Autonomous Driving Robot

CS39440 Major Project Report

Author: Josh Wells (jow102@aber.ac.uk)

Supervisor: Patricia Shaw (phs@aber.ac.uk)

6th May 2022

Version 3.0 (Release)

This report is submitted as partial fulfilment of a BSc degree in
Artificial Intelligence & Robotics (GH76)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work, I understand and agree to abide by the University's regulations governing these issues.

Name ……Josh Wells……

Date ……06.05.2022……

Consent to share this work

By including my name below, I hereby agree to this project's report and technical work being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name ……Josh Wells……

Date ……06.05.2022……

Acknowledgements

I am grateful to…

- Ben Weatherley (bew46@aber.ac.uk)
- Patricia Shaw (phs@aber.ac.uk)

Abstract

Increasingly cars are becoming more autonomous, giving warnings for speed limit signs and controlling the speed of the vehicle. Meanwhile, fully autonomous cars are being developed and tested on the roads in a variety of situations. There are many problems surrounding autonomous cars, and this project will start to explore some of those complexities.

The goal of this project was developing a Robot Operating System (ROS) based program to work with Turtlebot3, a relatively simple robot, inside a simulated environment (Gazebo) but with the potential for it to be transferred to a physical robot. The project aimed to tackle the issues surrounding lane detection, and the adjustment of the robot's driving behavior to accurately follow the detected lane.

The main body of this project involved developing a computer vision technique using OpenCV, to achieve an accurate lane detection algorithm that is also fast enough to be ran on a live video feed with minimal delay. The end-goal of the project was the development of a robot that can autonomously navigate a simulated road environment while obeying UK traffic laws.

Contents

1. Background, Analysis & Process

1.1. Background

The goal of this project was developing a Robot Operating System (ROS) based program to work with Turtlebot3, a relatively simple robot, inside a simulated environment (Gazebo) but with the potential for it to be transferred to a physical robot. The project aimed to tackle the issues surrounding lane detection, and the adjustment of the robot's driving behavior to accurately follow the detected lane.

The main inspiration for undertaking this project is the autonomous driving Tesla vehicles. Cars such as the Model S have the ability to self-drive, while it is not recommended that drivers of Tesla's relax completely it is possible for the driver to take their hands off the wheel and feet off the pedals completely. It is my belief that self-driving electric vehicles using computer vision techniques will be the future of transportation.

The main body of this project involved developing a computer vision technique using OpenCV, to achieve an accurate lane detection algorithm that is also fast enough to be ran on a live video feed with minimal delay. The end-goal of the project was the development of a robot that can autonomously navigate a simulated road environment while obeying UK traffic laws.

I did not have any previous practical experience with implementing computer vision techniques, but I have an interest in computer vision and its applications. I very much wanted to learn how to successfully implement the techniques I had studied into a functioning program. The coupling of computer vision and robotics is what motivated me to select this autonomous driving robot as my project.

In preparation for this project, I taught myself methods for implementing OpenCV to work alongside ROS on a live, simulated robot inside a simulated environment. Mainly using a YouTube tutorial about the basics of OpenCV ***(Murtaza's Workshop – Robotics and AI, 2020)***. I have plenty of practical experience programming in C++, so decided to continue the project using C++ over Python. I began by researching implementation techniques for OpenCV in C++, and created a few simple practice programs to help me get used to the OpenCV library.

In order to get OpenCV to work alongside ROS I had to make use of the CV_BRIDGE ***(wiki.ros.org, 2017)*** library available for C++. This required some research and a small bit of practice to get the live raw image from the robot into a format which I could then use for image processing with OpenCV.

I assessed a number of different existing systems, including a paper entitled 'Automatic Driving on Ill-Defined Roads: An Adaptive, Shape-constrained, Colour-based Method' ***(Ososinski and Labrosse, 2013)***. This paper investigated the creation of a system that can identify the size and shape of the road ahead of it, before navigating along the road. The method discussed in this paper was effective, but complicated. Designed for use on ill-defined roads (e.g. country roads or lanes), this method adds multiple layers of complexity that my project did not require. As my project would take place inside a simulated, well-marked road environment there would be no need for the extra steps taken by this method to identify the drivable area in front of the robot.

Another method which I investigated made use of a series of filters, that are available within the OpenCV library, that would be able to detect the lines on either side of the road ***(Pysource, 2018)***. This method works well when there are clear markings on the road, once the edges of the lane/road are detected I would then be able to successfully calculate the robots movement path along the center of the lane.

Before beginning any work on the project itself, I did some research into the robots that were available for me to use inside my simulation. The project would require a robot with a front facing rgb camera and the ability to drive. The Turtlebot3 was the most obvious choice, has it would have everything I need to complete the project, and did not have any excess features that would only end up getting in the way.

1.2. Analysis

The problem involves being able to successfully detect the edges of a lane/road, from the background work I can conclude that there are many different ways that one can achieve this goal. The approach taken by Marek Ososinski and Frédéric Labrosse ***(2013)*** provides a robust solution to detecting the width of the road ahead of the robot. By making use of a shape-constrained, colour-based method they were able to determine the exact edges of the road and draw a trapezoidal shape based on the shape of the detected road edges. This shape would then be able to determine the drivable area in front of the robot and use this information to steer, keeping the robot in the center of the road at all times. While this approach is extremely robust and would provide an accurate lane following behavior for this project, it adds a number of layers of complexity that I felt would not be achievable in the time-span of this project.

In light of this, I felt a better method for this project would be to attempt to identify the marked lines on either side of the road before then calculating the centre of the lane for the robot to drive along. This approach, while not as robust as the previously mentioned one, seemed far more achievable in the time-span of this project.

The main tasks of the project were as follows:

1. Create a simulated road environment for the robot to drive in.
    - Develop road model.
    - Create launch files to open world, and spawn robot at same time.
2. Develop an algorithm that can successfully detect the markings on the road.
    - Investigate existing systems for inspiration.
3. Develop an algorithm to calculate where the edges of the lane are.
4. Calculate the centre of the detected lane.
5. Develop a lane following behaviour for the robot that will drive along the centre of the lane determined by the previous algorithm.

I arrived at this list of objectives based on what I believe is achievable in the time available for this project.

1.3. Process

The life cycle model that I used to develop my project was based on an Agile Scrum approach. I chose to use an Agile Scrum approach as its designed to be used for developing software based projects. Agile Scrum is very adaptable and uses an incremental development process consisting of sprints to complete specific pieces of work within certain times. This allowed me to concentrate entirely on completing one specific task, during each sprint I would not work on any other area of the project. This kept me focused on the task at hand. Agile Scrum sprints aim to develop the most important features first, and then once they are complete (or near complete when the sprint ends) you go back and perform another sprint to complete the smaller features.

I adapted this for my needs, and used the iterative/incremental approach of Agile Scrum and the concentrated sprints to complete large sections of code for the most important features such as my image processing function and the algorithm for selecting the most appropriate Hough Lines. These were two of the largest and most important features of the project, and so I concentrated on completing these features first using sprints before going back to complete the smaller features, such as the driving function.

2. Design

2.1. Overall Architecture

The project has 3 major areas: image processing, selection of Hough Lines and the calculation of the centre of the lane and driving. The project makes use of a subscriber/publisher system using ROS. ROS acts as a middle man between the robot and the program, the program will subscribe and publish to ROS topics. These topics include the raw image from the robot, as well as the topic that controls the robots driving.

The image processing function takes the image from the robot and finds the edges of the lane using the white road markings. The detected road markings can then be used to decide the size of the lane. Using a selection algorithm, the program can decide the most appropriate points to use for lane detection.

The selection of the most appropriate Hough Lines/points can then be used to draw lines on the image to represent the edges of the lane. From these points it's possible to calculate the centre of the lane detected ahead of the robot. Once the centre of the lane is calculated, a line can be drawn on the image to represent the path which the robot will be required to follow. Using the position of the this centre line on the image, the robot can decide if it needs to turn left/right or continue straight.

The flow diagram *(Fig1)* for this project is relatively simple, the robot should remain in a loop until it either reaches the end of the road or the program is manually shutdown.

Within this loop, the program should retrieve the image from the ROS topic, before passing the image through the image processing function. The image processing has multiple steps of its own that is not shown in this simplified diagram. After image processing is complete, the program will then select the most appropriate Hough Lines before calculating the centre of the lane ahead of the robot. From this information the program can decide where to steer the robot.

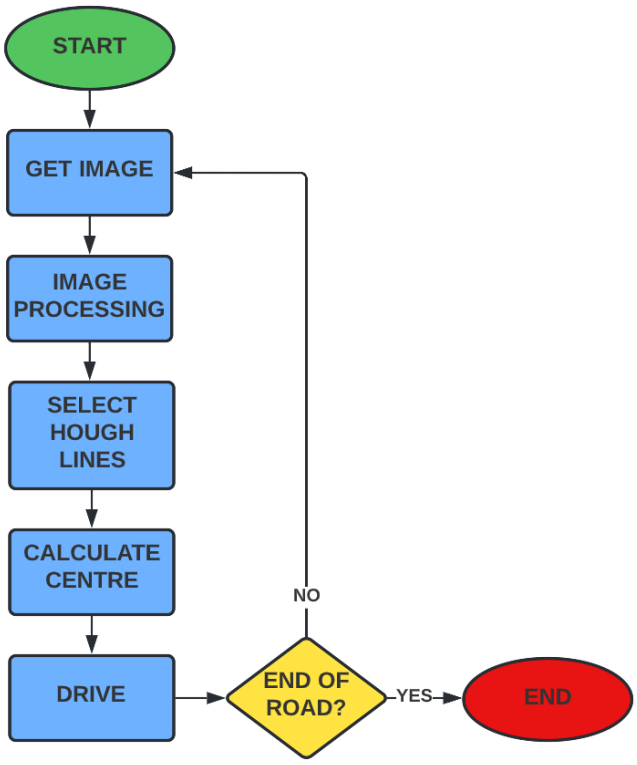

Fig1 (above): Project flow diagram.

2.2. Detailed Design

2.2.1. Support & Implementation Tools

The programming language selected for this project is C++; I chose this language as it provides an in-depth library that I feel gives me more control over the project that I do not get from other programming languages such as Python. Python in general is a fantastic choice for creating ROS programs, however I have far more practical experience working with C++ compared to Python. Due to this, C++ was the obvious choice for this project.

I made use of the CLion IDE to write the code, which gave me some basic debugging tools which helped to spot some of the more obvious errors before attempting to compile. I made use of Catkin to compile the ROS programs quickly, this would also provide me with fairly detailed error messages if any errors were found in the program which made turnaround times much faster when implementing new code.

This is a ROS based project, designed to run on a live robot using a subscriber/publisher system. ROS allows me to subscribe and publish to specific topics that then control the robot. The camera from the robot will publish its image to the 'camera/rgb/image_raw' topic, my program then subscribes to that topic where it can then get a message containing the raw image direct form the robot. Using the cv_bridge library, I am able to convert that message into an image that can then be used for image processing with OpenCV.

The robot is simulated within Gazebo, this allows me to do repeated tests with the program without needing a physical, real-world robot. Gazebo allowed me to create a test environment that mimicked a real-world road. Due to safety and other limitations with the project, a real-world robot operating on a real road was not possible. The environment model for the road was provided to me by Ben Weatherley, a student who completed a similar project last year entitled 'Autonomous Driving Robot – Detecting and Reacting to Roadsigns' ***(Weatherley, B., 2021)***. The road system modelled was that of a short loop, with a number of sharp 90 degree turns, 2 bus stops and 2 give-way junctions.

The project will make use of Github

2.2.2. Image Processing

The image processing is done in 5 steps; Cropping, Convert to HSV, Colour Mask, Edge Detection and finally Hough Line Generation. This process will result in a new edited image that has the edges of the lane marked and saved. This new image can then be used by the program for the later stages of the lane following behaviour.

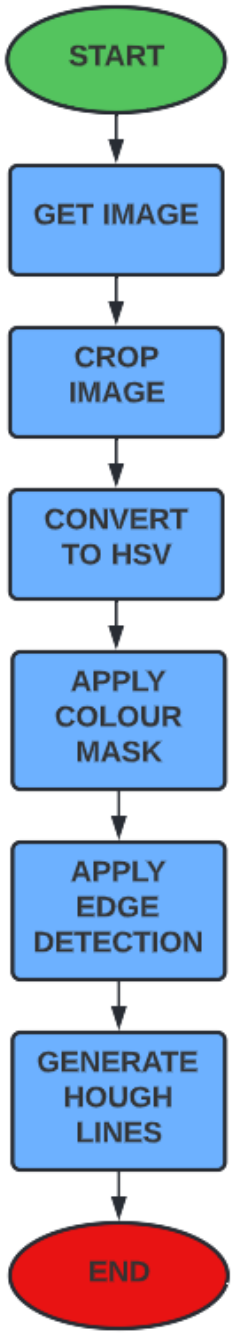

Fig2 (above): Image Processing Flow Diagram

The program will first retrieve the image from the ROS topic and convert it a useable OpenCV image ready for processing. This process will involve the use of the CV_BRIDGE library, as the ROS topic that stores the raw image from the robot will only send that image to the program as a message which will need to be converted inside a subscriber call-back function.

The first step in the image processing technique for lane detection is to crop the image. The raw image from the robot will contain a lot of information that is not useful, and can actually cause extra issues if not removed. The upper half of the image should be cropped out to reduce processing time and complexity.

The image from the robot is in the Blue, Green, Red (BGR) colour space. This colour space provides a natural looking image and colours however it is not particularly easy to mask. To get around this, the image will be converted into the Hue, Saturation, Value (HSV) colour space. The HSV colour space is much simpler to mask specific colours, which will make the next step of the process much easier.

The image will then have a colour mask applied to it so that only areas that match the set range of HSV values will be seen on the image. In the case of my project the algorithm will mask off any area that is not white, masking everything but the road markings. Masking the unneeded areas from the image allows the Canny Edge Detection in the next step to be more accurate.

The image will then be ran through a Canny Edge Detection filter, which will create a number of points along the detected edges on the screen. The Canny Edge Detector provides much smoother edges than those produced by the Sobel Edge Detector at the expense of slightly more processing time.

The final step for the image processing will use the Hough Lines filter to generate lines on the image based off the points generated by the edge detector. These lines will each have their start and end coordinates stored in a vector, which will allow the next step of the program to sort the most appropriate points to use for lane tracking.

2.2.3. Selection of Appropriate Hough Lines

The generated Hough Lines are stored inside a vector, each line has 2 points (start and end) and each point has 2 values (X and Y coordinates). Selecting the most appropriate lines to use as markers for the edge of the lane requires an algorithm that can systematically go through each of the stored lines and decide if the position of the current line is better than the previously checked line.

Fig3 depicts an example image from the robot. The black lines on the image are the detected Hough Lines, representing the road markings visible to the robot. Each of these lines will have start and end XY coordinates that will need to be checked in order to find the best possible points for tracking the edges of the lane.

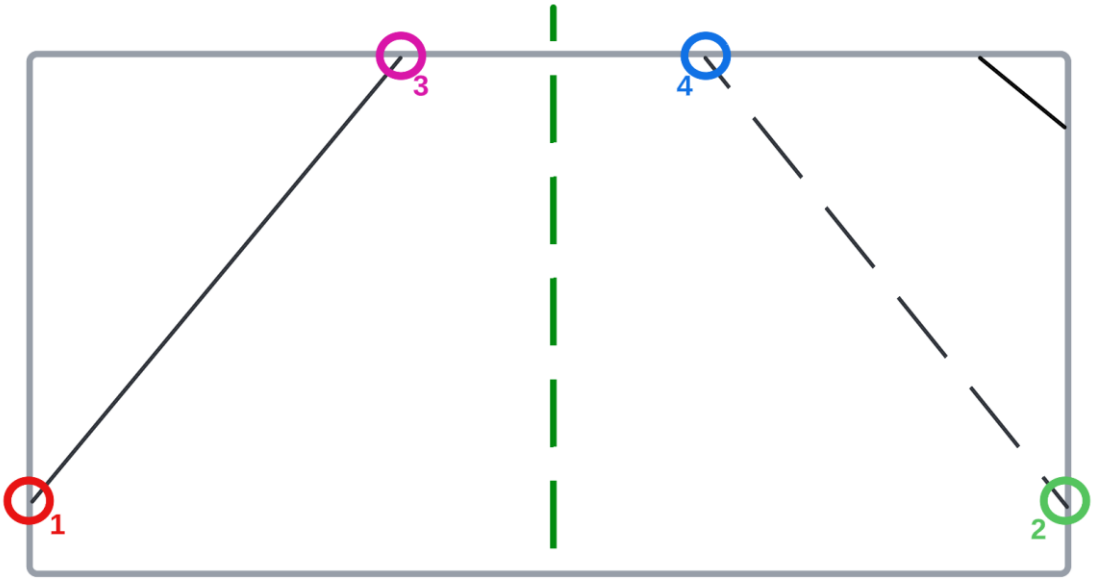

Fig3 (above): Example of Hough Lines selection.

The algorithm will select the following:

- The lowest point on the left side of the image: *(Fig3, CIRCLE1 (RED))*
- The lowest point on the right side of the image: *(Fig3, CIRCLE2 (GREEN))*
- The highest point on the left side of the image: *(Fig3, CIRCLE3 (MAGENTA))*
- The highest point on the right side of the image: *(Fig3, CIRCLE4 (BLUE))*

Once the algorithm has selected the 4 most appropriate points, the program will draw a left and right line on the image between the relevant points, that will represent the edges of the lane.

Fig4 shows the result of the Hough Line selection process. The drawn lines represent the edges of the detected lane *(Fig4, LINE1 (BLUE), LINE2 (RED))*.

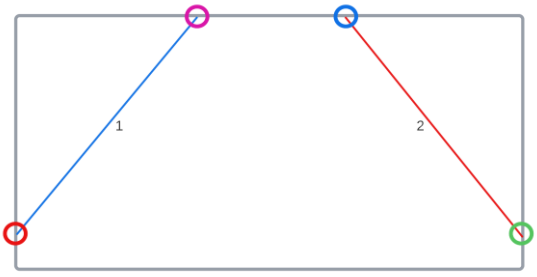

Fig4 (above): Example of lane edge detection.

2.2.4. Calculating Centre of Detected Lane & Driving

The centre of the lane is calculating using the results from the Hough Lines selection. The lines generated on the left and right side of the image are each made from the 2 relevant points selected by the algorithm. To draw the line in the centre of the lane, the program will calculate the centre of the top 2 points, as well as the centre of the bottom 2 points. This gives 2 new points that the program can then draw a line between, representing the calculated centre of the lane.

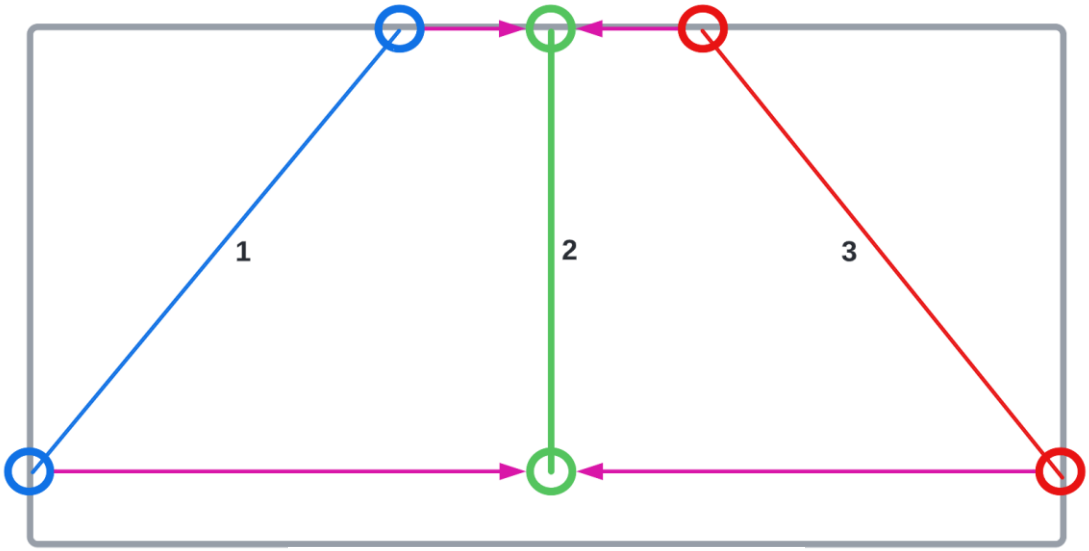

Fig5 (above): Example of centre line calculation.

Fig5 shows an example of the steps taken to calculate the centre of the lane *(Fig5, LINE2 (GREEN))*. The left and right lines are drawn from the most appropriate Hough Line points selected in the previous step *(Fig5, LINE1 (BLUE), LINE3 (RED))*. The arrows drawn between the points are shown for the purpose of this diagram, and are not actually drawn on the image from the robot. The centre points calculated by the algorithm are depicted by the circles at either end of the centre line *(Fig5, LINE2 (GREEN))*.

Each time the call back function updates the image from the robot, the image processing will repeat each of the steps mentioned in the above sections. When the centre line has been calculated, the driving function will make a decision based on the position of the line on the image.

An algorithm will calculate how far the centre green line is from the middle of the image, and if it is to the left or the right. From this, the program will decide if the robot needs to turn left/right, or continue straight. As the detected lane in front of the robot turns around corners, the centre line will skew to the right or left allowing the robot to figure out how much it needs to turn to remain in the centre of the lane. The program should always aim to have the centre of the lane in the centre of the image.

3. Implementation

3.1. Image Processing

3.1.1. Failed Design

Before deciding to go with the above design, I explored a different method of tracking the lines on the road. This method made use of 2 image perspective warps, one for each side of the lane. This warp would make each side of the image, where the lines were expected to be, appear as if from a birds-eye view inside 2 separate cropped images. In theory, the line would appear in the centre of the warped image as a straight line from top to bottom. This process was meant to make the detection and tracking of the line easier, as the program would be able to drive based off the position of the line on the image. Due to the 2 lines being tracked separately from each other, it would be possible for the robot to drive relatively accurately when just one of the lines is in view/being tracked.

However, this design fell apart rather quickly. The warped image technique relied on the lines being in the exact same place every time and could not account for varying lane sizes. Other issues with this design included the program losing track of the line extremely easily, even the slightest of turns by the robot would cause the warped image to become almost entirely useless. When the robot would approach a corner, it would lose track of the line completely. I attempted to correct some of these issues by increasing the area of the image that gets warped, hoping that the larger size of the warped image would allow the line to remain visible in small turns. This fixed the problem for driving in a straight lane, but sharp corners remained un-driveable. Road markings such as the bus lanes/side of road parking and narrower sections of road would also cause this design to fail. After a week of trying to make this design function, I decided to abandon it and go back to the drawing board. This failed design made me realise that the problem of lane detection is much more complex that I originally assumed.

3.1.2. Design Challenges

While implementing the successful design, I encountered a number of challenges. I overcame each of these challenges and produced a successful image processing technique for lane detection. Overall the implementation of the image processing was difficult, and I learned a lot about various filters and techniques while creating the program.

One of the first issues I encountered was with the application of a colour mask onto the image. The raw image from the robot is in the BGR colour space, which provides a natural coloured image that is great for viewing. However, BGR is not easy to apply a colour filter on, as it requires a more complicated set of filters. To get around this problem, I converted the image from BGR into HSV colour space. This colour space isn't great for viewing, but the application of a colour mask is far simpler. To apply a mask to an HSV image, you only have to set an upper, and lower limit for the Hue, Saturation and Value. This can be done using 2 Scalar types, then inserting those Scalars into the OpenCV library function 'inRange(imageIn, lower, upper, imageOut)'. The only parts of the image that are visible in the new image are the areas that had an HSV value between the upper and lower limits set in the function. For my program, everything but the white road markings would be masked off.

The 2nd challenge I encountered was tuning the parameters of the Canny Edge Detector ***(docs.opencv.org, n.d.)*** and Hough Lines functions ***(docs.opencv.org, n.d.)***. Running the program, noting the results of the functions and then adjusting the parameters before re-running the program was slow and hard to keep track of. I got around this by making use of Track Bars inside another image window. These Track Bars allowed me to assign a slider to a variable, which could be adjusted while the program is running. Doing this allowed me to run the program once, and adjust all of the parameters using the sliders and tune the functions in one go while also being able to see the live results of me changing the variables in the image windows output by the image processing function.

The Hough Lines required a lot more tuning than the Canny Edge Detector did. I repeatedly had to go back and adjust its parameters throughout the development of the project as I tested the program in various situations. I struggled to find the correct balance for the parameters, ensuring that it was sensitive enough to detect the smaller lines in the middle of the road, but not too sensitive that it mistook small errors in the edge detector as new lines. The settings that the Hough Lines function is currently using are good enough for the rough detection of the edges of the lane.

The image processing function was also far more intensive than I originally expected. My plan was to have the image processing done inside the call-back function for retrieving the raw image from the robot. However, this plan failed as soon as I tried to introduce a publisher object for the driving control. When ROS calls to update its call-back (subscriber) functions, they block other functions from being able to communicate with ROS. This meant that as the intensive and slow image processing was being completed, the driving function wouldn't work. The image processing would also be required to loop as fast as possible so that the lane detection is up to date at all times, meaning as soon as the image processing finished it would instantly loop again which gave no time for the driving function to operate.

To get around this issue, I moved the image processing into its own function which would be called in turn. This meant that the driving function should always get a chance to publish to ROS between loops of the image processing. This solution worked for a short time, but as the image processing got more complex it eventually became obvious that a single thread system was not going to work. I then moved the image processing into its own separate thread *(See*

Appendix B), and created some global variables to allow cross-thread communication. This solution presented some small issues at first to do with OpenCV exceptions but these issues were quickly resolved by adding in some artificial slow-downs into the loop as well as giving the program some buffer time when it first opens (Waiting around 3 seconds for the program to properly start before attempting any image processing).

3.1.3. Implementation of Image Processing

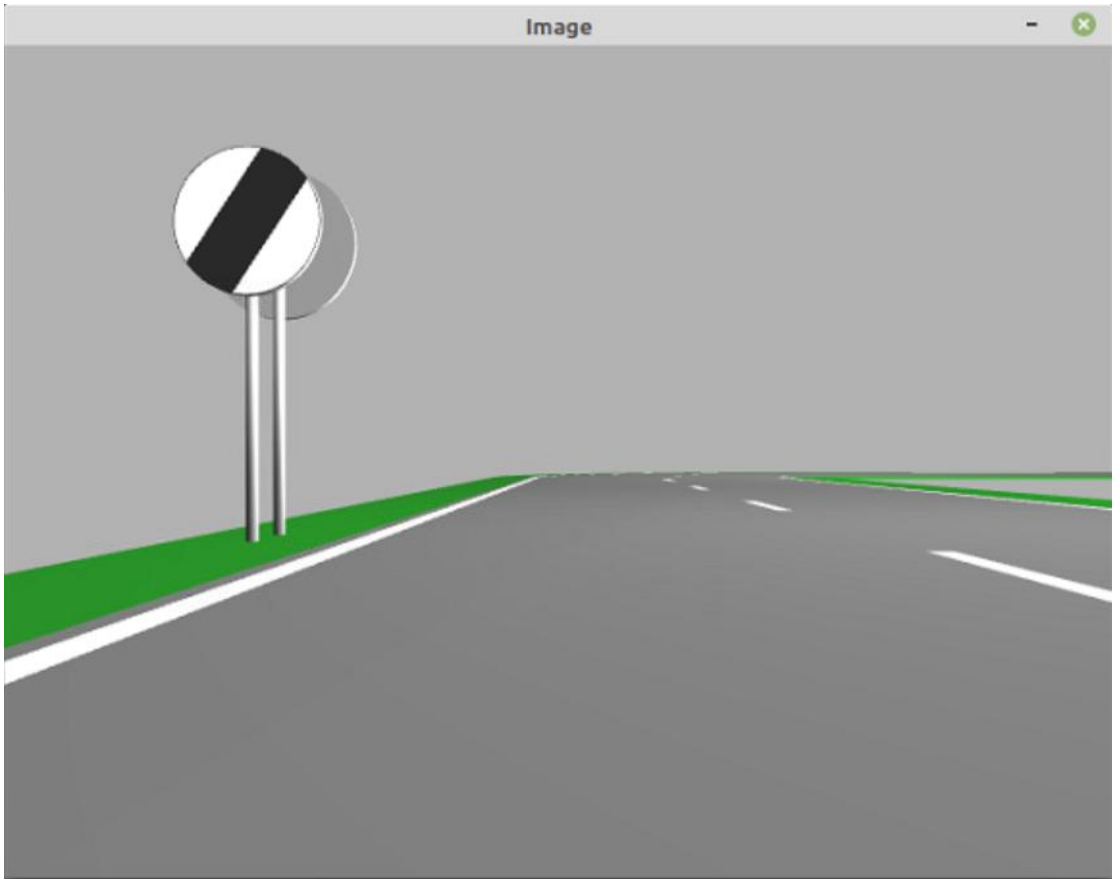

Fig6 (above): Raw image from robot.

The first stage of the image processing involves the retrieval of the raw image from the ROS topic 'camera/rgb/image_raw'. The above image *(Fig6)* shows the raw image after being converted to a useful OpenCV image, ready for processing.

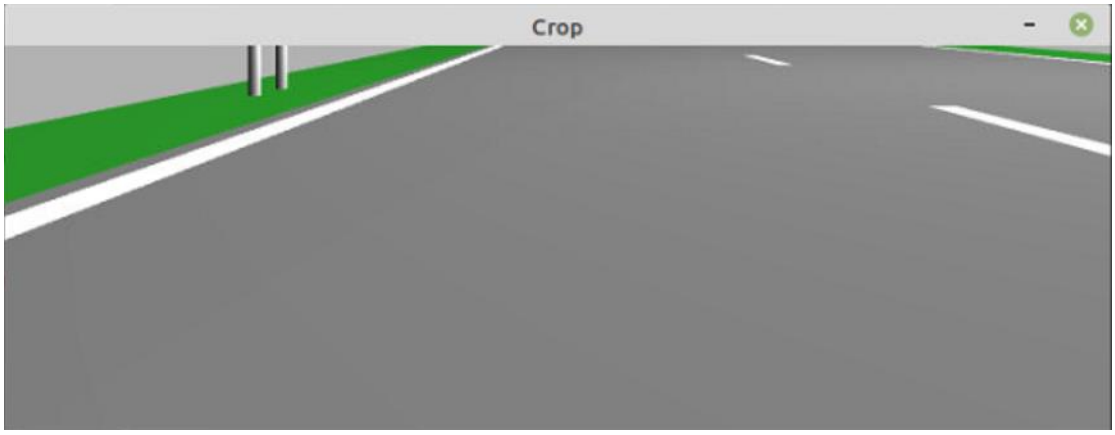

Fig7 (above): Image after being cropped.

The next step I implemented was the cropping of the image, this removes most of the unnecessary parts of the image and reduces image processing time in the next processing steps. The above image *(Fig7)* shows the results of the image being cropped.

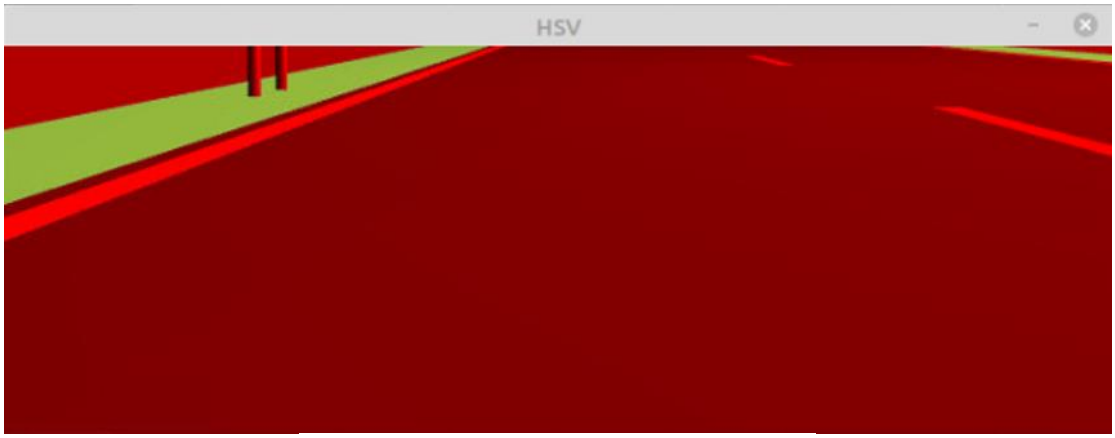

Fig8 (above): Image after conversion to HSV.

After the image is cropped, it then needs to be converted to HSV colour space in order to make the masking step simpler. The above image *(Fig8)* shows the results of the HSV colour conversion, as you can see the HSV image is not used for anything other than the application of the mask.

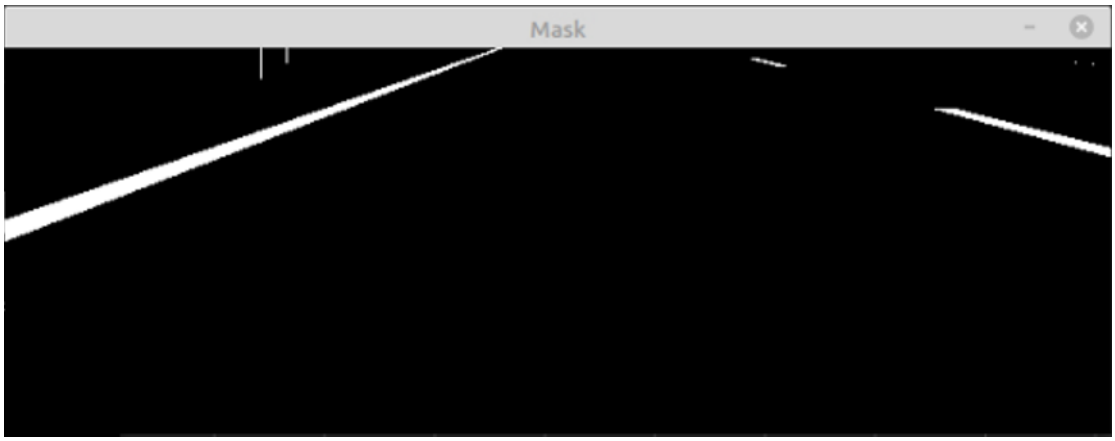

Fig9 (above): Image after being masked.

The image is masked so that only the white road markings are visible. This makes the detection of edges much simpler, and more accurate. Since I am only interested in tracking the lines on the road, masking off everything else reduces the chance for error in future steps. The above image *(Fig9)* shows the result of the image being masked.

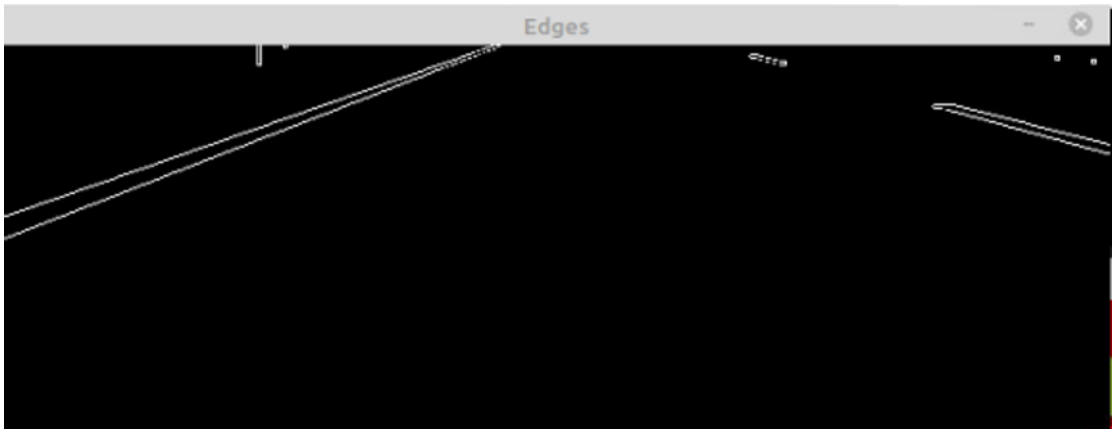

Fig10 (above): Canny Edge Detected image.

The image is fed through a Canny Edge Detector filter, which looks for edges on the image. Due to the image being masked in the previous step, the edges of the white lines are well defined against the pure black, masked background. The Canny Edge Detector creates a point cloud on the image, generating points along each of the edges. The above image *(Fig10)* shows the result of the Canny Edge Detector filter being applied to the image from the robot.

Fig11 (above): Generated Hough Lines.

After the image has been fed through the edge detector, it is then put through the Hough Lines P generator. This filter uses the point cloud generated by the Cany Edge Detector and attempts to draw straight lines along the detected edges. It then stores each of the generated lines using their start and end XY coordinates inside a vector. The above image *(Fig11)* shows the results of the Hough Lines P filter drawn over the original BGR colour space cropped image.

(See Appendix B for code)

3.2. Selection of Appropriate Hough Lines

3.2.1. Failed Designs

Developing the algorithm to select the most appropriate Hough Lines from the vector of lines created by the image processing function was one of the most challenging parts of this project. This algorithm was required to sort through all of the generated lines, and calculate which of the saved points within each of those lines was the most accurate in order to track the edges of the lane. The algorithm became extremely confusing, and required a number of iterations before it finally produced a useable and semi-reliable outcome.

The first method I implemented would sieve through the vector of stored lines, and choose the 2 lines (left & right) that best fit. This technique did not predict the edges of the lane correctly, and would often get confused by other markings on the road or select lines that were produced due to a small error caused by poor parameter tuning in the Hough Lines function during the image processing. This method would only take into account whole lines as generated by the Hough Lines function, these lines would often be incomplete or would not extend the whole length of the lane edge.

The second approach I attempted would try to find the 2 highest points on the image, and then use those for tracking. The lower 2 points were fixed on the image. This presented some issues with tracking accuracy, the calculated edges of the lane were often incorrect and the algorithm would sometimes choose overlapping points.

The third design used an approach similar to the one in the final program, but lacked some crucial checks, and a ‘deadzone’ to reduce sensitivity. Due to this, the third design would often pick overlapping points which would then confuse the driving function.

The algorithm for selecting the Hough Lines was written, and re-written many times. Due to the confusing design, many more designs were developed to tackle the issue. Most failed designs never made it past a few lines of simple pseudocode before they were tossed in the bin. This ordeal made me realise that the problems of lane tracking would take me longer than expected to solve.

3.2.2. Design Challenges

The algorithm became very complicated very quickly, and required many hours of testing different approaches before I landed on the current working method. The current algorithm is not perfect or very efficient, and has room for improvement. However, I ran out of time in this project and could not fix all of its problems. I wrote the algorithm's pseudocode out many times on a piece of paper as a way of visualising the logic. The Hough Lines would be generated from left to right, and so the algorithm would need to be able to identify if the line is on the left or right side of the image, as well as which direction it is facing based off of the start and end points stored for each line. This meant the algorithm would need to be able to flip the lines start and end points so that they would be drawn from the bottom to the top of the image, instead of left to right, which would make the selection of the most appropriate points simpler to figure out.

The current algorithm had issues when approaching sharp turns in the road, where it would lose track of one side of the lane entirely. This was a massive issue that required a number of redundancies to be programmed so that the robot may continue driving until such time that the program is able to re-acquire the line. The addition of 2 bool types, that would check if the program has managed to find both left points, and both right points helped to reduce the number of false readings being given to the driving function *(See Appendix B)*. If the program was not able to detect 2 points, it would set the coordinates to a default number that the program would never reach naturally. The program performs 2 checks on the coordinates, one for each side of the lane. If it was not able to draw the left or right lane then the relevant bool is set to false. The driving function can then see the false reading and decide how to continue instead of receiving the wrong information as previous designs gave it. This reduced the number of errors when driving and improved the overall accuracy of the robot's driving especially when going around sharp corners.

3.2.3. Implementation of Hough Line Selector

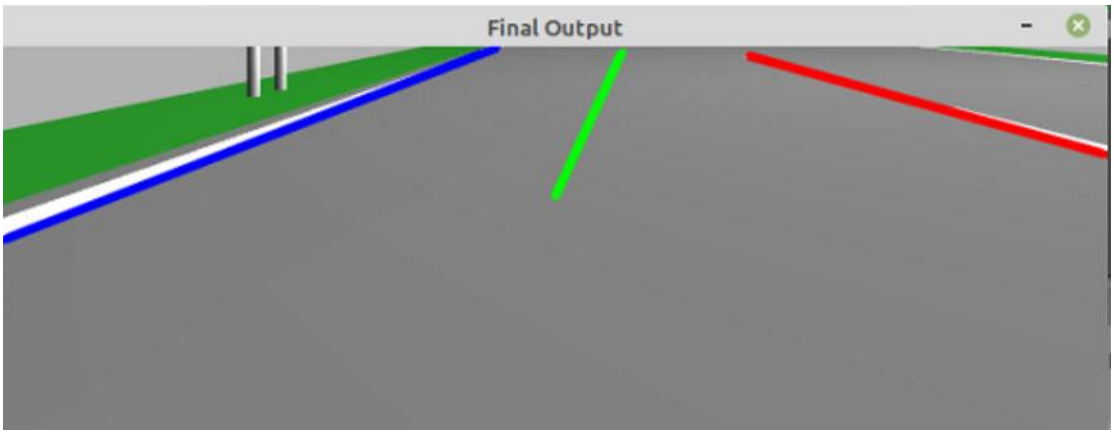

Fig12 (above): Final image after Hough Line selection.

The selection of the most appropriate Hough Lines made use of an algorithm that would pick out the 4 best points generated by the Hough Lines P filter and use them to calculate the edges of the lane ahead of the robot *(See Appendix B)*. The above image *(Fig12)* shows the final image output by the program after it has finished selecting the most appropriate Hough Lines, and calculated the centre of the lane. The robot will use this information to determine how to steer itself so it remains in the lane and turns corners.

The calculation of the centre of the lane as depicted *(Fig12)*, relies on both the left and the right lines being successfully detected. If one side fails to detect the edge of the lane, then the centre line is not drawn. In this scenario, the robot will begin a slow turn towards the side that cannot be detected in the hopes that it will be able to re-detect the line.

3.3. Driving

3.3.1. Design Challenges & Implementation

The implementation of the driving function was the most straight forward part of this project, however it was not without its own issues. The first implementation of the driving failed to function altogether. This was due to an unforeseen issue with the image processing which caused it to block communication with ROS. This was not the main issue with the design, however. The main issue was that the driving function (when working) would cause the robot to jitter left and right very quickly and not actually go anywhere.

This issue was caused by a lack of a 'deadzone' when calculating whether the robot should turn left or right, plus the driving/turning speeds had not been correctly set and were far too fast. If the centre line was not perfectly in the centre of the screen, on the exact centre pixel, the robot would try to turn in order to correct this. Which caused the robot to be constantly fighting to have the line in the exact centre, which is not possible.

This introduction of a deadzone to the driving function meant that the line had a larger zone to be in that the program would then consider to be centred. The next issue caused by this was trying to tune the size of the deadzone so that it was big enough so the robot could drive forward smoothly, but making sure it wasn't so big that the robot would drift too far out of the centre of the lane.

I also implemented the previously mentioned bool types that read true or false depending on if the relevant line was able to be drawn by the Hough Line selection process *(See Appendix B)*. For example, if the left line bool reads false then the program will apply a small left turn to the robot in an attempt to regain tracking of the left hand side line. The same process is used to regain tracking of the right hand side line if the program loses track of it. This driving function only changed a small amount from when it was first designed and implemented, it mostly operated without problem.

(The code for the implementation of the driving can be found in Appendix B)

3.4. Conclusion

In conclusion to the implementation section, I can compare the final project to the original requirements that are mentioned in the analysis section. The project has met each of the originally set requirements. It can successfully detect the left and right edges of the lane ahead of the robot, calculate the centre of the lane and then navigate the robot around the simulated road environment.

1. Create a simulated road environment for the robot to drive in.
    - Develop road model.
    - Create launch files to open world, and spawn robot at same time.
2. Develop an algorithm that can successfully detect the markings on the road.
    - Investigate existing systems for inspiration.
3. Develop an algorithm to calculate where the edges of the lane are.
4. Calculate the centre of the detected lane.
5. Develop a lane following behaviour for the robot that will drive along the centre of the lane determined by the previous algorithm.

The above requirements are what was set in the analysis phase of the project, and I believe the project has reached all of these targets successfully.

4. Testing

4.1. Overall Approach to Testing

I took an iterative testing approach when developing this project. Meaning that at each step of the project I would manually test that each feature was working as intended. I did this by repeatedly running the program throughout the project lifespan, ensuring that there was no major issues caused by the latest changes to the program.

I took this approach due to the nature of the project. Robotics projects can be difficult to test, especially when it comes to autonomous programs. It was hard to pinpoint exact testing requirements, as the autonomous nature of the program would mean that the result from each run of the program would vary depending on the starting position, the road ahead and the output of the image processing function.

This testing approach proved quite successful, as constant testing of the program each time a change is made helped to identify specific problems. For example, if I made a change to some of the logic and the next run of the program resulted in a completely wrong or unwanted result then it would be easier to figure out which section of the program was causing the issues. With it being more likely to be the code that had been changed since the last successful test. I believe this testing approach assisted in creating a more reliable program, as repeated iterative tests helped to ensure the results of the program were acceptable at each step of the development.

After the project had reached maturity, and the program was able to successfully detect and follow the lane ahead of the robot. I began a series of more specific tests. These specific tests were aimed to test the average accuracy of the lane following behaviour. I created a separate program to log the X & Y coordinates of the robot as it drove *(See Appendix B)*. The program achieved this by subscribing to the gazebo/model_states ROS topic, this topic stores the current positions and orientations of all models currently loaded inside the Gazebo world simulation.

To test the robots accuracy, I ran the logger program while manually driving the robot using the turtlebot3 teleop in a terminal. By doing this I was able to drive the robot along the lane as accurately as possible in order to gain a reference to how the autonomous program should drive. I then reset the robot to the exact same position I started driving from, restarted the logger program and then ran the autonomous lane following program for the same stretch of the road that I had driven manually. By extracting the position values of the robot twice per second on both logs I would not only be able to test how accurately the robot can stick to the centre of the lane, but also how quickly it was able to complete the section of road.

By taking the results of both logs, and placing them onto an overlapping XY scatter graph I would be able to see the average position of the robot compared to the lane centre when driving manually and driving autonomously.

4.2. Testing Results

Below *(Fig13)* shows an XY scatter graph of the positions of the robot from driving a section of the road both manually and autonomously in comparison to the centre of the lane. The section of road in question is straight, with the end of the road section approaching a sharp corner.

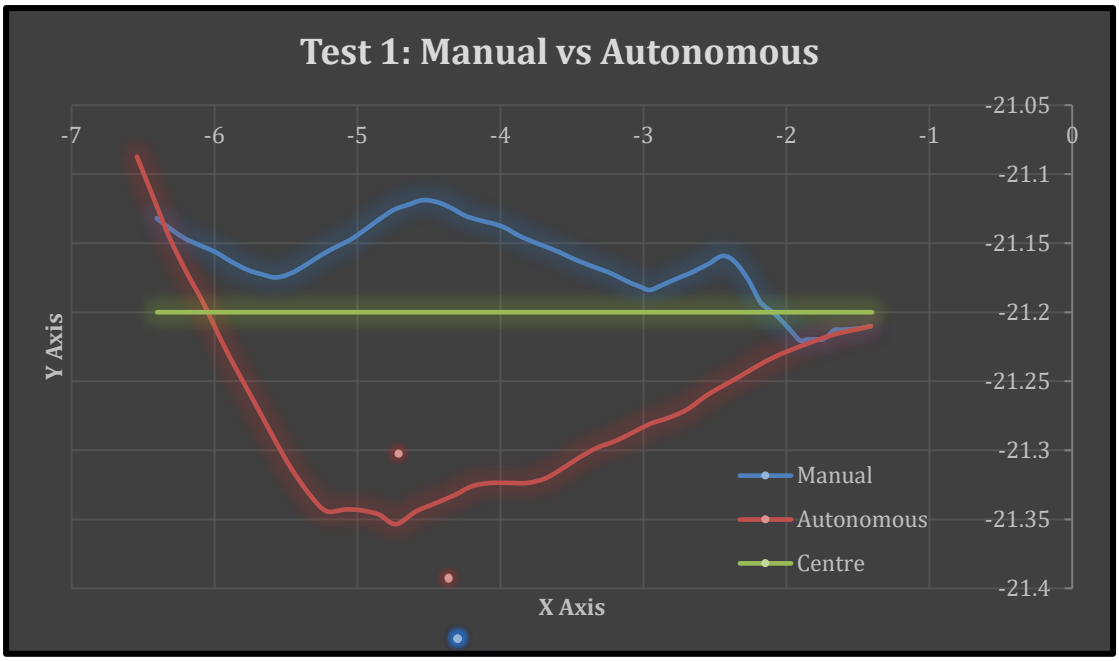

Fig13 (above): Testing results 1

The above scatter graph *(Fig13)* shows the logger data for 2 attempts at driving along the same short stretch of road. One attempt was driven manually, and the other driven by the autonomous lane following program developed during this project. The results of the test show that when driving autonomously, the program tends to favour sitting a bit further to the left of the lane. The program tended to drive closer to the left curb, as the calculation of the lane centre wasn't perfect, the robot detected the upcoming right turn in the road and began to adjust itself in preparation for turning this bend before the test was cut short. When compared to my manual driving, the robot is further from the centre of the lane, but was still within the acceptable parameters and did not cross the lines on either the left of right side of the lane which would be cause for a fail. Note that the test shown in the above graph *(Fig13)* starts from right and ends at the left.

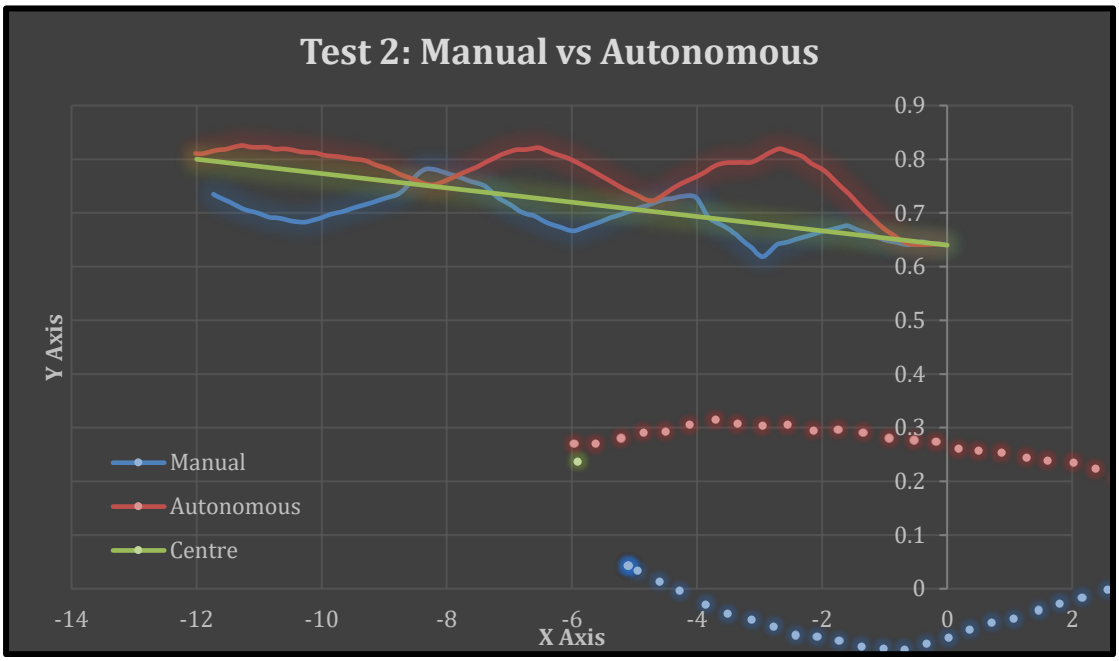

Fig14 (above): Testing results 2

This second test was performed on a different section of the road environment. This section is still straight, and does not feature any corners however it is a longer stretch of road. The longer section of road will give a better idea of the accuracy of the lane following over a longer period of time. As before, the graph shows the position log data from both manual driving and autonomous driving along the same stretch of road. Note that the data show in the above graph *(Fig 14)* starts at the left and ends on the right.

The autonomous lane following still prefers to sit closer to the left side of the lane, however on this longer stretch of road has veered in and out of the centre of the lane 3 times over the total distance travelled. As the robot veered further from the centre of the lane, it detected this and corrected itself back to the centre. However it then began to veer back to the left again. I believe this slight veering is caused by uneven motors, which is making the robot turn to the left slightly when it is supposed to be driving straight.

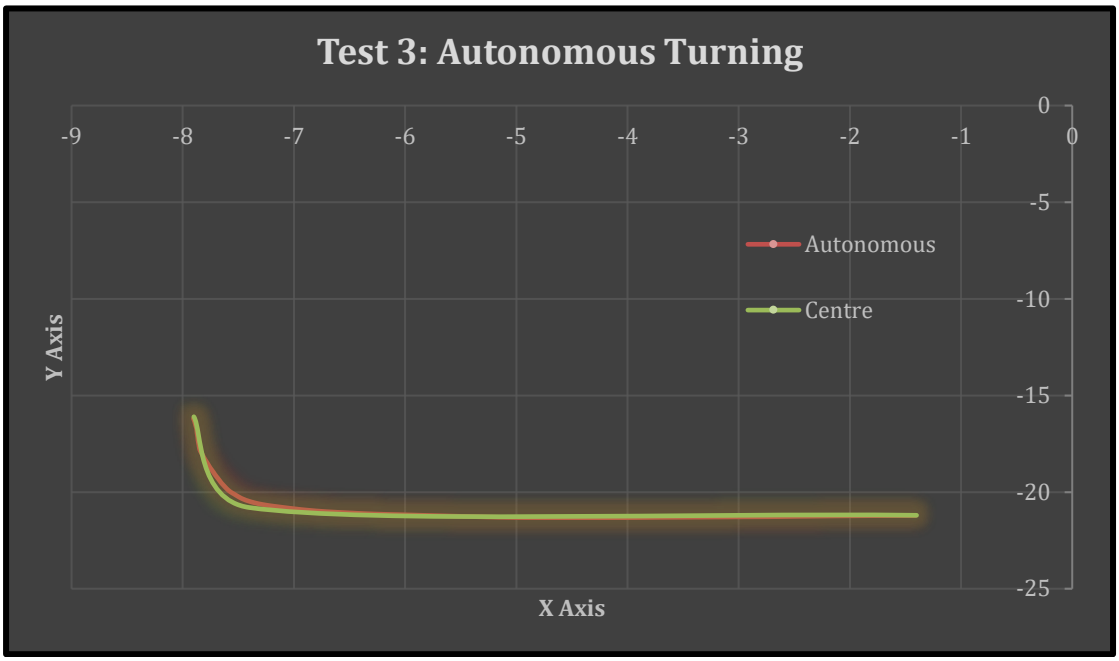

Fig15 (above): Testing results 3

The above graph *(Fig15)* depicts the logger data as the autonomous lane following robot approaches and steers through a sharp right hand turn. As is seen in the scatter graph *(Fig15)* the program successfully navigated this sharp right hand turn, and in general tracked the centre of the lane fairly accurately. The path that the robot took around the corner brought it close to the centre dividing line on the road, but it did not cross it. The long straight section leading up to the right turn demonstrates that the program is capable of continuous driving. Again the data shows that the autonomous lane following favours the left hand side of the lane centre as depicted in the graph, the robot only ventured onto the right hand side of the lane centre when turning around the corner. Note that the test shown in the above graph *(Fig15)* starts from the right and ends at the left.

5. Critical Evaluation

Generally speaking, I believe the project was a success. I was able to design and implement a fully autonomous driving robot program from scratch as per the requirement of the project outline. The project requirements were identified correctly, and provided me with a good target for where I should be at with my work each week as well as providing a clear end-goal for the project.

I enjoyed working on this project, as it challenged my problem solving skills and allowed me to gain more practical experience with multiple computer vision techniques. Beginning this project with very little practical experience with computer vision was a challenge but I believe that I have completed this project to a satisfactory level.

The area of the project that I am particularly proud of is the image processing function. This function was the main area of the project and required a lot of logical thinking to come up with a plausible design. I attempted a number of different designs which all failed, this area was extremely difficult to visualise and required a lot of handwritten pseudocode and planning. The final design makes use of a number of image processing techniques that when put together allow for the accurate tracking of the lines at either side of the lane. The algorithm that then calculates the most appropriate points to use for tracking the edges of the lane provides a decently reliable result. This part of the image processing required the most thought, and was written and re-written many times before settling on a design that was sufficiently accurate.

While the algorithm for the selection of the most appropriate Hough Lines is functional and semi-reliable, It is not perfect. The algorithm has a number of repeated lines of code, and in general is not extremely efficient. I believe I could have executed this part of the project much better if I were to repeat the project. The current design of this algorithm has room for improvement, as it sometimes struggles to tell the difference between the real edge of the lane and other, irrelevant road markings. This can cause the program to suddenly decide the edge of the lane is much further, or closer, than it actually is. This can then cause a ripple effect in the rest of the program, causing the calculation of the centre of the lane to be incorrect which then causes the driving function to make an incorrect decision to turn when in reality it should have kept driving straight.

6. References

[1] docs.opencv.org. (n.d.).
OpenCV: Hough Line Transform. [online] Available at: https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html. [Accessed 5 March 2022]

[2] docs.opencv.org. (n.d.).
OpenCV: Canny Edge Detector. [online] Available at: https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html. [Accessed 5 March 2022]

[3] Weatherley, B. (2021).
Autonomous Driving Robot - Detecting and Reacting to Roadsigns. Supplied 3D models to use in simulated environment.

[4] PySource. (2018).
Lines detection with Hough Transform – OpenCV 3.4 with python 3 Tutorial 21. [online] Available at: https://www.youtube.com/watch?v=KEYzUP7-kkU [Accessed 5 March 2022].

[5] Ososinski, M. and Labrosse, F. (2013).
Automatic Driving on Ill-defined Roads: An Adaptive, Shape-constrained, Color-based Method. *Journal of Field Robotics*, 32(4), pp.504–533. doi:10.1002/rob.21494.

[6] wiki.ros.org. (2017).
cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages - ROS Wiki. [online] Available at: http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages. [Accessed 2 March 2022]

[7] Murtaza's Workshop – Robotics & AI. (2020).
LEARN OPENCV C++ in 4 HOURS | Including 3x Example Projects Win/Mac (2021). YouTube. Available at: https://www.youtube.com/watch?v=2FYm3GOonhk. [Accessed 4 March 2022]

7. Appendices

A. Third-Party Code and Libraries

1) **Ben Weatherley** : **Autonomous Driving Robot – Detecting and Reacting to Roadsigns.** Ben Weatherley supplied me with 3D models from a similar project he completed last year and gave permission for me to use them in my project. These models were used to create the simulated road environment/world for the robot to drive on.

B. Code Samples

1) Image_cb function

This callback function is responsible for making use of the cv_bridge library to convert the incoming ros message containing the raw image to a useable opencv image ready for image processing.

```
void image_cb(const sensor_msgs::ImageConstPtr& msg)
{
    cv_bridge::CvImagePtr cv_ptr;
    try
    {
        cv_ptr = cv_bridge::toCvCopy(msg,
sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
        ROS_ERROR("cv_bridge exception: %s", e.what());
        return;
    }
    // Update image ready for processing
    img = cv_ptr->image.clone();
    imshow("Image", cv_ptr->image);
    waitKey(10);
}
```

2) Position Logger

This is the position logger used for testing purposes.

```
// Include ros lib
#include <ros/ros.h>
#include <gazebo_msgs/ModelStates.h>
#include <iostream>
#include <fstream>

// X&Y positions
double xPos = 0, yPos = 0;

void logger_cb(const gazebo_msgs::ModelStates& msg)
{
    // Set X&Y positions
    xPos = msg.pose[1].position.x;
    yPos = msg.pose[1].position.y;
}

int main(int argc, char **argv) {
    // Initialize ROS
    ros::init(argc, argv, "Pose_Logger");
    ros::NodeHandle mainNh;
    // Subscriber to gazebo/model_states
    ros::Subscriber sub = mainNh.subscribe("gazebo/model_states", 10,
&logger_cb);
    // Open log file
    std::ofstream logFile;
    logFile.open
("/impacs/jow102/catkin_ws/src/jow102_mmp/poseLog.csv");
    // Loop rate
    ros::Rate rate(10);
    // Main loop
    while(ros::ok())
    {
```

```
        ros::spinOnce();
        // Write to log file
        logFile << xPos << "," << yPos << std::endl; //%s, %d",
string1, string2, double1 << endl;
        // Timed loop using ros time - only allows 2 logs per second
        double startTime = ros::Time::now().toSec(), currentTime =
startTime;
        while(currentTime-startTime<0.5){
            currentTime = ros::Time::now().toSec();
            rate.sleep();
        }
    }
    logFile.close();
    return 0;
}
```

3) Bool types for checking and then drawing lines

These bools are used to check if the edges of the lane were able to be detected.

```
// Draw detected lane lines
lBool = false, rBool = false;
if(left[0] != -1 && left[1] != -1 && left[2] != -1 && left[3] !=
1000){
    line(imgHoughLinesP, Point(left[0], left[1]), Point(left[2],
left[3]), Scalar(255,0,0), 3, LINE_AA);
    lBool = true;
}
if(right[0] != -1 && right[1] != -1 && right[2] != 1000 && right[3]
!= 1000){
    line(imgHoughLinesP, Point(right[0], right[1]), Point(right[2],
right[3]), Scalar(0,0,255), 3, LINE_AA);
    rBool = true;
}
```

4) Driving Function

This is the function that controls the driving of the robot based off the centre line. Deadzones are used to make program less sensitive to change.

```
void drive(Publisher pub, geometry_msgs::Twist values){
    // Wait for startup confirmation
    if(wait){
        return;
    }

    // Driving
    int deadzone = 35;
    values.linear.x = 0.2;
    if (xTrack>320+deadzone || !rBool){
        values.angular.z = -0.1;
    } else if (xTrack<320-deadzone || !lBool){
        values.angular.z = 0.1;
    } else {
        values.angular.z = 0;
    }

    pub.publish(values);
}
```

5) Hough Line Selector

This is the code that is used to select the most appropriate Hough Lines.

```
// Left Line
Vec4i left;
left[0] = -1;
left[1] = -1;
left[2] = -1;
left[3] = 1000;

// Right Line
Vec4i right;
right[0] = -1;
right[1] = -1;
right[2] = 1000;
right[3] = 1000;

// Loop through vector, select most appropriate lines for lane
tracking
for( size_t i = 0; i < linesP.size(); i++ )
{
    // *** LINE TRACKING ***
    Vec4i x = linesP[i];
    //line(test, Point(x[0], x[1]), Point(x[2], x[3]),
Scalar(0,0,255), 3, LINE_AA);
    if(x[1]>x[3]){
        if(x[0]<320 && x[1]>left[1]){
            left[0]=x[0];
            left[1]=x[1];
        } else if(x[0]>320 && x[1]>right[1]){
            right[0]=x[0];
            right[1]=x[1];
        }
        if(x[2]<(right[2]-10) && x[3]<left[3]){
            left[2]=x[2];
            left[3]=x[3];
        } else if(x[2]>(left[2]+10) && x[3]<right[3]){
            right[2]=x[2];
            right[3]=x[3];
        }
    }else{
        if(x[2]<320 && x[3]>left[1]){
            left[0]=x[2];
            left[1]=x[3];
        } else if(x[2]>320 && x[3]>right[1]){
            right[0]=x[2];
            right[1]=x[3];
        }
        if(x[0]<(right[2]-10) && x[1]<left[3]){
            left[2]=x[0];
            left[3]=x[1];
        } else if(x[0]>(left[2]+10) && x[1]<right[3]){
            right[2]=x[0];
            right[3]=x[1];
        }
    }
}
```

6) Image Processing

This is all of the code for image processing steps.

```
// Crop image
Rect roi(0,257,640,223);
imgCrop = img(roi);

// Convert image to HSV
cvtColor(imgCrop, imgHSV, COLOR_BGR2HSV);

// Apply colour mask
Scalar lower (0,0,255);
Scalar upper (0,0,255);
inRange(imgHSV,lower,upper,imgMask);

// Apply Canny edge detection
Canny(imgMask,imgEdges, 50, 150, 3);

// Vector to hold results of HoughLinesP detection
vector<Vec4i> linesP;
HoughLinesP(imgEdges, linesP, 1, CV_PI/180, 15, 10, 90); // Detection
```

7) Creation of New Thread

This is the creation of a new thread specifically for the image processing function

```
// Create new thread for image processing
pthread_t thread;
int rc = pthread_create(&thread, NULL, imageProc, (void *)1);
if (rc) {
    cout << "Error:unable to create thread," << rc << endl;
    exit(-1);
}
```