ENPH 353 Final Report

Michael Villanueva, #99724791

Zephaniah Ko, #53299798

December 2020

A. Introduction

A.i Our Project

For our ENPH 353 final project, our goal was to develop an autonomous agent to navigate the competition space (Figure 1). Our robot drives through the environment, obeys traffic laws and reports license plates and corresponding parking stalls. This competition takes place in a Gazebo simulation.

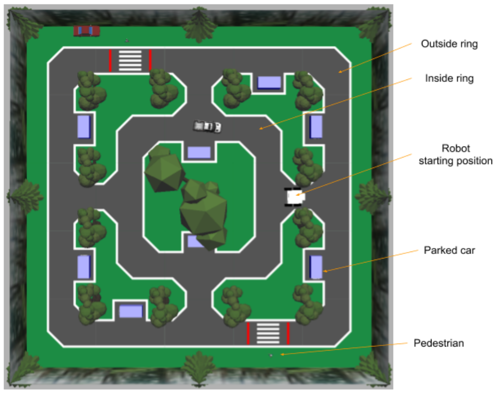


Figure 1: Overhead view of the competition space.

A.ii Our Strategy

For the development of this project, there were two major areas of focus: driving control and license plate detection and recognition.

Driving control could be broken down into completing a lap of the outer and inner circles, detecting and waiting for pedestrians and detecting and avoiding vehicle collision. We decided to use classical computer vision (CV) and proportional integral derivative (PID) control for line following for the outer and inner sections of the track. To detect pedestrians and vehicles, computer vision allowed us to recognize features around the stage to determine if we reached crosswalks or intersections with vehicles. Using background subtraction, we could detect movement at these dangerous areas and control our agent accordingly.

For license plate detection we use a combination of classical CV techniques and a neural network for character recognition. We use a color mask to determine when we are close enough to the car to recognize a license plate. We then apply a Scale-Invariant Feature Transform (SIFT) followed by a perspective transform to get an clear, straightened image of the license plate. By using contour detection, we extracted the relevant text from the plate and fed the image into our trained convolutional neural network (CNN) to generate a prediction of the individual characters on the plate.

These strategies, their rationale and implementation will be covered in more detail in the following sections.

B. Overall Software Architecture

B.i. Software Structure

Our competition was developed in a ROS environment. We are given access to a live camera feed from our agent, the ability to publish linear and angular velocities to our agent, and also can publish license plates. On our end, we naturally created two ROS packages, one for the control algorithm and one for the plate recognition software. Their respective GitHub links are:

https://github.com/Zephko/cnn-enph353
https://github.com/Zephko/control-enph353

The overall structure of our code can be seen in Figure 2. We created two main nodes to run our agent, plate.py and control.py. Here is a brief overview of what each Python file does:

- plate.py:
    - The ROS node for taking in image data, extracting plate numbers, and publishing them to the score tracker.
    - Roughly tracks position of car by detecting large variation in the amount of blue in frame
    - Matches each parking stall to its respective SIFT template
    - Loads the saved NN model and generates predictions
- train_char_recognition.py:
    - A script written to train a convolutional neural network to match images of license plate characters to their respective alphanumeric character representation
    - Uses keras framework to set model parameters and train the model
    - Can generate a new model or load an old model to continue training
    - Generates plots of the accuracy and loss of both the training sets and the validation sets
- augment_data.py and blur_data.py:
    - created datasets of alphanumeric characters to train our convolutional neural network on
- control.py:
    - the main node that controls the agent
    - takes in the live camera feed from the agent
    - with each frame, it can line follow or make decisions about stopping for pedestrians and vehicles

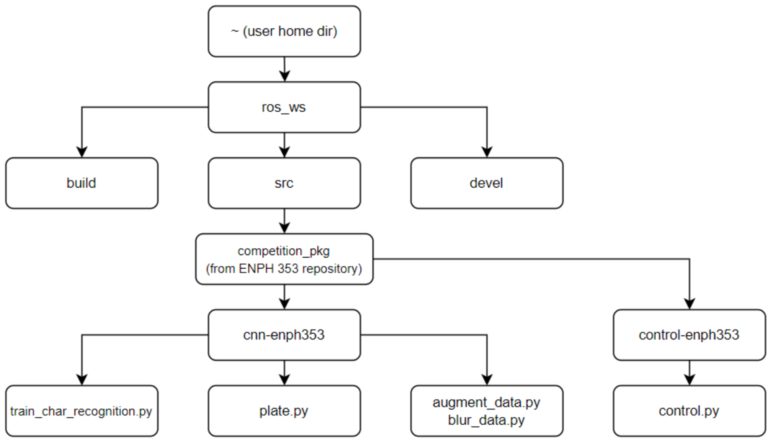

~ (user home dir)
ros_ws
build
src
devel
competition_pkg (from ENPH 353 repository)
cnn-enph353
control-enph353
train_char_recognition.py
plate.py
augment_data.py blur_data.py
control.py

Figure 2: Diagram of our software structure.

C. Plate Recognition

C.i. Perspective Transform

While driving near the cars, SIFT is activated to track key points between saved templates images and the current frame being published from the agent on the ROS network. These saved templates were obtained for each parking stall individually to maximize the number of possible key points to match. Another method we considered was to perform a turn to directly drive in front of the plate each time. We avoided doing this because we would need to break out of our PID algorithm and it would take significant amounts of time to perform each turn.

Ultimately, when we realized we could obtain a nice perspective transform even at extreme angles to the plate, we stuck with SIFT.

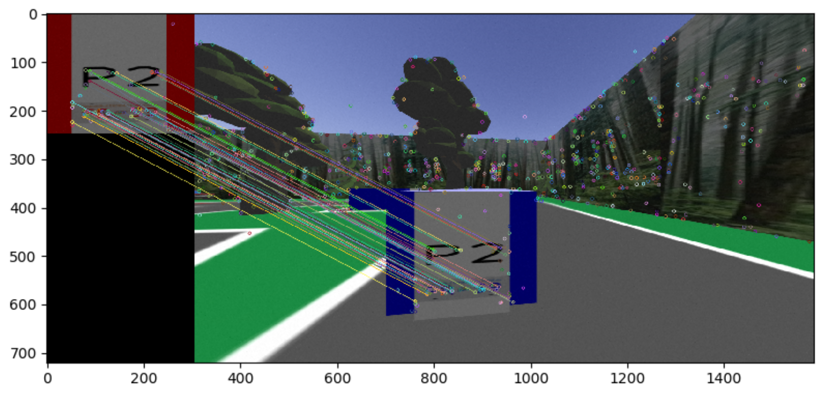


Figure 3: A visualization of feature matching between a template (left) and a frame from the simulation (right).

Once enough matches were found between our template and a license plate, we were able to generate a homography for the plate and perform a perspective transform. This proved to be quite difficult as the homography generated was not always perfect.

In Figure 4, you can see the types of transforms that could result from this method. The image on the right side of Figure 4 is what occurs when the four corners of the template are found to be in four locations in our video feed that did not make sense. For example, if the top right corner of the plate as seen in the template is matched to the bottom left of the plate in the video, then we will get a transform like the one on the right in Figure 4. To prevent these poor transforms from passing on to the character recognition stage, we removed any bad transforms by checking the relative position of the four corners generated from the homography and discarding any that were incorrect.

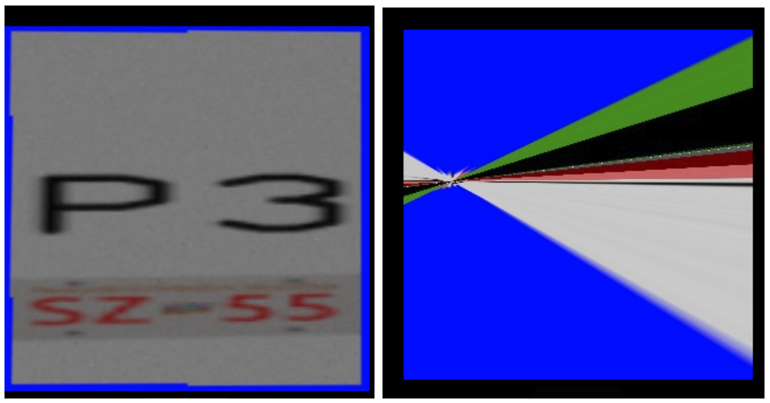


Figure 4: An ideal perspective transform result (left), Often occurring faulty transform (right).

C.ii. Text Extraction

To extract text from the plate, we used contour detection using OpenCV. Originally we had considered simply slicing the image according to set coordinates but this proved to be extremely difficult since the perspective transform would often be skewed. By running the contour detection on a transformed image of a plate, we are able to draw bounding rectangles for each contour even if the plate is not perfectly aligned. To get only the relevant contours we filtered the contours by size, taking only letters and numbers and avoiding the British Columbia flag and the larger contours around the whole image.

We also ran into issues with nested contours. Particularly in characters which have closed loops like the ‘P’ and the ‘6’ in Figure 5. To avoid feeding these contours into the neural network, we leveraged OpenCVs ability to return a hierarchical structure of contours. By examining the parent/child relations between different contours, we could discard the nested contours and only keep the six relevant characters from each car.

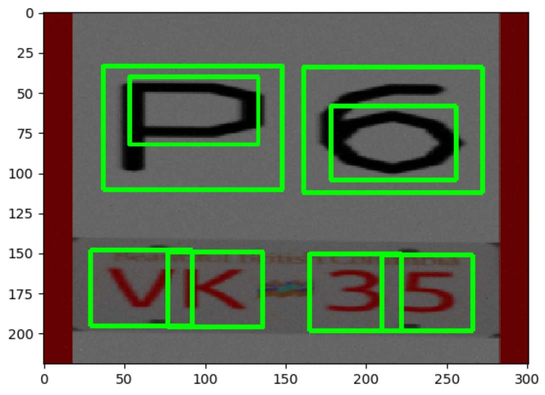


Figure 5: Contour detection on perspective transformed plate with nested contours to be removed.

C.iii. Neural Network (NN) Architecture

Since we are working with image classification, we chose to use a convolutional neural network. As inputs to our neural network we generated images of letters (see detail in C.iv) and matched each of those images to a one-hot encoding format. We went with a fairly common architecture that is commonly used for categorical classification methods. We alternate between applying a convolution with ReLU activation and a max pooling layer. The convolution filters are used to extract high level features from the input image while the 2x2 max pooling layers downsample the data and extract only the dominant features in each 2x2 area. After several repeats, we then flatten the data into a 1D structure and have a dropout layer with a 0.5 probability to avoid overfitting the data. Then the data gets fed through a ReLu activation layer which provides the input to the Softmax layer to classify each input. The resulting output is an array of size 36, each entry representing the probability that the image matches to its own corresponding alphanumeric character.
The full structure is as follows:

- 32 filter Convolutional Layer
- 2x2 Max Pooling
- 64 filter Convolution Layer
- 128 filter Convolution Layer
- 2x2 Max Pooling
- 128 filter Convolution Layer
- 2x2 Max Pooling
- Flatten
- Dropout with 0.5 Probability
- Dense ReLu Activation Layer
- Softmax Layer

C.iv. Training and Validation Testing and Performance

To generate data to train our neural network, we wrote a script that used the license plate generator from Lab 5 to generate approximately random license plates, clip each alphanumeric character and labelled its file name correspondingly. On our first iteration of training the neural network, with approximately 200 samples for each letter, the neural network was able to perform very well on its own training and validation sets. We were able to generate the following plots, converging on a validation accuracy of 0.995 and a validation loss of 0.0330. This statistics, however, are not quite indicative of the performance of this neural network when running in simulation.

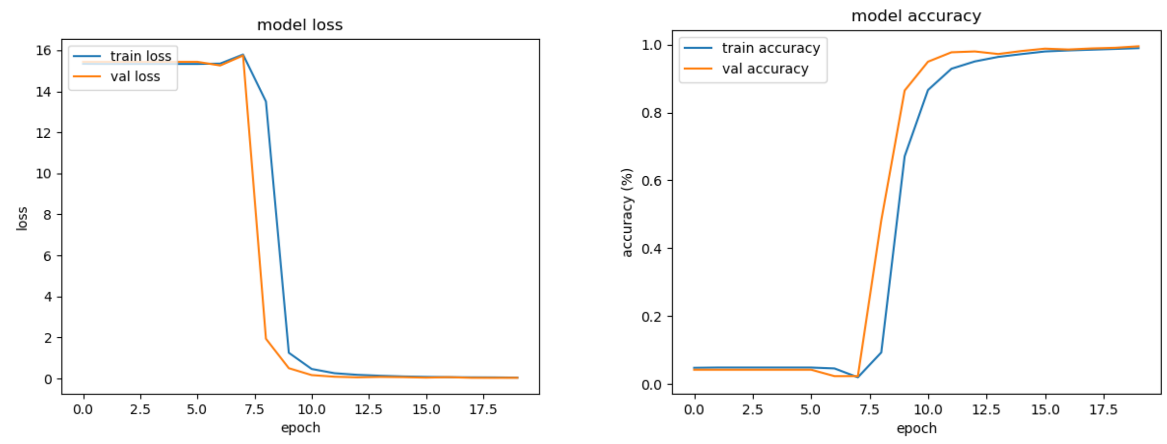


Figure 6: The model loss (left) and model accuracy (right) plots for our first iteration of training for character classification.

By analyzing this initial loss plot, it appears we do have a high amount of bias. This could help to explain why when we tested with images of characters from the simulated environment instead of from the cleaner license plate generator, this trained neural network did not transfer very well immediately. It is important to remember that the validation data was taken from the same original dataset as the training data so the variation between the training and validation sets is minimal.

The solution was to add more variation into the dataset. Looking at the actual plates in the simulation, we saw that the characters were more blurry, darker and out of perspective. To make our data resemble the actual plates, we used a data augmenter from Keras, as well as a Gaussian blur procedure from skimage. We created approximately 400 blurred, rotated, shifted and darkened characters to train from. During the second iteration, with the 400 new augmented images per character, we continued to train the same neural network with the new data. The second iteration of training yielded the following plots.

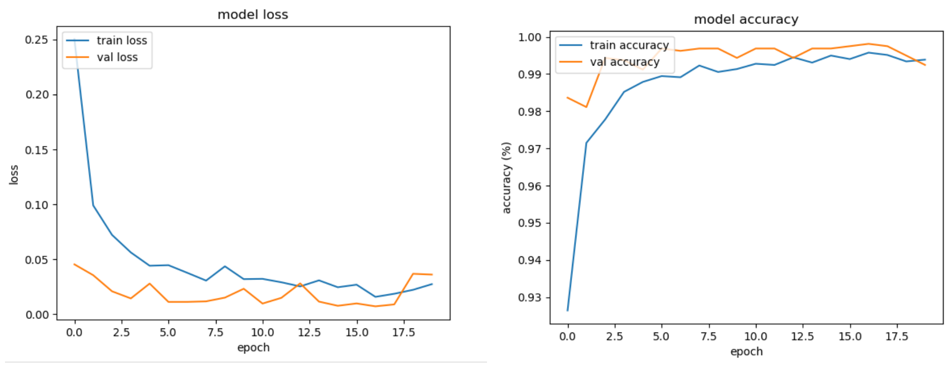


Figure 7: The model loss (left) and model accuracy (right) plots for our second iteration of training starting from the initial trained network.

Though the plots still seem to indicate high bias, the overall accuracy of the neural network improved. Since our training data is so similar to the images from the simulation, training for high bias rather than high variance seems like a good tradeoff in this competition. However, in the real world such a neural network would suffer greatly due to the low variance because license plates could vary greatly in size, colour, and shape. When tested on difficult images taken from the simulation this neural network performed excellently, guessing nearly everything correctly. It often helped to test our trained model and generate a confusion matrix like below to visualize how the neural network classifies the images and where we might need to make improvements. Looking back, our confusion matrix almost always gave us 100% accuracy because we took our validation sets from the augmented data we generated. To get a more accurate representation of performance, it would have been useful to generate validation data from the simulation itself as we drive through the world.

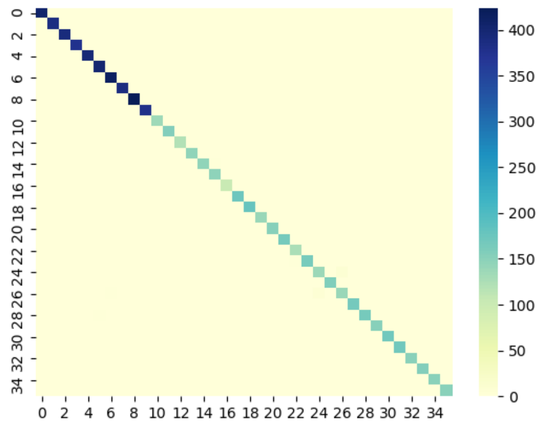


Figure 8: A confusion matrix that shows 100% accuracy of our neural network on a given data set.

The occasional error occurred on difficult to distinguish letters like ‘C’ and ‘G’ or ‘O’ and ‘Q’. We attempted to resolve these small errors by generating a new data set with many more samples of the troublesome letters. The performance when running our simulation did not seem to improve so we ultimately went with the neural network described above.

D. Robot Driving Control

D.i General Driving Control

The structure of our control algorithm is built as a state machine. Input from the camera on our agent as well as information published by our plate recognition node allowed us to switch states and control our robot. The actual control of our robot relied mainly on PID control, with some hard coded turns at key locations.

At the start of our development process, we considered imitation learning and reinforcement Q-learning for robot control. However, we decided early on that we did not have sufficient time before the deadline to reliably implement those strategies.

D.ii Outer Driving Control

At the start of our run, we have a hard coded left turn. We then start our general line following algorithm. Each frame passed in by the onboard camera is first grey scaled then passed through a high threshold. This gives us an image that only has the high intensity white lines. We also perform the CV processes erosion and dilation which remove noise from the frame. By iterating from the right of the frame to the left and stopping when we hit a white pixel, we are able to detect where the right line is relative to our agent, and allows us to adjust accordingly (right-side line following). We can also iterate from the left to right to detect where the left line is relative to our agent (left-side line following). These will both be used in our control algorithm.

For the outer loop, we used right-side line following. This following can be interrupted in three ways: reaching a crosswalk, reaching a parking stall or finding all the parking stalls in the outer loop.

D.iii Pedestrian Detection

On each frame, we apply a mask on the RGB frames to detect the characteristic bright red of the crosswalk markers. We then scan a specific region of this frame. If we detect the presence of this crosswalk red in this region, our agent will stop. The region is chosen so that our agent will stop exactly before the crosswalk. We then enter into a different state where we scan for the pedestrian. We do this with background subtraction. By subtracting adjacent frames from each other, we can detect differences between these frames, which corresponds to movement. Our agent waits for a certain amount of frames that has no movement, drives straight then continues to right-side line follow. Images of the pedestrian detection logic can be seen below.

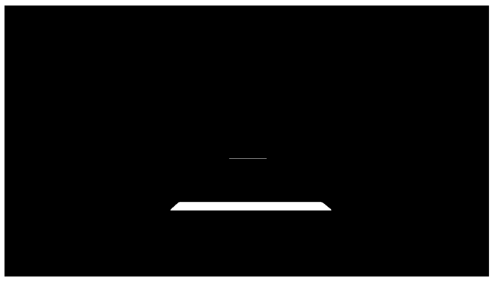
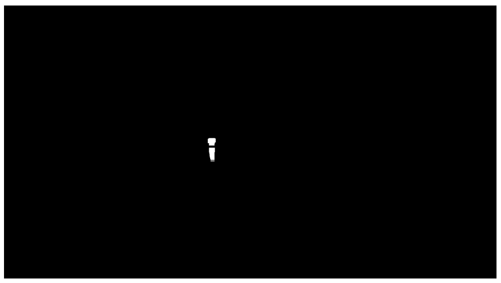



Figure 9.a. (Left): Presence of a crosswalk after applying a mask to the RBG frame.
Figure 9.b. (Right): Presence of a moving pedestrian after applying background subtraction at a crosswalk.

D.iv. Stopping for Plates

Our line following algorithms were also interrupted when we neared parking stalls. Our plate recognition node reports plates more accurately when frames are captured from a stationary agent. Our plate recognition node publishes a boolean value when we near a parking stall. In our control node, we simply stop for a few seconds then continue line following. Furthermore, we also had the ability to stop for longer at certain parking stalls that we found to continually give us problems, such as stall 4 and 6.

The parking stalls for the middle circle are deeper into the grass compared to the outer plates. Thus, we also perform a slight pivot towards the plate when we are in the center loop and receive this message.

D.v. Getting to the Middle

The plate recognition node also publishes when we have reached all six outer plates. At this point, we switch states and attempt to enter the inner circle of the map. After switching states, we also switch to our left-line following algorithm. While we left follow, we again passed our grey-scaled camera frame through a threshold. The purpose of this threshold is to differentiate the colour of the sky from the dark coloured trees and surrounding walls. As seen in Figure 10, on the approach to the inner circle, the top (approximately 100 pixels) of our thresholded view is always obstructed by trees or by the wall. As we continue line following, however, we eventually pass the trees and have a clear view of the sky in the top of our frame. At this point, we then stop our vehicle, pivot slightly to the right, and switch states into vehicle detection.

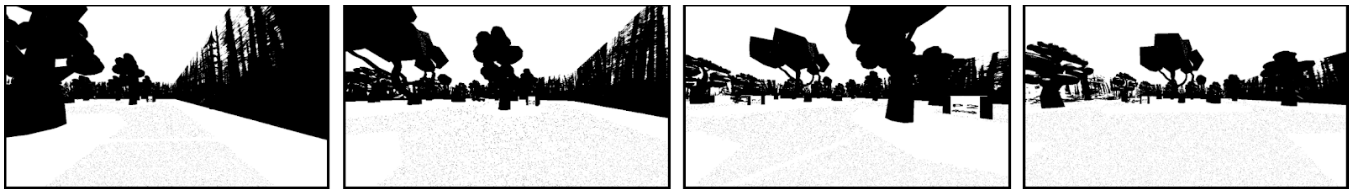


Figure 10: The sky-thresholded view of the agent as it approaches the intersection (from left to right). Note that the top of the fourth image is unobstructed by walls or trees.

D.vi. Vehicle Detection

Through the development of our PID, we realized that we line follow slower than the truck moves. Thus, if we were to enter the circle at a random time, we were likely to eventually get rear ended. Thus, only when the truck passes us, then we can enter the inner ring.

To do this, we perform a background subtraction procedure again, similar to the one that we used for pedestrian detection which is shown in Figure 11. Since we are pivoted to the right, if we see movement, then the car is near us and we wait. If we do not detect any vehicle movement for a certain number of frames (12), then we can enter the inner ring with our hard coded linear drive then left pivot.

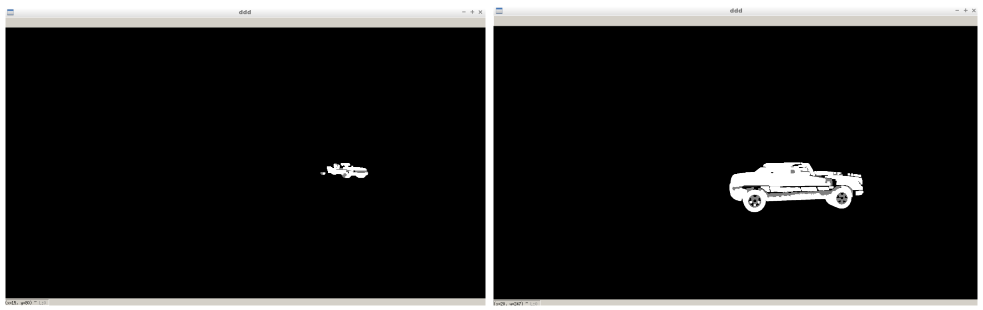


Figure 11: This is the output of the background subtraction procedure as the vehicle rounds the corner and passes in front of our agent.

D.vii. PID in the Inner Ring

The PID control for the inner ring was difficult, since there was no consistent boundary to follow. The left line had breaks where there were intersections, and the right side had breaks where there were parking stalls. To solve this issue, we decided to try using right-side line following by treating the left boundary of the parking stall closest to the road into continuations of the road lines.

To get the boundaries of the parking stall, we applied a mask to the blue colour of the parking stall. This worked quite well, and the summation of the lines and the side of the parking stall can be seen in Figure 12.a. However, a recurring problem that we had with the inner circle was that the parking stall shapes were too far to the right relative to the line. Thus, in testing, when we attempted to follow the line produced by Figure 12.a., we would always sharply turn and crash into the parking stall.

To solve this issue, we simply shifted the mask of the parking stall 150 pixels to the left. This result can be seen in Figure 12.b. Although the stall and line may not look continuous, this is an issue with how the frames look distorted from far away, but line up as you get closer. Shifting the frame made our line following much better. This iteration of our inner circle following had results comparable to our outside line driving.

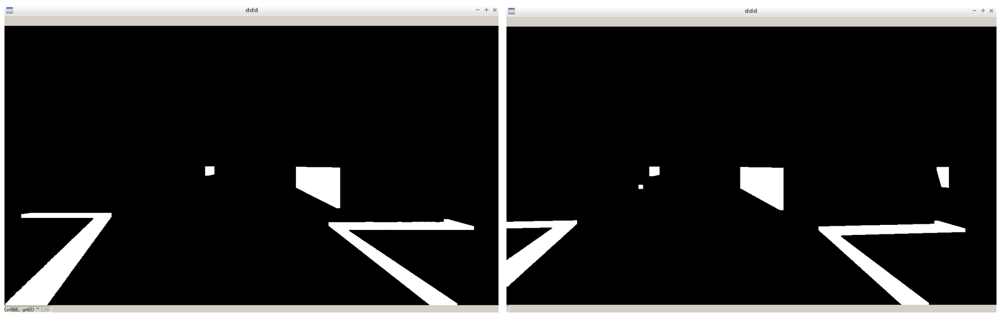
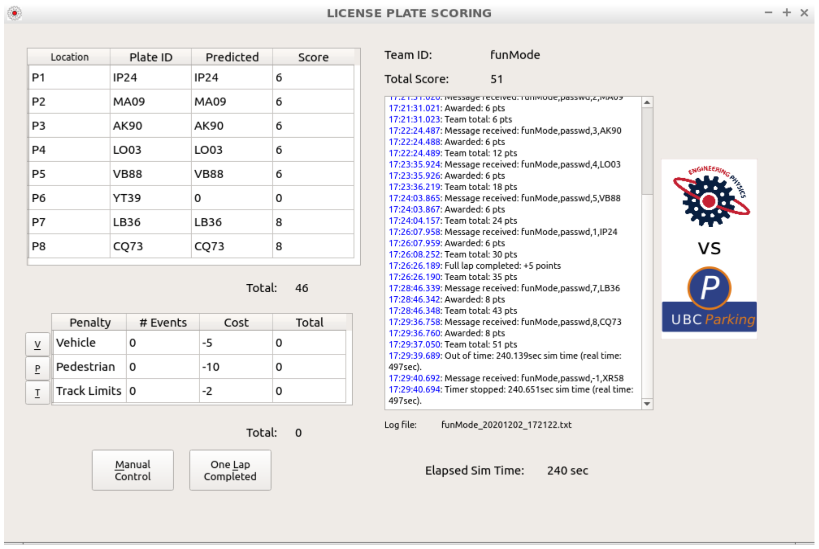


Figure 12: The left image (a) is the view before shifting, and the right view (b) is after shifting.

E. Summary

E.i Overall Performance and Competition

Throughout the days leading up to competition, we were able to set expectations for our robot. We have seen it achieve a perfect score in 220 seconds, but usually will score somewhere in the 40s. Our competition score was 51! We are very happy with our performance and to see our hard work pay off!



LICENSE PLATE SCORING

Location	Plate ID	Predicted	Score
P1	IP24	IP24	6
P2	MA09	MA09	6
P3	AK90	AK90	6
P4	LO03	LO03	6
P5	VB88	VB88	6
P6	YT39	0	0
P7	LB36	LB36	8
P8	CQ73	CQ73	8

Total: 46

Penalty	# Events	Cost	Total
Vehicle	0	-5	0
Pedestrian	0	-10	0
Track Limits	0	-2	0

Total: 0

Team ID: funMode

Total Score: 51

```
17:21:31.020: Message received: funMode,passwd,2,MA09
17:21:31.021: Awarded: 6 pts
17:21:31.023: Team total: 6 pts
17:22:24.487: Message received: funMode,passwd,3,AK90
17:22:24.488: Awarded: 6 pts
17:22:24.489: Team total: 12 pts
17:23:35.924: Message received: funMode,passwd,4,LO03
17:23:35.926: Awarded: 6 pts
17:23:36.219: Team total: 18 pts
17:24:03.865: Message received: funMode,passwd,5,VB88
17:24:03.867: Awarded: 6 pts
17:24:04.157: Team total: 24 pts
17:26:07.958: Message received: funMode,passwd,1,IP24
17:26:07.959: Awarded: 6 pts
17:26:08.252: Team total: 30 pts
17:26:26.189: Full lap completed: +5 points
17:26:26.190: Team total: 35 pts
17:28:46.339: Message received: funMode,passwd,7,LB36
17:28:46.342: Awarded: 8 pts
17:28:46.348: Team total: 43 pts
17:29:36.758: Message received: funMode,passwd,8,CQ73
17:29:36.760: Awarded: 8 pts
17:29:37.050: Team total: 51 pts
17:29:39.689: Out of time: 240.139sec sim time (real time:
497sec).
17:29:40.692: Message received: funMode,passwd,-1,XR58
17:29:40.694: Timer stopped: 240.651sec sim time (real time:
497sec).
```

Log file: funMode_20201202_172122.txt

Elapsed Sim Time: 240 sec

E.ii. Reflection

Although we did not have the time in this project, looking into imitation learning or reinforcement learning for driving control would be very interesting. In the competition, we saw how well the groups that implemented these performed. On the same note, finding ways to limit the amount of processing per frame would help our PID control.

Line following was difficult to get working and even more difficult to get our agent to go faster. The result of this can be seen in the development of our agent. For example, to not get rear ended by the inner vehicle, we have to wait for the inner vehicle to drive by us. We can also see in the results later that we were the slowest agent in comparison to similar scoring teams. Line following was a difficult task due to the amount of processing we were doing on each frame. On each frame, our code had to check various boolean values to determine state, and also perform many CV transformations. Each frame was at least transformed five times and also had to be scanned for the position of the lines for following and for the presence of crosswalks. As a result, we received frames very slowly and our agent had to PID control with a very low frequency of data.

Finally, the real time factor (RTF) of the simulation and related performance of our computers caused a lot of problems through our development of our robot and PID control. We first noticed these problems when we ran this code on different computers. The laptop that the PID code was mainly written on had good performance, while the agent almost couldn't follow on the other laptop. Furthermore, even on the same laptop, the varying real time factor caused many problems for our PID control including waiting for pedestrians or vehicles. Furthermore, the laptop that we were running the simulation on with Zoom screen share resulted in a real time factor of approximately 0.45. As a result, testing full runs took about 10 minutes and it was very hard to make small adjustments and test the agent in the days leading up to competition.

Despite being able to accurately predict most license plates, the speed of our method was not optimal. The reason for this is likely because SIFT is a compute heavy process. By eliminating SIFT in favour of finding the corner points of the license plate by applying a mask, we may have been able to generate more predictions for the characters on the plate and chosen the prediction with the highest probability for a certain character.

Overall, this project and course has been a very valuable learning experience that allowed us to explore computer vision, neural networks, reinforcement learning. By experimenting with these techniques we gained a much stronger understanding of how to develop solutions to problems in autonomy. We discovered how factors such as noise and the compute speed of our software can greatly affect our ability to implement an autonomous agent and how we can mitigate any resulting errors.