CSCI 5302 Project: Family Not Friends

1st Nishant Bhattacharya
M.S. Computer Science

2nd Aryan Choudhary
M.S. Mechanical Engineering

3rd Nicolas Garzione
M.S. Mechanical Engineering

4th Drew Imhof
M.S. Mechanical Engineering

5th Luke Roberson
M.S. Aerospace Engineering

6th Tanisshk Yadav
M.S. Mechanical Engineering

***Abstract*—Autonomous automobile racing has developed as a fascinating sector in which researchers and industry practitioners can investigate the abilities of autonomous systems. We offer a perception and object detection technique for developing an autonomous automobile racing system of one tenth scale capable of navigating racetracks at high speeds while avoiding collisions in this research. Our results show that our self-driving automobile racing system can attain competitive lap times on specific racetracks. We also discuss the approach's limitations and obstacles, as well as different control methods.**

***Index Terms*—SLAM, Autonomous Racing, ROS, Depth Following**

I. Introduction

In recent years, autonomous vehicles have become one of the hottest topics in the world of technology and transportation. With the potential to revolutionize the way we move, autonomous vehicles represent a major leap forward in the field of transportation technology. Autonomous vehicles, also known as self-driving cars, are capable of navigating without human intervention. These vehicles use a combination of sensors, cameras, and algorithms to perceive their environment, analyze data, and make decisions based on that data. The most advanced autonomous vehicles are able to handle complex driving tasks such as changing lanes, merging into traffic, and navigating through intersections.

This paper focuses on the design and deployment of a 1/10th-scale autonomous vehicle meant to operate within a closed course environment while optimizing numerous objectives at the same time. The team is made up of multidisciplinary fields from Computer, Mechanical and Aerospace graduate engineering departments. The vehicle platform was designed using the ODROID XU-4 single-board computer, which has 2GB memory, and the Intel RealSense D435 camera. The platform also included a 1/10 AMP MT 2WD Monster Truck RTR ECX03028T2, a Mini Maestro 18-Channel USB Servo Controller, an Adafruit IR Distance Sensor GP2Y0A710K0F, and a PhidgetSpatial 3/3/3 Basic 1042. These components were integrated using software tools such as Python and ROS (Robot Operating System) to enable the vehicle to operate autonomously in a closed course environment.

From the various challenges described in the Project draft the following challenges were chosen by the team:

A. Backing out of a collision and continuing driving.
B. Stop at a stop sign.
B. Real-time estimates of coefficient of friction.
C. (Appendix) Report on how deep learning could be used to improve performance.
C. (Appendix) Implement two control approaches and compare them.

A racing day protocol was established to test the abilities of the car. By race day, team had a fully autonomous car, with the only exception in autonomy being an "on/off" signal that communicates to the vehicle over WiFi to start and stop operation. Vehicles simply utilised the chassis, battery, compute, and sensor equipment that was supplied. Furthermore, the vehicle participated in multiple heats, having the best result used to evaluate the team's time and performance on objectives.

The course was held indoors and with suitable lighting, not wider than 4m or narrower than 2m, and free of most (but not all) obstacles. The course was traversed multiple times in a clockwise direction. In the following sections, we will go over the design and implementation of our autonomous vehicle platform.

II. Related Works

Simultaneous Localization and Mapping (SLAM). Simultaneous localization and mapping is an important aspect of autonomous driving to accurately locate the robot's position during operation. The usual methods of SLAM involve reading sensor data and using "landmarks" to accurately predict where the robot has been before, and where the robot should go. The real-world implementations of SLAM are fine-tuned to be application-specific and sometimes include additional features like loop-close detection which helps in map fidelity.

For this challenge we looked at Visual SLAM approaches that rely on RGB(D) images to detect points of interest in the scenery. While they are usually the inexpensive option, they suffer from tracking issues since a monocular cameras lack depth information. To achieve effective vSLAM, monocular cameras are paired with Inertial Measurement Units and/or depth cameras to improve the map. Even with these additions, significant challenges remain, especially when there is excessive blur in the RGB feed that make it difficult to detect conspicuous features, which in-turn throws off tracking. Furthermore, long, featureless hallways are a pain point for such algorithms because the localization errors accumulate

without any landmarks to provide the correction step. There are a couple of vSLAM implementations for the Robotic Operating System(ROS) which stand out for their ease of use and community support. These implementations require a couple of predefined topics to be published which are then subscribed to generate an accurate map for the race track.

The first implementation of vSLAM that stands out is Real-Time Appearance-Based Mapping(RTABMap). It supports a wide variety of cameras including standard RGB, fish eye and depth cameras in addition to LiDAR sensors and IMUs. Different combinations of sensors produce different levels of fidelity and decide the methodology used to generate the final map. RTABMap uses a probabilistic loop-closure algorithm which is followed by a graph optimization stage to refine the final map [1].

For the second approach we look at ORB-SLAM v2 [2]. Although the development has moved on to version 3 of this package, there are couple of key differences between RTABMap and ORB-SLAM v2 which make it an attractive choice. While ORB-SLAM supports the same variety of sensors that RTABMap does, it uses a real-time discreet bag-of-words approach for loop closure as opposed to RTABMap's probabilistic approach. This results in ORB-SLAM requiring a lower processing budget and a higher accuracy for the final output as evidenced by several studies. This is especially important for our resource constrained environment.

The final approaches we look at are gmapping, which uses Rao-Blackwell particle filters for grid mapping [3], and Hector mapping, which uses scan matching and has been shown to be robust in handheld mapping applications [4]. Gmapping in ROS requires both laser scan and odometry information to perform SLAM. Due to only having an IMU for odometry, this may be too noisy in estimating robot position to produce usable results. We are not able to observe our position, due to the need to perform double integration from the noisy IMU acceleration data to arrive at position. Hector mapping on the other hand only requires a laser scan to produce results. The issue with both of these approaches for our application is that we don't have LiDAR, we have a RGBD camera. Calculations can be done to create "point clouds" from the depth image, although these will be less accurate than LiDAR data. Using depth images as opposed to a LiDAR point cloud should reduce the amount of data being used, which is beneficial since we have limited onboard computing power.

Object Detection. There are several object detection implementations ranging from classical computer vision methods to neural network based methods. For this course, we focus on the classical computer vision based methods, since neural networks requires a higher computation budget for accurate results. Classical computer vision based methods like Haar Cascades try to find a set of features on the given dataset which provide the highest level of discrimination between the positive and negative examples. Multiple classifiers are then combined to progressively filter out non-object regions in the image. Histogram of Oriented Gradients uses an edge detector to classify the orientation of edges in the image into objects. The final classification is done using a Support Vector Machine(SVM) to group edges into known object shapes. SIFT and SURF are methods to extract salient scale and rotation invariant features from the image like edges, corners and blobs. These features are then fed to a SVM to arrive at a classification for each object in the image. SURF is a more efficient version of SIFT and is more robust to orientation and scale changes.

Multiple neural network based object detection algorithms exist with the oldest ones being RCNNs, Fast-RCNNs and Faster-RCNNs. These neural networks can have approximately between 30 million to upwards of 100 million parameters. This makes most of these implementations out of bounds for our platform with limited computing capacity even with optimizations like filter pruning and quantization.

State Estimation. There are a number of sensor fusion and filtering algorithms, the most common one being extended kalman filters. Extended Kalman Filters are the non-linear extension to Kalman Filters which are used widely in the robotics community to accurately predict the dynamics of a system by fusing sensor outputs. While Kalman filters can handle linear Gaussian systems very well, EKFs can handle non-linear systems with non-Gaussian noise.

The Madgwick filter is a way to estimate the orientation of an agent in 3D space using an IMU. This algorithm uses a gradient descent based approach to minimize a cost function that measures the difference between the predicted and the actual sensor data. While Kalman filters work under the assumption that the modelled system is fundamentally Gaussian, a Madgwick filter makes no such assumption.

III. Methods

A. Physical Design

Fig. 1. (a) 3D Model of the laser cut platform

We first look at the structure of our car shown in Fig 5. The platform on which all the sensors and the development board is mounted was laser cut out of a MDF sheet 3/8" thick. We designed our front and rear bumpers in SolidWorks and 3D printed them using PLA follwoing the schematics in Fig 2 and Fig 3. We mounted the Intel's Realsense D435 at the front to get the best viewing angle of the racetrack with the

Fig. 2. (b) 3D Model of the front bumper

Fig. 3. (c) 3D Model of the rear bumper

Fig. 4. (d) 3D Model of the camera and IR sensor mount

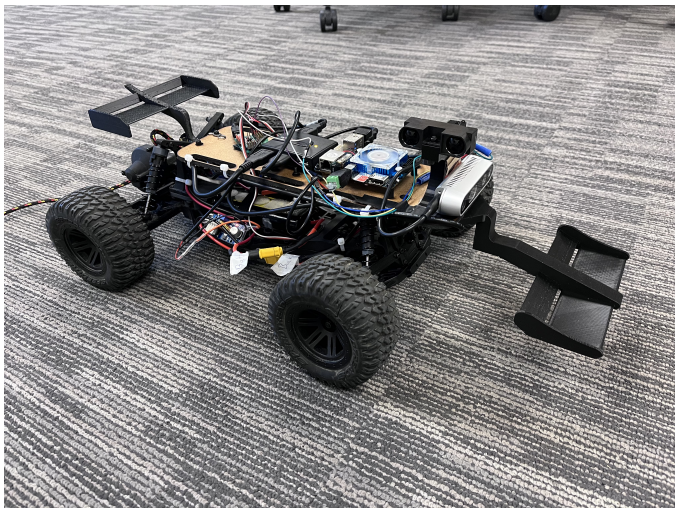


Fig. 5. Physical Design

least possible occlusion. Behind the RealSense camera, the main development board is mounted. We needed a USB hub to connect all the peripherals to the board which include the WiFi card, the sero controller, the IMU sensor, keyboard and mouse for debugging and the servo controller, adn a manual controller when needed. The camera USb was connected directly to the ODROID due to the camera requiring a USB 3.1 connection. The IMU is mounted directly under the USB hub and housed inside a 3D printed box filled with foam to reduce vibrations during vehicle operation.

The circuit diagram in Figure 6 shows how the different modules are connected to each other. The battery supplies a nominal voltage of 11.1 V to the motors. The Odroid has an ideal voltage range of 4.9V to 5.1V. We used an oscilloscope to tune the buck converter to supply a constant voltage of 5.3V under no load which sags to 5.1V under operation.

B. Software Design

The entire operational package is hosted on GitHub and the code is available publicly. We split this section into perception, planning and control which explains the working of the navigation stack as a whole.

1) ***Perception****:* The perception module is an important precursor to the planning phase which decides which action

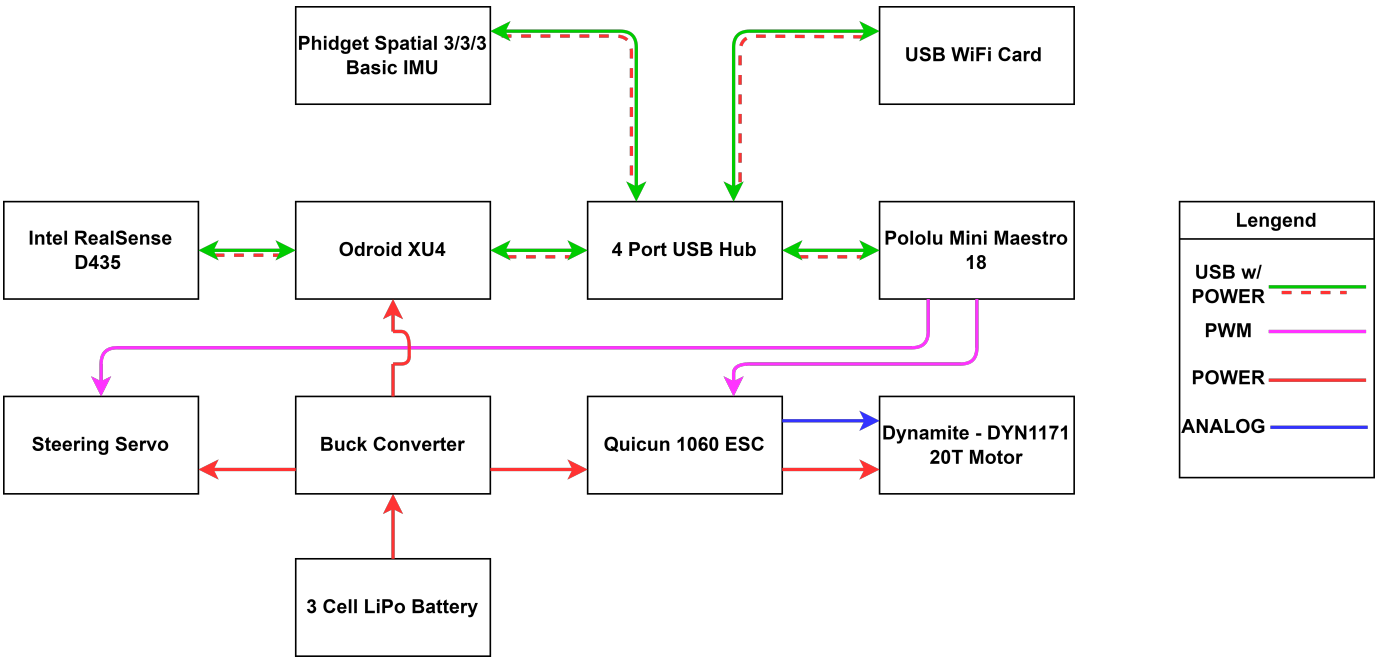


Fig. 6. Electrical Design

to take. The first detection algorithm we run is to detect the desired position of the vehicle using a stream of depth images. The depth feed is first converted to a binary image using a threshold value of 127. A region of interest is then extracted from the image between half the height of image and 0.66 times the height of the image. This region was specifically chosen to remove all the noise that occurs from the reflective surfaces at the bottom of the doorways which tend to throw off the predictions quite a bit. We then use the OpenCV's findContour function to find the largest object in the depth image and calculate its center of mass. This center of mass is then considered the ideal heading of the vehicle. We calculate the true heading of the vehicle by taking the center of the frame and subtracting it from the ideal heading. It should be noted that this difference is only calculated in the X direction in units of pixels. We use the following formula to calculate the heading error,

$$E = COM_x - \frac{width\_image}{16} - CoF$$

where COM_x is center of mass of the contour and CoF is the center of the frame in pixels. The second term in the equation is used to bias the ideal heading towards the wall away from the corner to achieve a racing line for the turns as opposed to always using the center line to improve lap times. The error is then normalized against the width of the image before publishing to the /depth_controller_error topic. To detect the corner, we sum the depth values in the region of interest and threshold it to a value of 7000. This is based on the intuition that when approaching a corner, there would be a wall directly in front of us which would drop the distance values for each pixel. The sum of these values would give us an idea of how close that wall is and we turn when we get close enough by publishing true to the /depth_turn_indicator topic.

We also detect red colors in the image for the stop sign detection task. We first threshold the colors in the image and mask them to eliminate everything else which isn't of importance. We then apply the same findContours function from before to detect large blobs in the image. We order these contours in the descending order of their area and count the number of edges in the shape. If the object detected is a hexagon for 5 consecutive frames, we send a true to the /stop_sign.

In estimating the coefficient of friction, the simple equation

$$\mu = \frac{F}{N} \tag{1}$$

was not determined sufficient to accurately guess the coefficient of friction between the rear wheels and the ground. Instead, a differential measurement using the intended force applied and the actual force applied was used. Equation 46 in "Estimation of Road Friction Coefficient in Different Road Conditions Based on Vehicle Braking Dynamics" [22] was used to find this difference. As the car moves, this approximates the rolling resistance of the car. When the car brakes hard and the wheels lock, this more accurately estimates the coefficient of friction.

$$\mu(\lambda_f) = \frac{2J_f z_2}{m_2 R_\omega g + m_3 R_\omega \ddot{x}} \tag{2}$$

Finally, for crash detection, we monitor the IMU data for a sudden spike. The intuition behind this is that when the vehicle hits an object, the sudden deceleration is recorded by

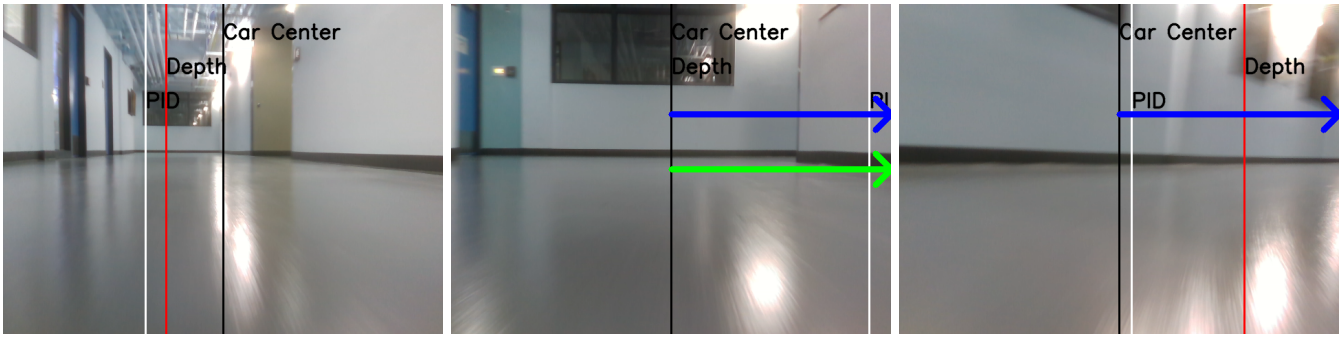
the IMU and could be used as the trigger for the recovery routine. Particularly, we monitor the IMU data for a sudden deceleration of 3m/s in the positive x direction. The positive x direction points from the IMU to the front of the race car, the location on the car that will most likely be impacted.

2) ***Planning****:* We initially wanted to use Visual SLAM algorithms to localize the robot in a global map but we realised that the performance budget on the ODROID doesn't allow us to use vSLAM algorithms in real-time. We instead opt for a simpler vision based planning system to keep the robot moving forward on the straights and we use the IMUs to turn the turn around the corner. We rely on the IMU to tell us the yaw rate of the vehicle at the starting of the turn which should ideally be zero. We can then integrate this yaw rate over discreet timesteps to turn the vehicle left or right depending upon the direction of travel. We hand the control back to the hallway follower once we have achieved a 60 degree turn.

3) ***Control****:* We adopt a bicycle model for our robot which means that there are two variables that we can control, namely, steering angle and speed. Both of these are set using Pulse Width Modulated(PWM) signals which can range from 1000 to 2000 microseconds. The velocity can be changed by supplying a specific PWM to the Electronic Speed Control(ESC). Our ESC's mode of operation is such that a PWM signal of 1500 signifies no movement, 2000 signifies full forward throttle and 1000 signifies full backward throttle. The correlation between PWM signals to velocity is shown in Figure 8 where we measure the time taken between two points separated by a known distance. There are three different modes of operation for the ESC where this behaviour can be altered. The ESC also has an initial calibration step after startup where it expects a PWM signal which it assigns as the neutral, our script sets that to 1500 right up until commanded to go. For the steering we send a PWM signal directly to the servo and there is no calibration required. A 1500 signal represents the steering wheel being straight, 1000 represents full left and 2000 represents full right steering. We wrote two nodes which can supply the PWM values by writing quarter microsecond values to /dev/ttyACM0 which represents the Polulu Maestro Controller. The controller has built in interpolation for consecutive PWM values to reduce fatigue on the structural components.

We implemented a proportional controller with a gain of 1.35 which gets multiplied by the normalized error supplied by the perception algorithm. This then calculates the offset which needs to be added to neutral to arrive at the correct steering PWM value. The straight line velocity is always set to 1640 PWM which is equivalent to roughly 11 miles per hour. During the corners, the speed drops down to 2 miles per hour to maintain traction with the floor. We also introduce some delays in the code between the time the corner is detected and the steering actually happens. These delays are highly sensitive to the velocity values and were tuned very carefully to achieve a good lap time. For the stop sign detection, the robot is stopped at the first detection point of the stop sign and then resumes normal operation after a delay of about 3 seconds. Similarly, for the object collision scenario, immediately after stopping the vehicle we reverse at 1300 PWM for 1 second after a brief pause of 2 seconds. The robot resumes normal operation after another wait period of 2 seconds.

4) ***Depth Following Controller****:* Depth cameras becoming increasingly ubiquitous for robotics applications, incorporates ROS framework providing a powerful and flexible platform for robot control and communication.
Using a depth camera allows the robot to navigate and interact with the environment more accurately and effectively than a standard camera, since depth information provides a three-dimensional view of the scene. By tracking a specific shape, the robot can follow a path or avoid obstacles, and the error signal can be used to adjust the robot's movement in real-time. Hence, this method executes a ROS node that tracks a white form with a depth camera and generates an error signal that can be used to regulate the movement of a robot.
Code Structure and Execution:
The code is organized into single class named DepthFollow-Controller. Upon instantiation, the class initializes a ROS node and sets up subscriptions to the depth image topic published by the camera, publishing data to several topics. CvBridge library is used to convert the depth image received from ROS messages to a format compatible with the OpenCV library for image processing. The main function of the program is the $depth\_image\_callback$ method which is called every time the depth image is updated. The depth image is first cropped to exclude the top and bottom third of the image and the leftmost part to obtain only the relevant portion of the image. This is done to reduce noise and improve the accuracy of the shape detection. The exposure of the image is then adjusted using the convertScaleAbs function of the OpenCV library. Threshold is applied to the image to convert it to a binary format, with white pixels representing the object of interest (the white shape) and black pixels representing the background. Image is then divided into two parts - the upper half and the lower half, which is done because the white shape is expected to be present in the upper half of the image. The number of pixels in the upper half with a value greater than or equal to 250 (white pixels) is counted. If this count is greater than a certain threshold (7000 in this case), it is assumed that the white shape is present in the image and the center of mass of the largest contour (white shape) is calculated using the moments function of OpenCV. Then the program calculates the error difference between the center of the image and the center of mass of the largest topic for use by other ROS nodes. If the error is negative, the robot needs to turn left, so a boolean value of True is published to the $'/depth\_turn\_indicator'$ topic. Similarly, if the error is positive, a boolean value of False is published to indicate a right turn. If no white shape is found in the image, the program publishes a default error value and sets the turn indicator to True, indicating a need for a left turn. In summary, the code provides a basic framework for tracking a white object on a depth camera stream, demonstrating how image processing techniques can be used to extract useful information from an image stream.



(a) (b) (c)

Fig. 7. (a) Initial Corridor Following (b) Depth Controller Initiates Turn (c) IMU Turn Process Continues Turn

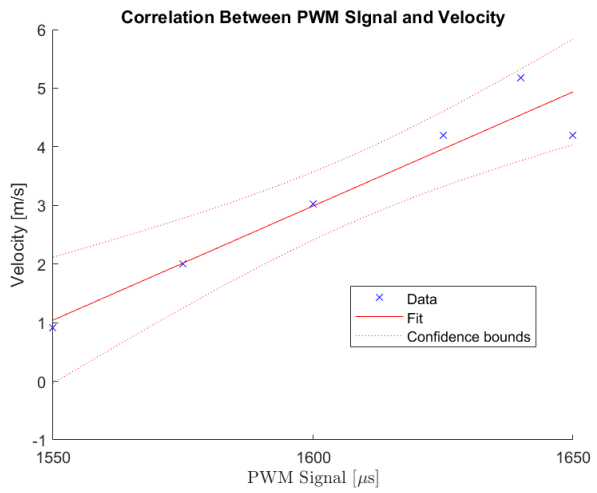


Fig. 8. PWM to Speed Correlation

5) ***Lane Finder****:* This was just a implementation method tried to check how efficient it actually is, although not used in main control application due to computational efforts and inaccuracies in detection.

Code Structure and Execution:

Python script was created that uses OpenCV to detect lanes on a road. It is part of a larger program that runs on a robotic platform equipped with a camera. The purpose of the program is to detect lanes on the road and control the robot's movements accordingly, so that it stays in the middle of the lane. The code defines a class called LaneDetector that contains several methods for processing the camera images and detecting the lanes. Here's a summary of the class and its methods:

- $\_\_init\_\_(self)$: The constructor of the class. It initializes several variables and sets up ROS subscribers and publishers for communicating with other parts of the robot's control system.
- $image\_callback(self, data)$: This method is called whenever a new camera image is received. It converts the image to grayscale, applies several image processing techniques such as Gaussian blur, Sobel edge detection, and Hough line detection to identify the lane lines, and then draws the detected lines on the image.
- $process\_color\_image(self, image)$: This method processes a single color image to detect the lane lines. It converts the image to grayscale, applies a Gaussian blur, performs Sobel edge detection, and then applies a mask to focus on the region of interest. Finally, it uses the Hough line detection algorithm to detect the lines, calculates their intersection points, and draws the lines on the image.
- $sobel\_edge\_detection(self, blur\_img)$: This method performs Sobel edge detection on a grayscale image that has been blurred with a Gaussian filter. It calculates the magnitude of the gradients in the x and y directions, thresholds the magnitudes to remove weak edges, and returns a binary image that contains the detected edges.
- $define\_roi(self, edges)$: This method defines a region of interest in the image where the lane lines are expected to be. It creates a mask that covers the bottom half of the image and a trapezoid-shaped region in the upper half, and then applies the mask to the edge-detected image.
- $calc\_line\_intersects(self, lines)$: This method calculates the intersection points of the detected lane lines with the bottom and top boundaries of the region of interest. It separates the lines into left and right lanes based on their slope, averages the slope and intercept of each lane, and then calculates the x-coordinates of the intersection points using the slope and intercept.

Analysing the issues for inaccuracies in detecting the lanes we figured that following might be the causes:

The code uses a fixed ROI (Region of Interest) for lane detection which we tried to play with, and may not be appropriate for this environment.

The code does not perform any filtering or smoothing of the lane lines, which may lead to jittery or unstable lane detections.

The code does not handle errors or exceptions in a robust way, which may cause the program to crash or behave unpredictably.

The code uses a global variable $(self.last_image)$ to store the last image, which may not be thread-safe in a multi-threaded

environment. Additionally it is most likely possible that the algorithm does not perform well in certain lighting conditions, or when the lane markings have too much reflection.

IV. Results

A. Prior Testing

The team successfully created a robot that was capable of travelling around the hallway below the Intelligent Robotics Lab by utilizing the team's depth following controller described in the Perception section. This controller was capable of travelling around the track in less than 30 seconds at a max speed of approximately 5 m/s in the straight-away sections. Reliability statistics are unknown due to the closure of the closure of the hallway the day prior to the final exam slot which prevented the team from collecting detailed data on the failure rate of the robot in its final configuration. It was determined that the main sources of unreliability were the battery charge level, lighting differences, and speed/angle at which the robot approach's each turn. The battery problem was solved by using a 11.11 V Lipo battery instead of a 7.2 Nimh battery. The lighting differences were overcome by careful tuning of the turn indicating section of the depth following program. The final issue, the speed and angle at which the robot approaches the track, was combated by numerous testing and tuning attempts. New edge cases would appear when the robot was given different speeds, P gains, turning angles, and racing lines to follow. Problems also appeared when the robot dodged obstacles mid-race and the trajectory it exists the obstacle avoidance causes the robot to approach the turns at a new angle and/or speed. This could only be solved by making the program more robust and decreasing the speed. The robot operates on the speed threshold where any faster causes the robot to fishtail uncontrollably and spin out. Further testing could reduce the robustness issue but unfortunately all testing had to cease and no further tuning at the below 30 second lap speed could be conducted.

B. Race Day Testing

The final race for the project was held at the 8th floor of ECOT. The team successfully completed three laps around the track in 34, 33, and 23 seconds. Because the track was novel, the speed of the robot was significantly decreased to be more robust against the more unfamiliar nature of the track. Racing around the floor below the lab during testing, we had been successful in completing laps in under 30 seconds with much longer hallways than the racetrack.

In addition to the race, the team chose to complete "A" and "B" objectives laid out in the introduction. The robot's performance on these objectives is below.

1) A. Backing out of a Collision: Backing out of a collision was demonstrated throughout the race. The robot would collide into a wall then attempt to continue driving. Throughout the teams race trials, the robot successful backed out of every collision it had with a wall (15 collisions). The issue was that the robot was not always able to continue on the course once it collided as it repeatedly attempted to drive through the wall. Out of the 15 total collisions, only 3 of those resulted in the robot successfully continuing along the track.

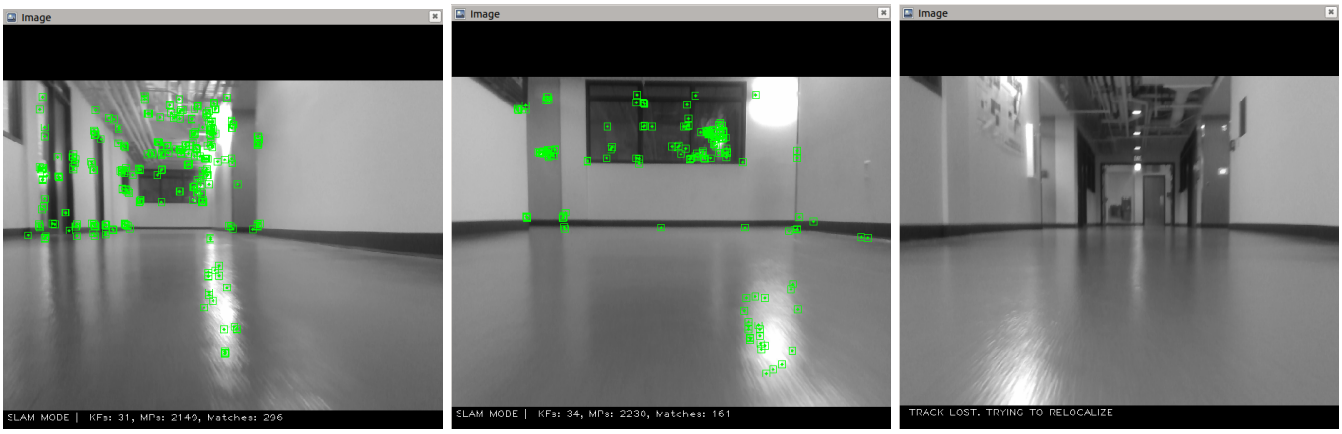
2) B. Estimates of Coefficient of Friction: The estimation of coefficient of friction was performed in the Intelligent Robotics Laboratory using a manual controller to demonstrate different dynamic scenarios. The values fluctuated between -0.02 and 0.02 throughout this demonstration which was not as expected. The team believes that this was because the lack of encoding on the rear wheels. The robot was inaccurately communicating the desired force vs the actual force. This caused the robot to believe it was decelerating while still giving an acceleration and producing a negative coefficient of friction estimate. A better state estimation system which filters the acceleration of the robot and an actual observer of the robot's rear wheel's angular acceleration would improve this implementation. Ideas are to implement the IR sensor as a makeshift wheel encode and draw a white line on the wheel to measure the reflection at each rotation. Another idea would be to use localization data from SLAM to be estimate the velocity and acceleration of the robot. Either way, the friction coefficient estimator did not work as intended in practice but produced expected results when tested on a rosbag of a race lap.

3) B. Stopping at a Stop Sign: The stop sign challenge was executed in the robotics lab. The robot successfully began travelling forward, recognized the stop sign, stopped, then continued forward with the stop sign still in view of the robot.

V. Discussion

While working on this project, the team began with high hopes for the plethora of algorithms that would be implemented and the different ROS nodes that could be used. The team quickly discovered that the Odroid XU4 was much more computationally limited than anticipated, senors were much noisier and unreliable than hoped, and that hardware and software integration are one of the most difficult challenges when developing a robot.

Originally, the team explored the use of pre-built ROS SLAM nodes, but quickly discovered that these nodes required more compute than the ODROID could handle. This lead the team to pursue navigation and control approaches outlined in the methods section and not algorithms such as ORB-SLAM, RTAB-Map, or even GMapping which are all packages that the team explored throughout the design process. Fig 9 shows the images taken during a test run. Initially, ORB-SLAM was able to pick up points when its set down (Fig V), but as the robot moves through the environment, it is unable to close the loop between similar points (Fig V). As the robot moves, it slowly loses track of the original points it picked up on (Fig V). As soon as the robot turns the corner it loses all points and doesn't pick up any more throughout the rest of the lap. Trying to implement ORB-SLAM showed us how sensitive the method is to motion blur. It also didn't seem like the ODROID would have enough compute to be able to use this during a lap. We only tested it on rosbags collected after laps were completed.



(a) (b) (c)

Fig. 9. (a) ORB-SLAM initial results from a test run (b) ORB-SLAM results several seconds later during run (c) ORB-SLAM results after first turn

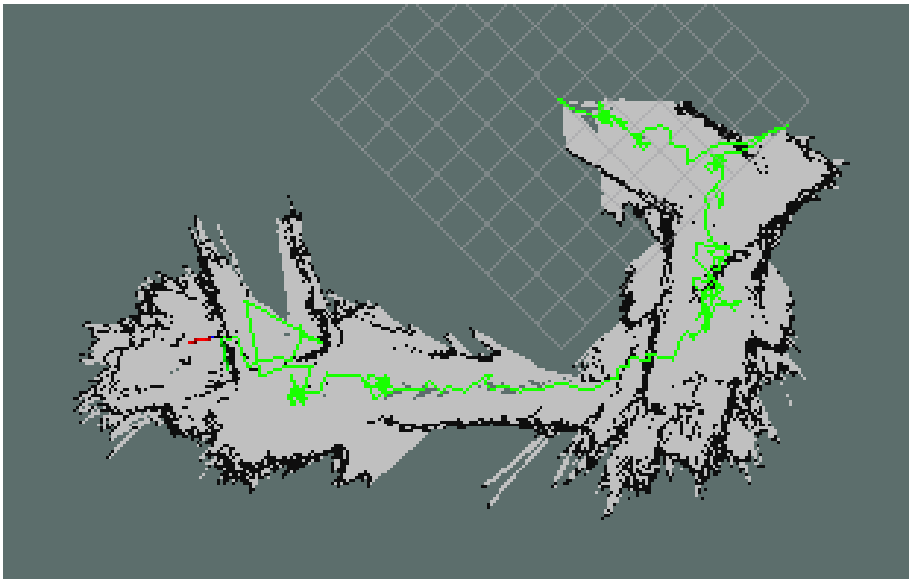


Fig. 10. Map resulting from using Hector SLAM on a race lap

We struggled to get a map using gmapping, even though we used the ROS extended Kalman Filters (EKF) in the Robot Localization package to try to feed GMapping some odometry data using our IMU data and trying to feed back in the PWM values we send to our drive motor which we correlated to velocity from testing. We then attempted to use Hector Mapping, which doesn't require odometry data. This had more promising results then other methods we tried for mapping, but we didn't have as much time to work with this package (Fig 10). We got a better map when fusing in IMU data from the EKF. Trying to get these SLAM packages working taught us a lot about ROS transforms as well as various sensor messages which can be used.

Another challenge the team encountered was the quality of measurements that the robot's sensors produced. The accelerator on the IMU was often too noisy to provide meaningful data for localization and the lack of wheel encoders hindered the robot's motion model for packages that required odometry to function with any amount of accuracy. To overcome the lack of odometry, the team attempted to use the commanded velocities to extract some odometry data, but this was unreliable as well due to the changing behavior of the robot in terms of acceleration and speed throughout consecutive runs.

The depth following worked pretty well at first, the difficulty was implementing the wall detection for turning. Due to the varying hallway sizes, lighting, and surfaces the depth camera was looking at, the depth image used for detection had to have exposure, threshold, and cropping applied. After these changes were applied, all that was required was 50+ laps to tune the depth controller for every possible speed, lighting amount, surface, barrier height, obstacle, and anything else the robot could encounter to make it turn at the wrong time. Testing this in a rosbag had limitations since the rosbag was deterministic and this controller had to be tested on hardware,

in the hallway, and at speed. This process required lots of time and iteration which added to the challenge of creating a robust controller.

The team also spent many hours getting the hardware to function properly when fully integrated. The most difficult issue to debug was issues with USB drivers that caused all the attached devices to crash intermittently. This issue would crash the RealSense camera and cuase the Pololu controller to itermiteently send commands to the servo and motor ESC leading to complete failure of the robot. These issues lead the team to perform multiple clean installs of Ubuntu 18.04 on the Odroid before ironing out all the issues. The team does not understand the complete picture behind why the USB drivers would fail, but believe that pulling and replacing USB devices during runtime may ave contributed to the issues.

VI. Conclusion

The team accomplished a lot while developing their autonomous racing vehicle. Many challenges were faced by the team regarding hardware and software integration, but many hours of hard work eventually led to a successful robot that was capable of completing the racing course under 30 seconds (on the original race track location). The team members have all gained valuable practical experience working with autonomous systems, and will take many of the lessons learned during this class into their future work.

Acknowledgment

Thank you to the TAs, Shivendra Agrawal and Xuefei Sun, for all their help in resolving the issues we had with our hardware and advice for completing different challenges. Thank you to Bradley Hayes and Chris Heckman for teaching us about the different algorithms we tried to implement for the challenge at hand and with all their help in office hours and class, not to mention their helpful comments on Piazza. Thanks for a great class with an interesting and challenging project.

References

[1] Labbé, M, Michaud, F. RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. J Field Robotics. 2019; 35: 416– 446. https://doi.org/10.1002/rob.21831

[2] Mur-Artal, Raul & Tardos, Juan. (2016). ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. IEEE Transactions on Robotics. PP. 10.1109/TRO.2017.2705103.

[3] G. Grisetti, C. Stachniss and W. Burgard, "Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters," in IEEE Transactions on Robotics, vol. 23, no. 1, pp. 34-46, Feb. 2007, doi: 10.1109/TRO.2006.889486.

[4] S. Kohlbrecher, O. von Stryk, J. Meyer and U. Klingauf, "A flexible and scalable SLAM system with full 3D motion estimation," 2011 IEEE International Symposium on Safety, Security, and Rescue Robotics, Kyoto, Japan, 2011, pp. 155-160, doi: 10.1109/SSRR.2011.6106777.

[5] Haarnoja, Tuomas, et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor." International conference on machine learning. PMLR, 2018.

[6] Stachowicz, K., Shah, D., Bhorkar, A., Kostrikov, I., & Levine, S. (2023). FastRLAP: A System for Learning High-Speed Driving via Deep RL and Autonomous Practicing. arXiv preprint arXiv:2304.09831.

[7] Ball, P. J., Smith, L., Kostrikov, I., & Levine, S. (2023). Efficient online reinforcement learning with offline data. arXiv preprint arXiv:2302.02948.

[8] Cai, Peide & Mei, Xiaodong & Tai, Lei & Sun, Yuxiang & Liu, Ming. (2020). High-Speed Autonomous Drifting With Deep Reinforcement Learning. IEEE Robotics and Automation Letters. PP. 1-1. 10.1109/LRA.2020.2967299.

[9] Sun, X., Zhou, M., Zhuang, Z., Yang, S., Betz, J., & Mangharam, R. (2022). A Benchmark Comparison of Imitation Learning-based Control Policies for Autonomous Racing. arXiv preprint arXiv:2209.15073.

[10] Pan, Y., Cheng, C. A., Saigol, K., Lee, K., Yan, X., Theodorou, E., & Boots, B. (2017). Agile autonomous driving using end-to-end deep imitation learning. arXiv preprint arXiv:1709.07174.

[11] Kahn, Gregory, et al. "How We Trained a Deep Neural Pilot to Autonomously Fly the Skydio Drone - IEEE Spectrum." Spectrum.ieee.org, 13 Feb. 2020, spectrum.ieee.org/deep-neural-pilot-skydio-2. Accessed 5 May 2023.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in Neural Information Processing Systems (NeurIPS), 2017.

[13] Prakash, Aditya, Kashyap Chitta, and Andreas Geiger. "Multi-modal fusion transformer for end-to-end autonomous driving." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2021.

[14] C. Li et al., "TLCD: A Transformer based Loop Closure Detection for Robotic Visual SLAM," 2022 International Conference on Advanced Robotics and Mechatronics (ICARM), Guilin, China, 2022, pp. 261-267, doi: 10.1109/ICARM54641.2022.9959319.

[15] Hamid Laga, Laurent Valentin Jospin, Farid Boussaid, Mohammed Bennamoun. A Survey on Deep Learning Techniques for Stereo-based Depth Estimation, April 2022. https://doi.org/10.48550/arXiv.2006.02535

[16] F. Gottmann, H. Wind and O. Sawodny, "On the Influence of Rear Axle Steering and Modeling Depth on a Model Based Racing Line Generation for Autonomous Racing," 2018 IEEE Conference on Control Technology and Applications (CCTA), Copenhagen, Denmark, 2018, pp. 846-852, doi: 10.1109/CCTA.2018.8511508.

[17] A. Sharma, K. S. Reddy, and M. Vatsa, "A Robust and Efficient Real-Time Monocular Depth Estimation Framework for Autonomous Vehicles," in IEEE Transactions on Intelligent Transportation Systems, doi: 10.1109/TITS.2021.3087586, 2021.

[18] C. C. Lin and C. C. Chung, "A Robust Lane Detection Algorithm for Autonomous Vehicles," in IEEE Transactions on Intelligent Transportation Systems, vol. 7, no. 3, pp. 309-321, Sept. 2006, doi: 10.1109/TITS.2006.877882.

[19] A. Mianjy, A. Rahimi and H. Deldari, "A novel lane detection method using image processing and fuzzy inference system for autonomous vehicles," 2019 3rd Conference on Swarm Intelligence and Evolutionary Computation (CSIEC), Fardis, Iran, 2019, pp. 54-58, doi:10.1109/CSIEC.2019.8861563.

[20] S. Rathore, V. Verma, A. Saxena and A. Gupta, "Lane Detection and Tracking Using B-Spline Curves for Autonomous Vehicles," 2019 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS), Goa, India, 2019, pp. 1-6, doi: 10.1109/ANTS47419.2019.9118093.

[21] K. Hwang, J. Lee, and J. Paik, "Real-Time Lane Detection and Tracking using Gaussian Mixture Models," in Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Seattle, WA, USA, May 2015, pp. 3235-3240, doi: 10.1109/ICRA.2015.7139653.

[22] Zhao, You & Li, Haiqing & Lin, Fen & Wang, Jian & Ji, Xuewu. (2017). Estimation of Road Friction Coefficient in Different Road Conditions Based on Vehicle Braking Dynamics. Chinese Journal of Mechanical Engineering. 30. 982-990. 10.1007/s10033-017-0143-z.

VII. Appendix

A. Deep Learning Applications in Autonomous Racing

Deep learning and its application to autonomous driving is a widely studied field and is becoming increasingly relevant with the push for autonomous vehicles and advancements in computing. Research has been conducted using many different kinds of deep learning including reinforcement learning, imitation learning and even transformers. Transformers have recently gained more interest due to popular applications such as ChatGPT and GitHub Copilot. The use of transformers for autonomous driving and racing has not been widely studied yet and this seems like a good area for future studies. This section discusses some of the interesting applications of deep learning as it relates to our challenge of autonomous racing.

1) Learning High-Speed Driving via Deep RL and Autonomous Practicing: Even with manual control, successfully navigating a vehicle at high speeds requires lots of practice to get a feel for how the vehicle reacts in different situations. Creating algorithms to try to autonomously navigate at high speeds is even more challenging.

To solve this problem, Stachowicz *et al.* propose FastRLAP, a system that uses offline learning on prior datasets to learn a navigation policy from a low-speed vehicle. This policy is frozen then deployed to an autonomous RC car which will use this network in combination with online learning during autonomous practice to develop a high-speed driving policy. The reward function of the online learning method attempts to maximize velocity towards the next checkpoint, while penalizing collisions. When a collision is detected, a recovery mode is entered, allowing the robot to reset its testing and continue practicing without human intervention [6].

They bring their online learning to a new environment by taking a slow lap to help the robot get a feel for the course. This had been shown in previous works to accelerate online learning [7]. FastRLAP was able to effective learn high-speed driving strategies in new, real-world situations in under 20 minutes. New environments helped the network to acquire new behaviors and improve its robustness.

These results show the potential for deep RL methods to be deployed with autonomous vehicles, but has limitations which would prevent use for our project. The onboard compute used in this study was a NVIDIA Jetson Xavier NX, which has much better computation abilities than the ODROID-XU4 that we are using. Some of the computation was also offloaded to a desktop via WiFi or LTE to improve performance. The robot is also provided a general map with checkpoints to travel through. We have assumed no prior knowledge of the map for our application.

2) Autonomous Drifting with Deep Reinforcement Learning: Autonomous drifting has traditionally been done using mathematical models of the vehicle's dynamics. Accurately deriving these dynamics necessitates a thorough understanding of the system and and the ability to calculate values such as the tire forces and moments. Besides the vehicle dynamics, drifting is a complicated maneuver that may be challenging to model properly. Model-free deep reinforcement learning could be used to develop a drift controller without modeling of the complex dynamics that go into vehicle drifting. Cai *et al.* [8] used a model-free approach using a soft actor-critic (SAC) to train a closed-loop drift controller.

SAC is a deep reinforcement learning approach where the actor aims to maximize both the expected rewards and entropy of the system. It promotes robustness and exploration, working well for continuous action spaces, such as in robotics. Due to its off-policy methods, it can learn from past experiences and generalize well [5].

The SAC-designed controller used by Cai *et al.* was formulated to replicate professional driver drifting with the goal of allowing the vehicle to drift at high speeds while following specified trajectories.Their testing was all done in the open-source simulator, CARLA, using different racing maps to test the ability of their SAC method to generalise. They compared their method with three other RL methods: DQN, DDPG, and SAC-WOS and found that their method was able to achieve the fastest lap time on each course.

To be able to implement something similar to this, we would likely need to use simulation to develop a drifting model due to the amount of data/evaluations required to develop an accurate model. We would need to ensure this generalizes well to be able to work in real world scenarios. Then we would need to compare simulated performance with real world performance to see if this would be viable on the actual course. Besides the amount of data needed, another potential challenge with this method and other deep learning methods for navigation and control comes from the difficulty in human interpretability of the resulting model. These methods tend to produce black boxes which require extensive testing before deploying in safety-critical applications. For our specific application of racing around a "closed" course, this wouldn't be as big of a concern.

3) Imitation Learning for Autonomous Driving: Along with reinforcement learning, imitation learning (IL) is another machine learning technique which has shown some promising results for autonomous racing, but still has many limitations. Imitation learning uses expert data, typically from a human operator, to train a policy. Up to this point, limited studies have been conducted implementing IL for autonomous racing compared with other more popular machine learning methods. Some of the common approaches used in IL include behavioral cloning (BC) and inverse reinforcement learning (IRL). BC, a branch of direct policy control (DPC), trains a novice policy with an expert policy but has limitations that make it a poor choice for autonomous racing. IRL involves directly learning a reward function, potentially making it better at generalizing than direct policy methods that try to learn expert actions rather than intentions [9].

Dataset Aggregation (DAgger) is a method which tries to improve upon the issues of BC. DAgger is used to continuously learn from new data and new environments. This method has been used by Pan *et al.* [10] to train a scaled down rc car for autonomous off-road navigation using end-to-end deep

imitation learning. The expert is considered to have access to expensive and accurate sensors and makes use of model-predictive control, whereas the learner is a DNN policy with access only to a mono camera and throttle commands. The ability for the learner to have cheaper sensors is a big benefit applications such as ours, as long as one has access to expert-level datasets to learn a policy from. This study attempted DAgger with both online and batch IL algorithms. Both methods were able to achieve similar speeds, although the online learning methods tended to achieve better performance in terms of imitation loss, completion ratio, and generalization. The autonomous vehicle used in this study was equipped with what essentially amounts to a small scale desktop PC, including a quad-core i7 CPU, 16GB RAM, and an NVIDIA GTX 750 Ti GPU. This is another compute system that is much more powerful than the abilities we would have with our onboard ODROID. Our system would likely not be able to keep up with the computation speed required to navigate successfully using this method as the study reports execution of samples and actions at 50 Hz.

Skydio, a company developing autonomous drones used a wide range of purposes such as for defense, safety, inspection, and cinematography, claims to have trained their autopilot with the use of expert data [11]. Skydio had tons of expert flight data available based on on the motion planner used in their autonomy engine. The initial tests they conducted with imitation learning didn't work well as it was not capable of generalizing in challenging situations.

Skydio then moved to an approach they coined Computational Expert Imitation Learning, or CEILing. They don't explicitly state it, but it sounds like CEILing is closer to an IRL approach, as it aims to mimic what the expert cares about and not what it does. They state that CEILing behaved better in high speed scenarios containing dense obstacles, such as a forest, than the previous version of their autonomy. This is due to the ability of the deep neural pilot to use acausal data to know that flying closer to objects such as trees will make the drone more likely to collide with something, such as small, thin branches that would be difficult to see until they are very close.

While their applications may be slightly different than ours, some of the same concepts would still apply. Data could be gathered by driving laps around the racecourse or different potential courses manually. This data could then be used to learn a reward policy for our control system by labeling data off the things that seemed important to the expert operator. This may require adding labels to the data so the learner can properly understand good versus bad behavior. If trained well, the controller could potentially perform better than an expert, by learning patterns from our training data, such as potential under or over-steering caused by different turning patterns. This should also allow it to continuously improve as more testing is conducted. We would need to be careful to make sure the model is not overfitting based on the new data and is instead using it to generalize and learn new techniques.

4) Transformers for Sensor Fusion in Autonomous Driving Environments: With the recent rise in popularity of transformer networks such as ChatGPT, it is only fitting to look at ways in which they could be applied to autonomous driving and racing. The structure of transformers allows them to efficiently learn complex hierarchical representations. They have been shown to achieve state-of-the-art results at certain tasks. They also achieve faster training times than RNNs or CNNs due to their parallelizable architecture [12].

Prakash, Chitta, and Geiger proposed an end-to-end autonomous driving solution for dynamic environments using transformers for multi-modal fusion [13]. They demonstrated that imitation learning policies previously used to replicate sensor fusion methods tend to perform poorly in dynamic situations such as those that would take place in real-world driving scenarios. Their transformer network is used to integrate image and LiDAR representations enabling the self-attention mechanism to gain global context. Testing using CARLA simulation, they found that feeding in multiple scales of resolutions for the image/LiDAR data improved performance. They also found that the ability of the transformer network to focus on dynamic agents and traffic lights to improve its understanding of the global scene enabled it to produce state-of-the-art performance on CARLA. Compared to geometry-based fusion methods, the multi-modal transformer reduced collisions by 76%.

One limitation of the approach was that it tended to run red lights, although all of the other methods they compared their method to tended to do the same. The authors stated that the errors here in their method were because they didn't use semantic supervision for this particular task, thus producing weak signals for the network to learn this. This was an issue that should be able to be improved in future studies when it is accounted for.

The potential real-world scenarios faced in CARLA simulations in this study are certainly more complicated than the rather static world that our rc car should face during the race, but there are still some important potential takeaways for our application. First, transformers seem to be well-suited for dynamic object detection. This would come in handy for the B challenge to avoid the rolling ball or in other dynamic situations like if people are walking on the racecourse. When trained properly, the attention method brings focus to areas that would likely need it the most, such as trying to predict motion of dynamic objects.

Due to their ability perform well in object detection, it would be interesting to observe transformers for SLAM applications, especially in Visual SLAM and loop closure. Studies on this are limited up to this point, but this could be a good area for future research projects. Li *et al.* did just that, applying a transformer for loop closure detection for Robotic Visual SLAM. Their transformer based method performed higher than traditional loop closure detection methods across several public datasets and also provided slightly higher accuracy (3.18%) than state-of-the-art convolutional neural network based loop closure detection methods [14].

B. Comparison of Control Methods

Depth Follower vs Lane Finder:

The general difference between the lane finder and depth follower is as follows:

- Objective: Depth detection is used to gauge the distance between the vehicle and other objects in the immediate environment, while lane recognition is used to locate the car in the lane and prevent it from straying out of it.
- Sensor engineering: Depth detection use a variety of sensors, such as monocular depth estimation algorithms, LiDAR, or stereo vision, whereas lane detecting commonly uses image processing methods with a camera sensor.
- Data processing: While depth detection involves more extensive computation of the sensor data, lane detection algorithms often use computer vision methods to identify the pertinent aspects of the lane markers.
- Intricacy: Due to the requirement to interpret various input sources and combine them into a coherent representation of the environment, depth sensing is typically more complicated than lane identification.
- Efficiency: Both techniques are crucial for autonomous driving, although depth sensing is typically seen to be more necessary for avoiding obstacles and preventing collisions, while lane detecting is more concerned with keeping the car in its proper place on the road.

1) Implementation Comparison:

1) Lane Finder The lane finder was a purely vision based approach on our part to utilize the optical characteristics of the hallway track to maintain the central lane throughout the lap. The objective was to return a way point to steer to which would have ideally remained in the center of the lane. A number of methods were deployed to achieve this.

a) Initial Approach

All approaches began with filtering of the image received from the camera. The first filter was the canny filter. This was utilized to detect and isolate edges from the image.

The image is then subjected to a Gaussian blur and gray scaled for simpler processing. The image by default is too rich and detailed with color which increases compute significantly. The filters do not need high resolution color images to be able to function so this step is beneficial.

The next step involved cropping our image to the region of interest. Most cameras record a far wider image than necessary for detection, to prevent unnecessary detection and anomalous behavior in the program, the image has to be cropped.

We then apply the Hough lines probabilistic filter to detect lines in the image. The Hough transform connects the edges from the canny filter into straight lines by obtaining the most likely segments. It does this by extracting discrete points in the image and running an accumulator array that stores the probability of a line segment existing at that point. The algorithm checks for other points connected to the point to evaluate likelihood of a line existing by converting the point into a radius and slope values and running the probabilistic model across all this curve and incrementing them. Our final step consisted of averaging the lines detected and fitting a curve segment across the lane lines detected from the image. This involved the use of a 'polyfitting' function which created the segment when specified a set of coordinates. This gives our vehicle a permissible environment to operate in and would return way point values for the vehicle to maintain.

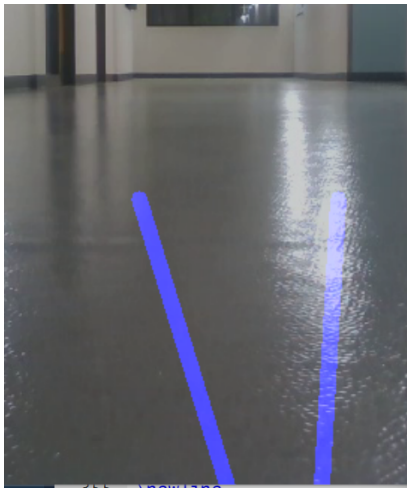


Fig. 11. Lane detection

b) Final Approach

The lane line detection approach did not work in our benefit as the hallway did not have consistent and distinct visual identifiers throughout its length. As a last hurrah, we tried to isolate the black tile feature or the gray flooring in order to attempt lane finding.

Our idea was simpler this time around. We isolated the gray pixels which have the same red,blue,and green value so we could identify different shades of gray. We took a pixel value for the detected gray pixels and hoped to average their image coordinates so we could provide a consistent central viewpoint. The idea was that the way point would lean towards the opposite of the side with the higher concentration of black/gray pixels as this was the black tile feature and make the car turn. As luck would have it, the color gradients were not consistent for turning but the way point was reasonably stable on straights. This was a clean

break from the conventional lane line detection approach but was quite underrated in terms of navigation.

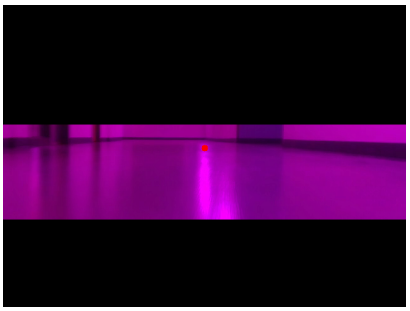


Fig. 12. Gray Detection

2) Depth Follower The term ”depth following” refers to the determination of desired heading by the location of the furthest point in a depth image in reference to the robot’s current heading. The error difference between the desired heading and current heading can be used as a steering input to drive the robot’s heading to face the furthest point away from the robot. This technique is useful when the robot needs to follow a corridor, since the furthest point away from the robot is the desired heading. This approach will drive the robot away from the walls and towards the center of the corridor.

   a) Implementation of Depth Following
   As described in the main report, the implementation of depth following used computer vision algorithms to transform the raw depth camera image in to a point the robot can follow. This involved converting the scale of the image to increase the intensity of the grayscale, making the end of the corridor ”brighter”. This was then thesholded and the center of mass was determined of the resulting ”blob”. This center of mass was used to determine the heading difference and the error was output as a control command. Note in this example image the control output was biased left in order to keep the car left of center and create a ”racing line” for higher speed turns. Without the bias, the controller follows the center of the white ”blob” precisely.

   This control approach falls apart when the car encounters a 90 degree turn. As the car approaches the wall head-on, the controller can no longer give an accurate output due to the depth contour dissipating. When left to just the controller, the direction the car turns is unreliable and often turns in the wrong direction. This was fixed by adding a separate control method using the IMU and the known direction the car has to turn that is triggered when the depth controller can no longer operate correctly. The arrows in the image visualize the depth controller handing off control to this separate process.

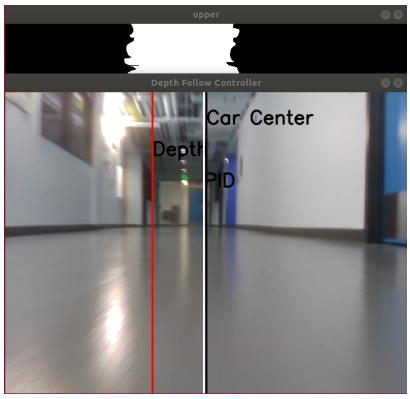


Fig. 13. Depth Following Control Visualization

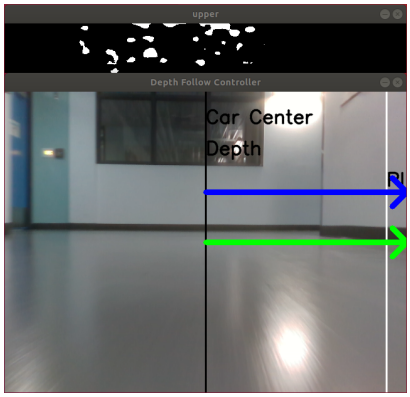


Fig. 14. Depth Controller Failure at Wall

   b) Benefits of Depth Following
   The benefit of depth following was a very easy method of staying in the center of a corridor. The controller naturally acted as an integral controller and drove the car to the center over time. The controller even worked to guide the car through gentler, curved turns. The controller could dodge obstacles naively, since it is trying to find the furthest away point, the obstacle biases the controller away from it and the robot drives around it. The controller could also detect when a wall was directly in front of it and turn out of the way. If no 90 degree turns or alternate corridors were encountered, this controller alone could have been the only guiding process of the robot.
   c) Drawbacks of Depth Following
   The depth follow controller proved to be an unreliable controller. The most drastic influence on the controller was surface reflections. The glossy, reflective floor of the engineering center along with

the reflective metal surface of the lab doors caused the controller to be unreliable. In the depth image, reflective surfaces showed up as the ”furthest” points in the image. This would make the robot drive in to the doors and be confused where to go when looking at the floor. This created a necessity to crop the depth image so tightly it removed a lot of context from the image. In the future, using the entire depth image to guide the robot alongside the color image would be the best approach, since the two images should compliment each other and cancel out each others drawbacks, but unfortunately the lane following was unsuccessful so the robot had to compensate for the depth controller's shortcomings with extensive tuning and automated processes at points when the depth controller no longer works.

3) Bonus: Perspective Convergence An additional approach was attempted where the convergence of the lines of the hallway caused by perspective were analyzed to find the horizon point at which the hallway converges to. This was thought to both act as the depth follow controller but also better predict which direction the robot should make at a turn. This approach looked promising at first, but was deemed too computationally expensive since too many line intesections had to be analyzed to produce a decent control output.

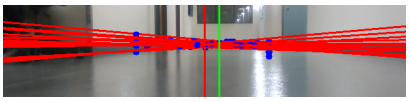


Fig. 15. Perspective Convergence Controller

2) Different approaches of Depth Follower in Autonomous Car:

1) [15] is a study that examines how autonomous vehicles can utilize deep learning methods to estimate depth using stereo images. The authors point out that as it is essential for activities like obstacle identification, scene comprehension, and path planning, depth estimation is a vital issue in autonomous driving. The study gives an overview of the various deep learning methods, such as convolutional neural networks (CNNs), fully convolutional networks (FCNs), and generative adversarial networks (GANs), that have been used for stereo-based depth estimation. The authors go over each technique's advantages and disadvantages as well as the many datasets that have been utilized to develop and test these models. The caliber of the stereo camera system, the precision of the calibration procedure, and the choice of suitable training datasets are some of these variables. The authors also emphasize the significance of taking into account actual aspects like lighting and climate, as these can significantly affect how well depth estimate algorithms function. The paper offers a thorough overview of cutting-edge deep learning methods for stereo-based depth estimation in autonomous vehicles.
It also underlines several crucial factors to take into account when trying to achieve accurate and dependable depth estimate in actual driving situations. The research also identifies certain crucial elements that are crucial for autonomous cars to achieve precise depth estimate.
2) In [16] , proposed is a model-based method for designing racing lines for autonomous vehicles with rear-axle steering and depth sensor. In order to create a reference path for the vehicle to follow, the authors first create a point cloud representation of the track using a depth camera. Based on the reference path and the depth data from the camera, real-time trajectory adjustments are made to the vehicle using the rear-axle steering. The authors test their method on a model racing track and demonstrate that it performs better than a baseline method that does not use depth information when depth information is added to the process of generating racing lines.
The findings imply that including depth information into the control system can improve performance and that it can be a useful source of information for autonomous racing. The study illustrates the potential advantages of employing depth data in the context of autonomous racing and offers insights into how this data might be efficiently incorporated into the control system to produce better racing lines.
3) [17] the authors offer a live monocular depth estimation framework that estimates depth maps from a single RGB image using a convolutional neural network (CNN). The framework's CNN design consists of an encoder-decoder network with skip links. The encoder network employs a ResNet-50 architecture that has been prepared on the ImageNet dataset, whilst the decoder network employs upsampling layers to retrieve the full resolution depth map. By adding characteristics from previous layers, the skip connections across the encoder and decoder networks serve to refine the depth map.
The trained model performs at the cutting edge in terms of efficiency and accuracy when tested on the KITTI dataset, which covers real-world driving events. The authors use the situations of road surface estimation and obstacle detection to show the utility of depth estimation system. The depth map is used to locate and locate impediments on the road in the obstacle detection scenario, and to determine the height and orientation of the road surface in the road surface estimation scenario. The usefulness of depth estimate in the scenario of autonomous driving is demonstrated by such uses.
Overall, the research emphasizes the significance of depth estimation in autonomous driving and suggests a continuous monocular depth estimation paradigm that provides state-of-the-art precision as well as efficacy.
The authors train the CNN using the MegaDepth dataset, a large-scale dataset of deep monocular pictures.

3) Different approaches of Lane Finder in Autonomous Car:

1) Preprocessing, edge identification, curve fitting, and postprocessing are some of the processes that make up the suggested algorithm. The input image is first made grayscale in the preprocessing stage, and then it is filtered to remove noise. The Canny edge detector is used to find edges during the edge detection phase. A polynomial fitting procedure is used to turn straight lines found during the Hough transform step of curve fitting into curves.
   The detected lanes are then filtered based on their geometric characteristics in the postprocessing stage to weed out erroneous detections. The use of a Hough transform for recognizing straight lines and a polynomial fitting technique for fitting curves, which enables the detection of lanes with complex geometry, are among the paper's important components. [18]
2) A brand-new lane recognition technique for self-driving cars is proposed [19]. It makes use of fuzzy inference system (FIS) and image processing. For autonomous vehicles to drive effectively and securely on the road, lane recognition is a crucial responsibility.
   The different phases of the proposed method are picture preprocessing, lane recognition, and FIS-based decision-making. Supplied image is preprocessed in the first step to improve the lane edge and reduce noise. The Sobel edge detection technique and the Hough transform are used in the second stage to find the lane's edges. In the third step, the FIS estimates the location of the vehicle with respect to the lane center and makes judgments based on the state of the road.
3) The solution for self-driving cars in the research is a lane detection and tracking system based on B-spline curves [20]. The suggested method uses a region of interest (ROI) selection algorithm to focus the search for the lane markings to a particular area in the image, a color-based segmentation technique to extract the lane markings, and B-spline curves to model the lane boundaries. In comparison to existing methods that rely on edge detection and Hough transform-based techniques, the authors assert that their approach is more resistant to interference and other fluctuations in the image.
4) For [21] A real-time lane recognition and tracking strategy based on Gaussian Mixture Models (GMMs) is presented by the authors in this work. The suggested method detects lane markers using a sliding window technique and models them as GMMs that provide lane hypotheses. An adaptive programming approach is then used to verify the lane hypotheses, ensuring lane continuity and removing false positives. The important components of this approach are the modeling of lane markings using GMMs, the detection of lanes using sliding windows, and the tracking and continuity of lanes using dynamic programming. In numerous driving circumstances, the approach is proven to be reliable and efficient for lane detecting and tracking in real-time.