsor, Windsor, Jan 2023. [accessed Jun. 18, 2023].

[24] Dr.J.Wu, Class Lecture, Topic: "Image Features, Edge Detection, Canny Edge Detector." ELEC 4490, Department of Electrical and Computer Engineering, University of Windsor, Windsor, Jan 2023. [accessed Jun. 18, 2023].

[25] Dr.J.Wu, Class Lecture, Topic: "Image Feature Extraction & Hough Transform." ELEC 4490, Department of Electrical and Computer Engineering, University of Windsor, Windsor, Jan 2023. [accessed Jun. 22 2023].

[26] "Object-Detection/coco.names at main · sidpro-hash/Object-Detection," *GitHub*. https://github.com/sidpro-hash/Object-Detection/blob/main/coco.names (accessed July. 15, 2023).

[27] “Object-Detection/ssd_mobilenet_v3_large_coco_2020_01_14.pbtxt at main · sidpro-hash/Object-Detection,” *GitHub*. https://github.com/sidpro-hash/Object-Detection/blob/main/ssd_mobilenet_v3_large_coco_2020_01_14.pbtxt (Accessed 15 July. 2023).

[28] “Object-Detection/frozen_inference_graph.pb at main · sidpro-hash/Object-Detection,” *GitHub*. https://github.com/sidpro-hash/Object-Detection/blob/main/frozen_inference_graph.pb (accessed July. 15, 2023).

[29] L. Liu *et al.*, “Computing Systems for Autonomous Driving: State-of-the-Art and Challenges,” *IEEE Internet of Things Journal*, pp. 1–1, 2020, doi: https://doi.org/10.1109/jiot.2020.3043716. (accessed July. 29, 2023).

[30] “Recurrent Neural Networks and LSTM: Overview and Uses | Turing,” *www.turing.com*. https://www.turing.com/kb/recurrent-neural-networks-and-lstm (accessed July. 29, 2023).

[31] “State-of-the-art and trends of autonomous driving technology,” *researchgate*. https://www.researchgate.net/publication/328082602_State-of-the-art_and_trends_of_autonomous_driving_technology (accessed July. 29, 2023).

[32] C. Gray, “Top 10 companies developing autonomous vehicle technology,” *aimagazine.com*, Jun. 29, 2022. https://aimagazine.com/technology/top-10-companies-developing-autonomous-vehicle-technology (accessed July. 29, 2023).

[33] “Autonomous Driving Technology - Learn more about us,” *Waymo*. https://waymo.com/about/ (accessed July. 29, 2023).

[34] “Inside Tesla as Elon Musk Pushed an Unflinching Vision for Self-Driving Cars (Published 2021),” Dec. 06, 2021. Accessed: Aug. 03, 2023. [Online]. Available: https://www.nytimes.com/2021/12/06/technology/tesla-autopilot-elon-musk.html#:~:text=At%20Autopilot (accessed July. 29, 2023).

[35] “Self-Driving Cars Technology & Solutions from NVIDIA Automotive,” *NVIDIA*, 2019. https://www.nvidia.com/en-us/self-driving-cars/ (accessed July. 29, 2023).

[36] B. Inc, “Baidu to Add 200 Fully Driverless Robotaxis to Fleet in 2023, Aiming for World’s Largest Fully Driverless Ride-hailing Area,” *www.prnewswire.com*. https://www.prnewswire.com/news-releases/baidu-to-add-200-fully-driverless-robotaxis-to-fleet-in-2023-aiming-for-worlds-largest-fully-driverless-ride-hailing-area-301711296.html (accessed July. 29, 2023).

[37] A. M. • Bookmark +, “Battery Capacity an Overlooked Factor in Autonomous Vehicle Deployment, Performance,” *www.fleetforward.com*. https://www.fleetforward.com/345940/battery-capacity-an-overlooked-factor-in-autonomous-vehicle-deployment-performan (accessed July. 30, 2023).

[38] “Limitations of Self-Driving Cars | Staver Accident Injury Lawyers, P.C.,” *Staver*, Jul. 12, 2016. https://www.chicagolawyer.com/blog/limitations-of-self-driving-cars/ (accessed July. 30, 2023).

Appendix A

Relevant Software Code

1.1: Lane Detection Algorithm

(Youssif)

```
#Team 13

#Lane Detection using CV

#This program will emulate the POV of a camera mounted infront of a vehicle

#Thefore it will allow the camera to centre upon the curent lane of travel to ensure the system stays within these bounds


import cv2

#imported cv2 for computer vision related tasks (reading and processing videos or images)

import numpy as np

#imported numpy for linear algebra calculations involving matrices and array operations

import matplotlib.pyplot as plt


#Importing the image used for testing the program

#The image must be the POV of a centered front end camera

img = cv2.imread(r'C:\Users\youss\PycharmProjects\capstone\StaticImageTest.jpg')

#Convert the image to greyscale, this is done to easily recognize the change in intensity

#and idenitfy edges

grayscale_img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)


#Here begins the first step of the canny edge detector

# We applied a gaussian blue inorder to reduce noise and smoothen out the intesnity values

#This is done to prevent false edge detection caused by noise

Gaussian_noise_smoothing = cv2.GaussianBlur(grayscale_img, (7, 7), 0.2)


# original image

plt.subplot(1, 3, 1)

plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))

plt.title('Original Image')

plt.axis('off')
```

```
# grayscale image
plt.subplot(1, 3, 2)
plt.imshow(grayscale_img, cmap='gray')
plt.title('Grayscale Image')
plt.axis('off')

# Gaussian Filtered Image
plt.subplot(1, 3, 3)
plt.imshow(Gaussian_noise_smoothing, cmap='gray')
plt.title('Gaussian Filtered Image')
plt.axis('off')
plt.show()

#comparing the work to the function
Canny_edge_detector = cv2.Canny(Gaussian_noise_smoothing, 50, 150)

plt.imshow(Canny_edge_detector)
plt.title("Canny Edge Detection Image")
plt.show()

# create a zero mask that will be applied later
mask = np.zeros_like(Canny_edge_detector)

#As seen in the plot above, there are other white objects that are identified, but are not within our interested lane
#Therefore, we must eliminate them buy focusing only on our lane
#Next, we will need to define the area in which the lane is visible for a camera
#Since the camera is front and center mounted, we will assume it covers the coordinates below
#Note: these coordinates are not a cartesian system, but an Image Coordinate system for pixels obtained from the input image
#Define the 4 side figure to define the region of interest
heightY, widthX = img.shape[:2]
bottom_left=np.array([0, heightY])
top_left=np.array([449, 315])
top_right=np.array([490, 315])
bottom_right=np.array([widthX, heightY])
```

```
four_side_Area_of_Interest = np.array(
  [[
     tuple(bottom_left),
     tuple(top_left),
     tuple(top_right),
     tuple(bottom_right)
  ]],
  dtype=np.int32
)
#Next, we will convolute the area of interest on the zero mask, create a white 4 side figure to represent the lane
cv2.fillPoly(mask, four_side_Area_of_Interest, 255)
plt.imshow(mask)
plt.title("Relevant Area")
plt.show()

#The line below will be used to isolate for the lane in the canny image
Lane_in_Canny = cv2.bitwise_and(Canny_edge_detector, mask)
plt.imshow(Lane_in_Canny)
plt.title("Detected edges in Mask")
plt.show()

# Define the Hough transform parameters

#Dictoronary for the defined hough trasnsform paramters
hough_params = {
  'ρ': 1,
  'θ': 3.14 / 180,
  'threshold': 3,
  'min # of pixels for line': 10,
  'max # pixels between line': 10
}

# Run Hough Transform on the edge-detected image
#The Hough Transform will store the result in the form of line segments with start points (x1,y1) and
```

```
#end point segments of the line as (x2,y2)
Hough_Transform = cv2.HoughLinesP(
  Lane_in_Canny,
  hough_params['ρ'],
  hough_params['θ'],
  hough_params['threshold'],
  np.array([]),
  hough_params['min # of pixels for line'],
  hough_params['max # pixels between line'])



#Create a blank image to isolate and display the location of the detected lanes on
dup_img_1 = np.copy(img) * 0

dup_img_1 = np.zeros_like(
   img, dtype=np.uint8
)
dup_img_2 = np.zeros_like(
   img, dtype=np.uint8
)

# Draw lines on the blank image that will then be planted on the original image
for line in Hough_Transform:
  first_x_point, first_y_point, second_x_point, second_y_point = line[0]      #extract the current line detected in
the hough transform
  cv2.line(dup_img_1,  #on the blank image
          (first_x_point, first_y_point),  #draw line segment from this starting point
          (second_x_point, second_y_point),  #to this end point
          (255, 0, 0), 10)

plt.imshow(dup_img_1, cmap='gray')
plt.title("Hough Transform - Line Detection")
plt.show()
# merge the canny greyscale canny image to obtain the 3-channel RGB image
dup_img_2[:, :, 0] = Canny_edge_detector  #blue channel
```

```
dup_img_2[:, :, 1] = Canny_edge_detector  #green channel
dup_img_2[:, :, 2] = Canny_edge_detector  #red channel


plt.imshow(dup_img_2)
plt.title("dummy picture for testing")
plt.show()


# Draw the lines on the edge image
lanes_Detected_Before_AOI = cv2.addWeighted(dup_img_2, 0.8, dup_img_1, 1, 0)


#Draw the 4 side figure to represent the area of interest
lanes_Detected_In_AOI = cv2.polylines(lanes_Detected_Before_AOI, four_side_Area_of_Interest, True, (0, 0, 
255), 10)
plt.imshow(lanes_Detected_In_AOI)
plt.title("lanes_Detected_In_AOI")
plt.show()


# Draw the lines on the original image
def superimpose_lines(image, lines, color=[255, 0, 0], thickness=10):
  for line in lines:
     (
        p1, s1, pp2, ss2
     ) = line[0]
     cv2.line(
        image,
        (p1, s1),
        (pp2,ss2),
        color, thickness
     )


# Create a blank image to isolate and display the location of the detected lanes on
lanes_detected_img = np.copy(img)
superimpose_lines(lanes_detected_img, Hough_Transform, color=[255, 0, 0], thickness=5)


# isplay the original image with detected lanes
```

```
plt.imshow(cv2.cvtColor(lanes_detected_img, cv2.COLOR_BGR2RGB))
plt.title("Lanes Detected on Original Image")
plt.show()
# For Video/ real time navigation use the code below
#α=0.7 means the original image will have 70% influence in the final output.
#β=1.0 means the Hough lines will have 100% influence in the final outpu

def main(image, α=1, β=1., γ=0.):
  edges_img = edge_detection(image)
  masked_img = Relevant_Area(edges_img, Obatin_4sided_Polygon_Coords(image))
  houghed_lines = hough_lines(masked_img, ρ=1, θ=np.pi / 180, threshold=20, line_pixels=20,
distancepixels=180)

  # Perform weighted addition of the Hough lines image and the initial image directly
  output = cv2.addWeighted(image, α, houghed_lines, β, γ)
  return output

#To itterate through multiple frames, aka a video, we will creata a function for each step

def Obatin_4sided_Polygon_Coords(image):
   rows, cols = image.shape[:2]  # Get the number of rows and columns (height and width) of the input image
   # Define fractions of the image width and height to determine the vertices
   bw = 0.2
   tw = 0.5
   h = 0.5
   vone = [cols * bw, rows]  # Bottom-left vertex @ 15% width and at the bottom of the image
   vtwo = [cols * tw, rows * h]  # top-left vertex @ 45% width and 60% height
   vthree = [cols * (1 - bw), rows]  # Bottom-right vertex @ 85% width and at the bottom of the image
   vfour = [cols * (1 - tw), rows * h]  # top-right vertex @ 55% width and 60% height


   v = np.array([[vone, vtwo, vfour, vthree]], dtype=np.int32)
   return v

def edge_detection(img, low_threshold=180, high_threshold=240):
  # Convert the image to greyscale, this is done to easily recognize the change in intensity
  # and idenitfy edges
  gray_img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
  # We applied a gaussian blue inorder to reduce noise and smoothen out the intesnity values
  # This is done to prevent false edge detection caused by noise
  blurred_img = cv2.GaussianBlur(gray_img, (5, 5), 0) #5x5 kernel with 0 std
  #apply canny edge detector, which will be used to detect areas of rapid intensity change
  #values above high threshold are edges, values below are not, while values inbetween are edges based on their
negbours
  canny_img = cv2.Canny(blurred_img, low_threshold, high_threshold)
  return canny_img     # Return the resulting edge-detected image.


def Relevant_Area(img, vertices):


  # Create a blank mask image that is the same size as the original image
  mask = np.zeros_like(img)
  # Define the fill color (transparency) for the mask
  fill_color = (255, 255, 255, 100)  # Transparent white (R, G, B, Alpha)
  # create the white polygon figure on the blank image
  cv2.fillPoly(mask, vertices, fill_color)
  # AND the polygon figure and the canny img
  #if both pixels are non zero, the resulting pixel is retain its orginal value
  #otherwisse, it will be set to black
  masked_image = cv2.bitwise_and(img, mask)
  return masked_image


#The Hough Transform is a feature extraction technique used in computer vision
# it was used to detect lines, to identify lane segments in this code
# it converts a cartesian plane to the hough space, where a line in image is a point in hough
#ρ, rho, represents the perpendicular distance from the origin
#θ,theta, represents the angle (in radians) between the x-axis and the line perpendicular to the detected line.
#the hough transform will consdier all possible linea for an edge point, with each edge point voting for lines that
can go through it
#The accumulator array where votes are stored will have a threhold, and votes above this will be considered a line
with rho and theta
#that will be converted to cartesian coordinates, tielding the endpoints of a detected line
```

```
def hough_lines(img, ρ, θ, threshold, line_pixels, distancepixels):
  #the HoughLines function returns an array of line segments represented by their endpoints (x1, y1, x2, y2)
  lines = cv2.HoughLinesP(img, ρ, θ, threshold, np.array([]), minLineLength=line_pixels,
                        maxLineGap=distancepixels)
  hough_output = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
  #using the endpoints foind ine "lines", the slope function will create a line from the endpoint and
  #display it on the image
  hough_output = line_fitting(hough_output, lines)
  return hough_output



#the following function will be used to identify which lane is the left lane and which is the right lane from the
hough transform output
def find_lane_side(img_lines):
  #iterate over each line detected by the hough transform
  LL = [
     (
        (y2 - y1) / (x2 - x1),
        y1 - ((y2 - y1) / (x2 - x1)) * x1
     ) #calcaulted slope and y intercept of each line
            #consider only the line segments with non-vertical slopes (x1 != x2) and negative slopes (indicating the
left lane)
            for l in img_lines
            for (
        x1,
        y1,
        x2,
        y2) in l
            if x1 != x2 and (y2 - y1) / (x2 - x1) < 0]
  #
  #do the same but for right lane
  RL = [
     (
        (y2 - y1) / (x2 - x1),
        y1 - ((y2 - y1) / (x2 - x1)) * x1)
             for line in img_lines
            for (
        x1,
        y1,
        x2,
        y2) in line
            if x1 != x2 and (y2 - y1) / (x2 - x1) >= 0
 ]

 left_l = np.mean(LL, axis=0)
 right_l = np.mean(RL, axis=0)
 return left_l, right_l

#fitting the left and right lane lines to the detected lane segments and then drawing and highlighting
# these fitted lines on the input image
def line_fitting(img, l):
 #create a copy image
 img = img.copy()
 #store vertices of the polygon that will be filled to highlight the lane
 poly_vertices = []
 order = [
    0,
    1,
    3,
    2]
 (
    LL, RL
 ) = find_lane_side(l)
 # Iterate over the fitted lines (left and right) to draw them on the image
 for m, b in [LL, RL]:
    rows, cols = img.shape[:2]
    s1 = int(rows)
    ss2 = int(rows * 0.6)
    p1 = int((s1 - b) / m)
    pp2 = int((ss2 - b) / m)
    poly_vertices.append(
        (p1, s1)
    )
    poly_vertices.append(
        (pp2, ss2)
    )
    # Draw lane line on image
    Line_Overlay(img, np.array(
        [
            [[p1, s1, pp2, ss2]]])
                )
  # Rearrange the order of vertices to form a polygon for filling
  poly_vertices = [
    poly_vertices[i] for i in order
  ]
  cv2.fillPoly(img, pts=np.array([poly_vertices], 'int32'), color=(254, 135, 0))
  return cv2.addWeighted(img, 0.7, img, 0.4, 0.)

def Line_Overlay(img, lines, color=[0, 255, 0], thickness=10):
  for line in lines:
    for (
            p1,s1,pp2,ss2) in line:
        cv2.line(img,(p1, s1),(pp2, ss2),color,thickness)

fig, axes = plt.subplots(1, 2, figsize=(20, 10))

# Plot the input image
axes[0].imshow(img)
axes[0].set_title("Input Image")
axes[0].set_xticks([])
axes[0].set_yticks([])



# Plot the Lane Detection Result
result_img = main(img)
axes[1].imshow(result_img)
```

```
axes[1].set_title("Lane Detection Result")
axes[1].set_xticks([])
axes[1].set_yticks([])



input_video_path = r"C:\Users\youss\PycharmProjects\capstone\VideoTest.mp4"
output_video_path = r'C:\Users\youss\PycharmProjects\capstone\Result.mp4'

# Load the video file
cap = cv2.VideoCapture(input_video_path)

# Get the video's frame rate and size
fps = int(cap.get(cv2.CAP_PROP_FPS))
fw = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
fh = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

# Create the output video file
fourcc = cv2.VideoWriter_fourcc(*'mp4v')
out = cv2.VideoWriter(output_video_path, fourcc, fps, (fw, fh))

# Process each frame of the video and save the processed frames to the output video
while True:
  ret, frame = cap.read()
  if not ret:
    break

  # Apply the lane detection algorithm to the frame (replace main with your lane detection function)
  processed_frame = main(frame)

  # Write the processed frame to the output video
  out.write(processed_frame)
```
[21], [22], [23], [24], [25]

1.2: Static Object Detection Algorithm

(Mario)

```
# Team 13

# This program will perform object detection on a static image

# Import OpenCV library
import cv2

# Read the input image from the disk
image = cv2.imread('stop.jpg')

# Set the threshold value for object detection
threshold = 0.5

# Create an empty list to store class names in
EmptyClass = []

# Specify the file name that contains the class names for the dataset
ObjectNames = 'coco.names'

# Read the class names from the file and store them in the 'ObjectNames' list
with open(ObjectNames, 'rt') as f:
    EmptyClass = f.read().rstrip('\n').split('\n')

# Print the class names to the console
print(EmptyClass)

# Identify the paths from the pre-trained model configuration and files
TrainingPath = "ssd_mobilenet_v3_large_coco_2020_01_14.pbtxt"
ModelPath = "frozen_inference_graph.pb"

# Create a DetectionModel object using the pre-trained model files with OpenCV's computer vision commands
net = cv2.dnn_DetectionModel(ModelPath, TrainingPath)

net.setInputMean((127.5, 127.5, 127.5)) # Set the mean values for the input image data
net.setInputSize(220, 220) # Set the input size for the model to 220x220 pixels
net.setInputScale(1.0 / 127.5) # Set the scale factor for the input image data to 127.5
net.setInputSwapRB(True) # Set the flag to swap the Red and Blue channels in the input data to correct the color
channel
```

```
# Detect objects in the input image using the detection model
ObjectIds, confidence, boundingbox = net.detect(image, confThreshold=threshold)

# Print the detected Object IDs and bounding boxes to the console to highlight the object
print(ObjectIds, boundingbox)

# If objects are detected in the image
if len(ObjectIds) != 0:
  # Loop through all detected objects in the image
  for ObjectId, confidence, box in zip(ObjectIds.flatten(), confidence.flatten(), boundingbox):
    # Draw a rectangle encircling the detected object on the image, color set to green and thickness set to 2
    cv2.rectangle(image, box, color=(0, 255, 0), thickness=2)

    # Put the object name of the detected object above the rectangle, position of the object ID can be modified.
    cv2.putText(image, EmptyClass[ObjectId - 1].upper(), (box[0] + 5, box[1] + 35),
cv2.FONT_HERSHEY_PLAIN, 1.25, (0, 255, 0), 2)

# Display the image with the detected object in a window titled "Output"
cv2.imshow("Output", image)

# Wait for a key press (0 means wait indefinitely until a key is pressed)
cv2.waitKey(0)
```

[26], [27],[28]

1.3: Real-Time Object Detection Algorithm

(Mario)

```
# Team 13

# This program will perform object detection in real-time

# Import the OpenCV library
import cv2

# Set the threshold to detect objects
threshold = 0.5

# Initialize the webcam (Camera) for video capture
camera = cv2.VideoCapture(0)

# Set the resolution of the webcam to 1280x720 pixels
camera.set(3, 1280)
camera.set(4, 720)

# Create an empty list to store class names in
EmptyClass = []

# Specify the file name that contains the class names for the dataset
ObjectNames = 'coco.names'

# Read the class names from the file and store them in the 'ObjectNames' list
with open(ObjectNames, 'rt') as f:
    EmptyClass = f.read().rstrip('\n').split('\n')

# Print the class names to the console
print(EmptyClass)

# Identify the paths from the pre-trained model configuration and files
TrainingPath = "ssd_mobilenet_v3_large_coco_2020_01_14.pbtxt"
ModelPath = "frozen_inference_graph.pb"

# Create a DetectionModel object using the pre-trained model files with OpenCV's computer vision commands
net = cv2.dnn_DetectionModel(ModelPath, TrainingPath)
```

```
net.setInputMean((127.5, 127.5, 127.5)) # Set the mean values for the input image data
net.setInputSize(220, 220) # Set the input size for the model to 220x220 pixels
net.setInputScale(1.0 / 127.5) # Set the scale factor for the input image data to 127.5
net.setInputSwapRB(True) # Set the flag to swap the Red and Blue channels in the input data to correct the color
channel

# Start an infinite loop to continuously process frames from the webcam
while True:
  # Read a frame from the webcam
  success, image = camera.read()

  # Detect objects in the input image using the detection model
  ObjectIds, confidence, boundingbox = net.detect(image, confThreshold=threshold)

  # Print the detected Object IDs and bounding boxes to the console to highlight the object
  print(ObjectIds, boundingbox)

  # If objects are detected in the frame
  if len(ObjectIds) != 0:
    # Loop through all detected objects
    for ObjectId, confidence, box in zip(ObjectIds.flatten(), confidence.flatten(), boundingbox):
      # Draw a rectangle encircling the detected object on the image, color set to green and thickness set to 2
      cv2.rectangle(image, box, color=(0, 255, 0), thickness=2)
      # Put the object name of the detected object above the rectangle, position of the object ID can be modified.
      cv2.putText(image, EmptyClass[ObjectId - 1].upper(), (box[0] + 5, box[1] + 35),
cv2.FONT_HERSHEY_PLAIN, 1.25, (0, 255, 0), 2)
      # Put the confidence score of the detection above the rectangle, position of the confidence score can be
modified.
      cv2.putText(image, str(round(confidence * 125, 2)), (box[0] + 180, box[1] + 35),
            cv2.FONT_HERSHEY_PLAIN, 0.5, (0, 255, 0), 2)

  # Show the image with the detections in a window titled "Output"
  cv2.imshow("Output", image)

  # Wait for a key press (1 millisecond delay)
  # If a key is pressed, the loop continues; otherwise, it exits
  cv2.waitKey(1)
```

[26],[27],[28]

Appendix B

Project Schedule

January 6th, 2023:

- Discussed the topics that Dr. Muscedere talked about in the first capstone lecture.
- Looked over previous years' capstone projects to get an understanding of what the project entailed.
- Brainstormed potential capstone project ideas.
- Went over potential professors who could advise the project.

January 7th, 2023:

- Started to narrow down an idea for a potential report.
- Narrowed down which professors we could advise on the project.

January 13th, 2023:

- Review of advisors listed from previous meeting.
- Discussion of past projects of which the potential advisors have taken part in, and which best suits the current design topic.
- Creation of email proposal to potential advisor.

January 17th, 2023:

- Meeting with Dr. Muscedere warranted a meeting regarding the existing idea and how it can be improved.
- New information as presented from the Mechatronics lab on potential projects of which they are looking for engineering students.
- Emailing of professor who is overseeing the mechatronics lab capstone projects.

January 24th, 2023:

- Discussion of how to share the load for letter of intent.
- Review of information from Dr. Saeed about the project
- Review of Letter of intent syllabus
- Discussion of weekly schedule for capstone design project in order to maintain progress with the foresight of course schedule and course load.

February 1st, 2023:

- Review of edits made by Dr. Saeed within the letter of intent.
- Changing of scope within the Letter of Intent to align with the changes made by Dr. Saeed

February 17th, 2023:

- Review of topics discussed with Dr. Saeed
  - Creation of google collab as suggested by Dr. Saeed
- Review of TensorFlow documentation

February 27th, 2023:

- Creation of CNN trained to depict 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck.
- Adjusting of parameters to note how different parameters within the model affect the total accuracy of the model.

March 12th, 2023:

- Set up a meeting with the other groups and create a contact sheet.
- Continue to work on the AWS vehicle.
- Continue to do research about GNSS and decided which model is best to implement on the AWS vehicle.
- Set up the Raspberry Pi
- Implement database onto the CNN.
- Did research on mapping, GNSS, RNN, and CNN.
- Emailed advisor and evaluators to schedule a meeting for the progress presentation.
- Continue working on the progress report.

March 29th, 2023:

- Continue to research CNN.
- Continue to research GNSS.
- Continue to research RNN.
- Continue to research mapping methods.
- Continue finalizing the progress presentation.

March 30th, 2023:

- Rehearsed the presentation beforehand.
- Presented the progress presentation in front of our evaluators.

May 2nd, 2023:

- Met with our advisor who gave us a few ideas to implement to our project.
- Discussed our ideas and the next steps we should take for our project.
- Another robot was obtained to work with.

May 3rd, 2023:

- Began working on the new robot.
- Assembled the robot and installed a camera on the Raspberry Pi.
- Downloaded all the required software but could not proceed because we were missing an SD card.

May 8th, 2023:

- Begin the installation process for the robot.
- Downloaded and set up virtual box for the machine.

May 10th, 2023:

- Downloaded Ubuntu and began installing an ROS on the virtual machine.
- Modified the address of the virtual machine to configure the ROS software with the local IP address.

May 15th, 2023:

- Prepared the microSD card and reader.
- Downloaded the SBC image and burned the file.
- Downloaded the Raspberry Pi imager and wrote the image onto the microSD card.
- Restored the disk image and resized the partition of the microSD card in order to increase its capacity.
- Configured the Wi-Fi network settings by editing the directory in the microSD card. Replaced the WiFi_SSID and the WiFi_Password with the local WiFi_SSID and password.
- Booted the Raspberry Pi and began to configure the ROS network.

May 19th, 2023:

- Completed the OpenCR setup by installing the required packages onto the Raspberry Pi.
- Tested to see if the setup was successful by pressing the two switches on the OpenCR board. Switch 1 moved the robot 30cm forward and switch 2 rotated the robot 180 degrees.

May 24th, 2023:

- Began running the roscore from PC.
- Ran into some issues when trying to connect the PC to the Raspberry Pi with the SSH command line.
- Began looking at tasks that can be done in parallel.

June 5th, 2023:

- Fixed the issue with connecting the PC to the Raspberry Pi by changing the IP address used in the command window.
- Ran into issues connecting to the wifi at school as Ubuntu did not let us connect.
- Began researching mapping, specifically SLAM and how to create a map for the robot to read.

June 21st, 2023:

- Met with the sensors group along with the project advisor and technical advisor.
- Another meeting has been scheduled later to discuss if any problems arise.

June 23rd, 2023:

- Fixed the Wi-Fi issue by using a hotspot instead and finished the setup for the robot.
- Began working on the mapping.

July 3rd, 2023:

- Met with the sensors group to discuss the progress each group has made.
- Worked on the mapping portion of the robot. Created a "course" for the robot to drive around so it could generate a map.
- Successfully guided the robot from beginning of the course to end autonomously.

July 5th, 2023:

- Continue running trials of getting the robot to drive autonomously by creating a new map.
- Programmed drivers to get the camera operating.

July 8th, 2023:

- Researched and working on traffic sign detection programs.
- Researched on lane configuration and detection.
- Installed packages for both intrinsic and extrinsic camera calibration and viewed the camera through an image viewer.
- Calibrated the intrinsic camera using a printed checkerboard pattern where certain variables needed to be reached in order to fully calibrate.

July 11th, 2023:

- Began working on code for lane detection and lane configuration.

- Continued working with RViz (the mapping software) in determining which situations the mapping worked well and to calibrate the parameters within configuration files.

July 15th, 2023:

- Worked on the lane detection program to obtain results of images with lanes.
- Worked with Gazebo to create an extrinsic map for the machine to create a virtual map within RViz.

July 19th, 2023:

- Worked on the final presentation poster.

July 20th, 2023:

- Worked on the final presentation poster.

July 21st, 2023:

- Worked on the final presentation poster.

July 22nd, 2023:

- Continued working on code for the object detection algorithms to see if a working real time simulation can be produced.
- Continued working on lane and line detection algorithms to see is it can be shown from a live video.
- Continued working on the mapping with Gazebo and RViz to get a better map.

July 24th, 2023:

- Adjusting the final poster design.
- Continued working on code for the object detection algorithms. Obtained results for static image inputs. Began working on a real time model.
- Continued working on lane and line detection algorithms. Obtained results for static images of roads that detected lines and lanes. Began working on a real time model of a driving car to detect its lanes.

- Continued working on the mapping with Gazebo and RViz to get the robot to drive autonomously.

July 25th, 2023:

- Submitted the final poster design for printing.
- Continued working on code for the object detection algorithms and obtained results for real time object detection using a camera.
- Continued working on lane and line detection algorithms and obtained results for real time lane detection.
- Continued working on the mapping with Gazebo and RViz which outputted a map that allowed the robot to drive autonomously.
- Began working on a video of the results to be shown on presentation day.

July 26th, 2023:

- Obtained the physical poster and handed it in.
- Continued working on the video of the results.
- Began rehearsing the presentation.

July 27th, 2023:

- Finalized the video of the results.
- Continued rehearsing for the final presentation.

July 28th, 2023:

- Presented the final project.

July 29th, 2023:

- Began working on the final report. Discussed each section and assigned the work accordingly.

July 30th, 2023:

- Continued working on the final report.

July 31st, 2023:

- Continued working on the final report.

August 1st, 2023:

- Continued working on the final report.

August 2nd, 2023:

- Continued working on the final report.