ENPH 353: Final report

Michelle Cao and Manmeet Kaur

Introduction

Our general strategy for the competition incorporated the following components:

- Using HSV thresholding for road detection and finding the centroid of the road
- Using PID for driving and controlling the robot's motion around the outer and inner loop
- Detecting movement and the presence of a pedestrian through HSV thresholding and change in pixels in subsequent frames
- Extracting license plates from parked cars through HSV thresholding, contour detection, and perspective transform
- Reading license plates from extracted images using a CNN
- Generating data points and applying Gaussian blur to better represent simulation conditions
- Taking data points (pictures of the license plates) from the simulation to train the CNN model
- Testing the CNN model in the simulation and fixing any accuracy issues by better thresholding the parked cars or feeding the model more data points

General Software Architecture

In broad terms, we implemented two separate files for the overall model: one for robot control and one for license plates. The general file system hierarchy is as follows:

- ros_ws was the root directory, and the src folder inside it contained the competition environment code, gymGazebo, and the controller package
- controller_pkg (repository created by us in src) contained three separate folders: Line_follower (the license plate and robot movement code), launch (the launch files for the world), and node (subscriber and publisher communication nodes)
- Line_follower: contained two separate python files
    1. move_robot.py: PID code
    2. COMPETITION_VER_license_plates.py: license plate detection, license plate prediction, and score_tracker code
        - Note: the exact file used during the competition was integrationOnly5.py. COMPETITION_VER_license_plates.py is a

copy of integrationOnly5.py created post-competition with all unused variables and commented-out code removed
    - The code for creating the CNN used in the COMPETITION_VER_license_plates.py file is stored in a separate Google Colab file (see link below)

The subscriber nodes for the robot (image feed, score_tracker, timer, etc.) were in the competition package. To communicate to these, we published to the above-mentioned nodes by initializing them in the robot file.

License Plates

License Recognition Google Colab Notebooks

1. ImageDataGenerator version
2. Competition version

License Plate Detection

SIFT did not work well for our team, so we decided on contour detection and perspective transform for license plate detection. Our team used the following process:

1. **Filter out everything except the blue areas of the cars using HSV thresholding.**
   The best HSV ranges for each car were found individually and a masked image was created for each. The masked images were then summed to create one masked image that filtered out everything besides the blue areas of the eight cars.

2. **Identify the rectangle containing only the car position and the license plate using contour detection.**
   This was done using cv2.findContours(). Our team chose the cv2.CHAIN_APPROX_SIMPLE method for speed and the cv2.RETR_EXTERNAL mode to limit the identified contours to only outer contours. We then filter out all contours except the two with the largest area and apply binary thresholding before determining the position of the two contours relative to each other. Taking the right edge of the left contour and the left edge of the right contour, we then find the points of a bounding rectangle for the two edges and crop the image to the bounding rectangle.

3. **Align the image using perspective transform.**
   We do this by calculating the height to width ratio to find the desired height of the output image. After adding an offset factor to the desired height to

achieve a ratio closer to the desired image, we locate points for perspective transform in the source and destination images and complete the transformation.

4. **Crop the image to only the license plate**.
   This was done by summing the first fifty columns of a grey-scale version of the aligned image and identifying subsequent rows with an intensity change greater than a certain threshold. However, this did not work at all times due to noise. To account for this, we determined the approximate location of the license plate in the image (approximately the 800th row to the 1100th row through trial and error) and filtered out all of the identified indices that were within 800 pixels of the top of the image and 100 pixels of the bottom. We then cropped the image to the first and last identified indices.

5. **Resize the image to the 600 x 298 dimensions expected of the machine learning pipeline.**
   This process worked well when the robot's camera direction was parallel to the road and the robot was within approximately half a meter of the license plate it was detecting. However, this was only the case for cars P5 and P2. The positions of the rest of the cars in the outer loop were immediately after a turn, so the robot would not properly align with the car in time. A potential solution to this that we did not have time to implement is modifying the contour detection target to the license plate and car position directly rather than the blue car. This would be more robust, as our current method may cause issues if there is more than one car in the frame at a time.

   Another idea we didn't get the chance to implement is using bounding rectangle detection to crop the license plate into its four characters and resize them to the desired size instead of hard-coding the crop positions. This would've made our image preprocessing more adaptable to images that were sheared or had a width or height offset.

Neural Network Architecture

For our neural network, we created a convolutional neural network using two convolution layers and ReLU activation. We used training data as our metric. Dropout and early stopping were implemented to reduce the likelihood of overfitting and ensure the trained model would generalize well.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 163, 103, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 81, 51, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 79, 49, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 39, 24, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 59904)             0
_________________________________________________________________
dropout (Dropout)            (None, 59904)             0
_________________________________________________________________
dense (Dense)                (None, 512)               30671360
_________________________________________________________________
dense_1 (Dense)              (None, 36)                18468
=================================================================
Total params: 30,709,220
Trainable params: 30,709,220
Non-trainable params: 0
```

Figure 1: Model summary for model used in competition

Some ideas for improving model performance that we did not have time to implement or test include:

- Using validation loss as our metric instead of training accuracy to reduce the chances of overfitting
- Using ROC AUC as our metric instead of training accuracy to reduce bias

Training Data Generation and Acquisition

The model used in the competition was trained only on 260 generated license plates, with Gaussian blur applied to better match competition conditions. Variation in blur intensity was achieved by choosing a random Gaussian kernel size between 19 and 27 for each input image. This gave better results than the models we trained with other sets of training data, including:

1. 200 generated license plates with no data augmentation
2. 500 generated license plates with no data augmentation
3. 1080 generated license plates with data augmentation (Gaussian blur, noise, channel shift, shear, and zoom)
4. 1080 generated license plates + 160 license plates acquired from the simulation with data augmentation (Gaussian blur, noise, channel shift, shear, and zoom)

Our team ran into model bias issues with the first two sets of training data, where the model would be heavily biased towards a few letters and numbers. Model 2 was slightly better than the first and was able to correctly identify one license plate in a few trial runs. We suspected the prediction bias was due to a class imbalance in the training data, so the set of 1080 generated license plates was created with a letter counter to ensure an equal distribution of letters (80 of each). Models 3 and 4 employed ImageDataGenerators so that our team could use Keras preprocessing layers for data augmentation. However, the resulting models were even worse than Model 2, so we started from scratch and stopped using ImageDataGenerators for our final model. With our final training data set of 260 generated license plates, we implemented a number counter in addition to the letter counter already implemented. The resulting model was much more accurate than previous ones, but still biased towards T's and 7's (specifically, if the model received a badly cropped image as input, it would almost always predict TT77).

Our team trained one last model using the 260 generated license plates that our competition model trained on plus a few extra generated license plates that focused on letters and characters that our competition model incorrectly identified in our trial runs. However, we ran into file corruption errors and did not have time to field test the newly trained model before the competition, and thus fell back on the previous model.

Some ideas that we had but didn’t get the chance to implement are training a model only on license plates acquired from the simulation and experimenting more with data augmentation. Our team also attempted training on dynamic thresholded images, but we ran into errors that we didn’t have time to fix.

Training and Validation Test Performance

Training and validation test performance was evaluated based on both training and validation loss. With more time, we would've used a test set in addition to a training set and a validation set to ensure that a certain model was not overfitting and would generalize well.

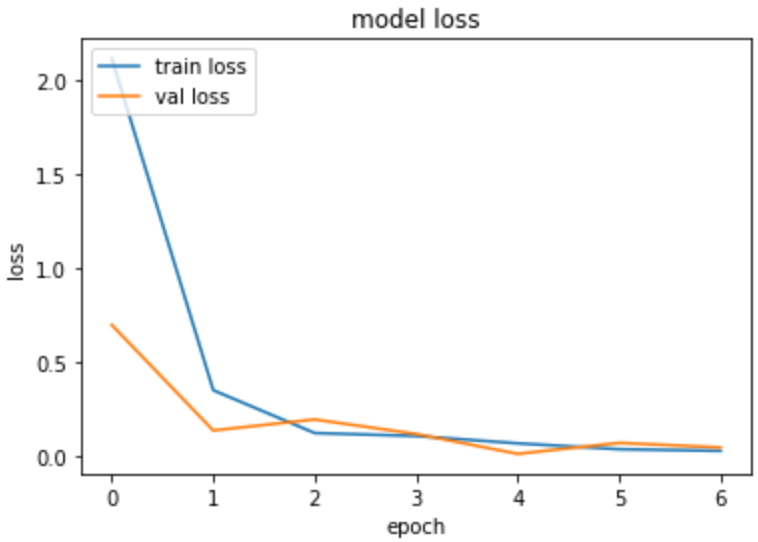

Figure 2: Plot of model loss vs. number of epochs for model used in competition

A confusion matrix for the validation set was also plotted at the end of training to visualize the model's performance on each character.

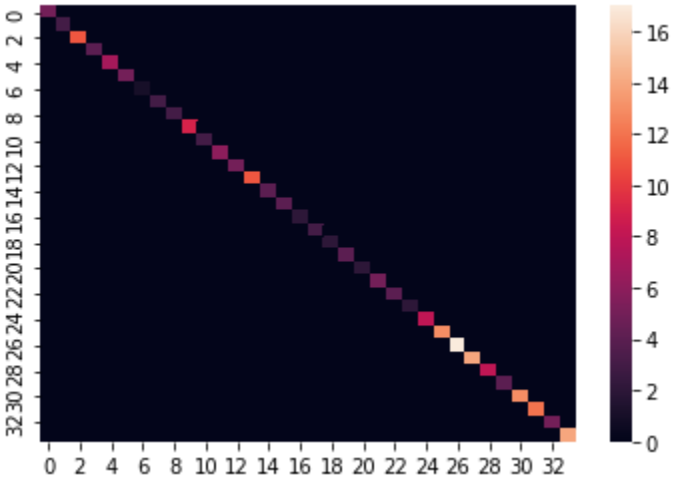

Figure 3: Confusion matrix of model used in competition

Robot Control

Driving

For driving the robot around the outer loop our team implemented PID. The PID algorithm was based around finding the centroid of the road and maintaining that centroid in the center of the robot's frame at all times. The difference between the centroid of the road and the robot's center (center of the image feed) was used as

the error term in our calculations. The general formula used to calculate PID values was: Kp*P + Kd*D + Ki*I. We did not implement derivative and integral terms. Tuning the PID and perfecting the line following included the following components:

1. Finding the centroid of the road

We first thresholded the road using binary thresholding and HSV thresholding. This was done to eliminate any surrounding noise in such a way that the robot would only detect the gray area of the road. Our first method was to calculate the centroid of the road using a for loop and moment of inertia calculations but since this computation would take place inside the callback function (which is another for loop), this increased the running time of the callback function to $\Theta(n^2)$. This in turn slowed down the processing and delayed the response of the robot to detecting the centre and actually staying in the centre.

This issue was mitigated using an OpenCV built-in function called cv2.moments() which calculated the x and y moments of inertia giving us the value of the centroid. This improved the robot's response by quite a bit.

Another issue that we encountered with driving and the image frame was that the width of the image feed was too big to accurately calculate the center of the road (which only occupied ⅓ of the frame). Therefore, we cropped the image to extract the middle ⅓ of the frame which further helped in eliminating any noise from the surroundings and better centre the robot on the road path.

To aid us in detecting, debugging, and resolving the above issues, we displayed three image windows in total--one with the raw image from the robot's camera, one with the cropped frame and binary thresholded image, and one with a circle drawn on the centroid of the road to visualize and affirm that our centroid calculations were correct. Displaying the image feed cut down debugging time from one day to 3 hours.

2. Trial and error for the correct P value (and a D value to reduce oscillations)

We implemented the PID value as a function of the angular speed with a fixed linear speed.

After finding and confirming the centroid calculations for the road path, it was a matter of choosing the correct P value in order for the robot to respond fast and efficiently to the error difference. In the beginning, we performed trial and error and figured out the best P value to be around 9. This led to a huge performance issue that we did not anticipate in the beginning: on the sharp 90 degree turns, the error increased by more than 60% and this increased the maximum angular speed from 0.5 to 1.2 which is more than a 100% increase. This led to oscillations and the robot going off track frequently.

We resolved this issue by placing an upper bound on the maximum angular speed and this value was 0.5. Similar was done for the minimum value which was set to -0.5.

3. Problems with the first turn and hard-coding it using time intervals

Despite implementing the PID and reducing the current frame of the robot to be in the middle ⅓, the robot had difficulty with the first turn. We attempted everything from adjusting the P value to implementing a D value but nothing seemed to work. This was resolved by hard coding the first turn using the time elapsed between the start of the motion of the robot vs. the current instant in time. This difference was calculated as follows:

- A time variable was initialized in the def init function as follows: self.Time = rospy.getTime()
- In the callback function, the difference was calculated as follows: diff = rospy.getTime() - self.Time

One of the most prominent issues with using rospy.getTime() to initialize a variable was the discrepancy between the initialization time of the subscriber and publisher nodes. This led to unpredictable behaviours including corrupting the value of the above-mentioned self.Time variable. This was resolved by using adding time.sleep(1) at the beginning of the initialization function.

Pedestrian

Our primary method of pedestrian detection was HSV thresholding combined with frame subtraction. Our first implementation was to sleep the robot for a certain time interval at the red stop line and then start looking for the pedestrian. We quickly realized the problem with this algorithm: since the robot's processes were

put to sleep when it detected the red line, it was too late to respond to the pedestrian moving in its frame. This resulted in consistently hitting the pedestrian no matter how small the sleep interval was. The solution was to make the linear and angular speeds fully zero at the red line and keep detecting the pedestrian. Using HSV thresholding and the absDiff method in openCV, we were able to detect the pedestrian. The absDiff method was used to compare the robot's current and the subsequent frames to find the change in the number of pixels. If the change was consistently greater than a threshold value, this meant that the pedestrian had crossed the road and it was time to zoom! We also checked for the case where the pedestrian did not spawn properly and there was no pedestrian. This meant that if the change in the number of pixels was smaller for 3 or more frames than the threshold value, there was no pedestrian and the robot would safely move.

The response time after detecting the pedestrian on the road was a small window since once the pedestrian crossed the road, it would turn around and continue to do so. To respond in such a short interval, we implemented a sequence where if the above conditions (pedestrian detected 3 times or no pedestrian for 5 times), the robot would move at 5x (0.5) times its original speed for a certain number of seconds! This resulted in the robot tipping over once it tried to restore its state back to the original linear speed (0.08). This was resolved by gradually decelerating the robot from 0.5 speed to 0.2 to 0.1 and finally to 0.08. This reduced the sudden jerk experienced by the robot and prevented tipping.

Inner Loop

Although in our current implementation, we did not implement the inner loop, we had ideas for it and are hoping to implement these over the break. Some of the ideas include:

- Complete one lap on the outside loop and then turn at the first turn by ignoring the road in front and taking the left ⅓ of the image feed.
- Once on the pathway to the inner loop, stop completely and wait for the car on the loop to pass and get ahead of the robot in order to prevent being rear-ended by the vehicle. This can be achieved by another set of HSV values which would be tailored to detect just the car and eliminate the surrounding noise.
- Once inside the inner loop, maintain the same speed as the car inside.
- Navigation on the inner loop can be done the same way as the outer loop or can even be programmed using the time difference between the time when the robot entered the loop and the current time.

References

Contour Detection and Perspective Transform

1. Contour Detection and related OpenCV methods
2. Information about perspective transformation
3. https://www.geeksforgeeks.org/perspective-transformation-python-opencv/
4. OpenCV methods for perspective transformation
5. Bounding rectangle questions and required answer
6. OpenCV documentation on contours

Image Data Augmentation

7. Image augmentation
8. Image augmentation example
9. Adding gaussian noise to images

Appendix

Image windows used in debugging and visualizing the robot control environment:

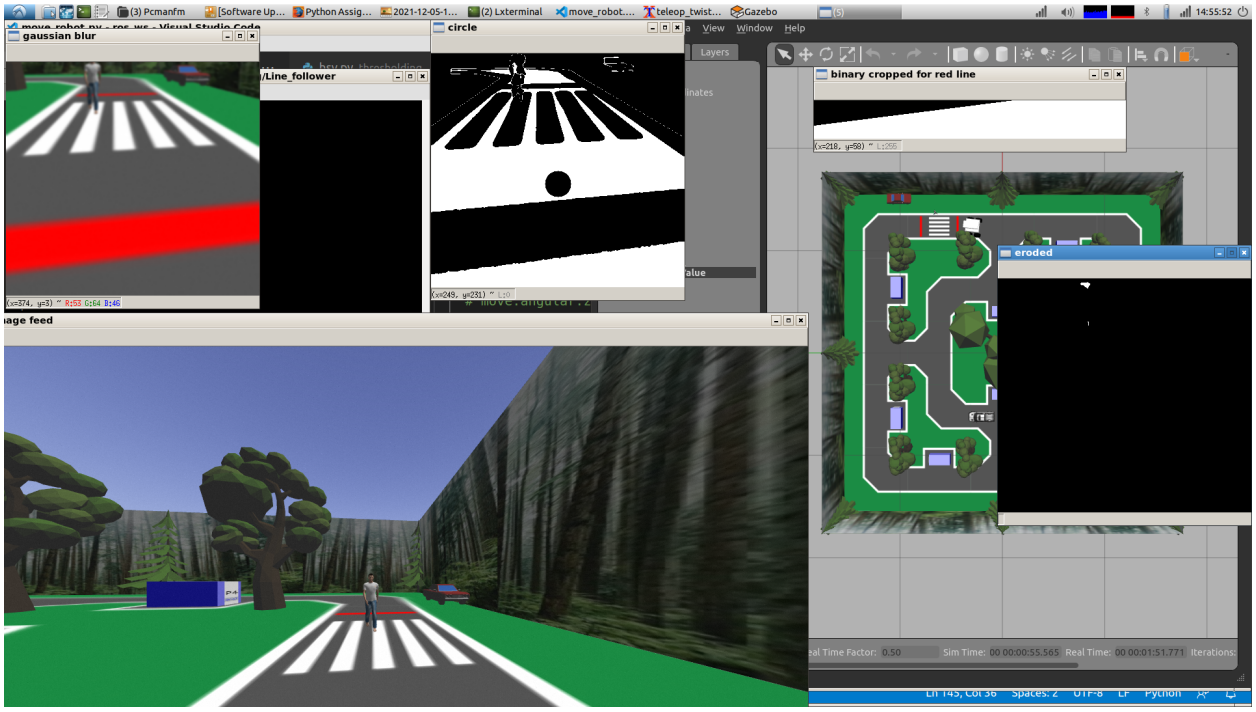

Figure 4: The four image windows used to debug the motion code.

- Top Left: Gaussian blur applied to the original frame
- Top Middle: Binary image result of thresholded road path window

- Bottom Left: The raw image from the robot's camera feed
- Right (labelled eroded): Binary image result of the thresholded pedestrian view window

As we can in the above image, in the "circle" window, there is a black circle. This circle represents the centroid of the road and moves as the robot moves. The centroid shifts to the right or left as the road path changes.

The following images show the same result but from different angles:

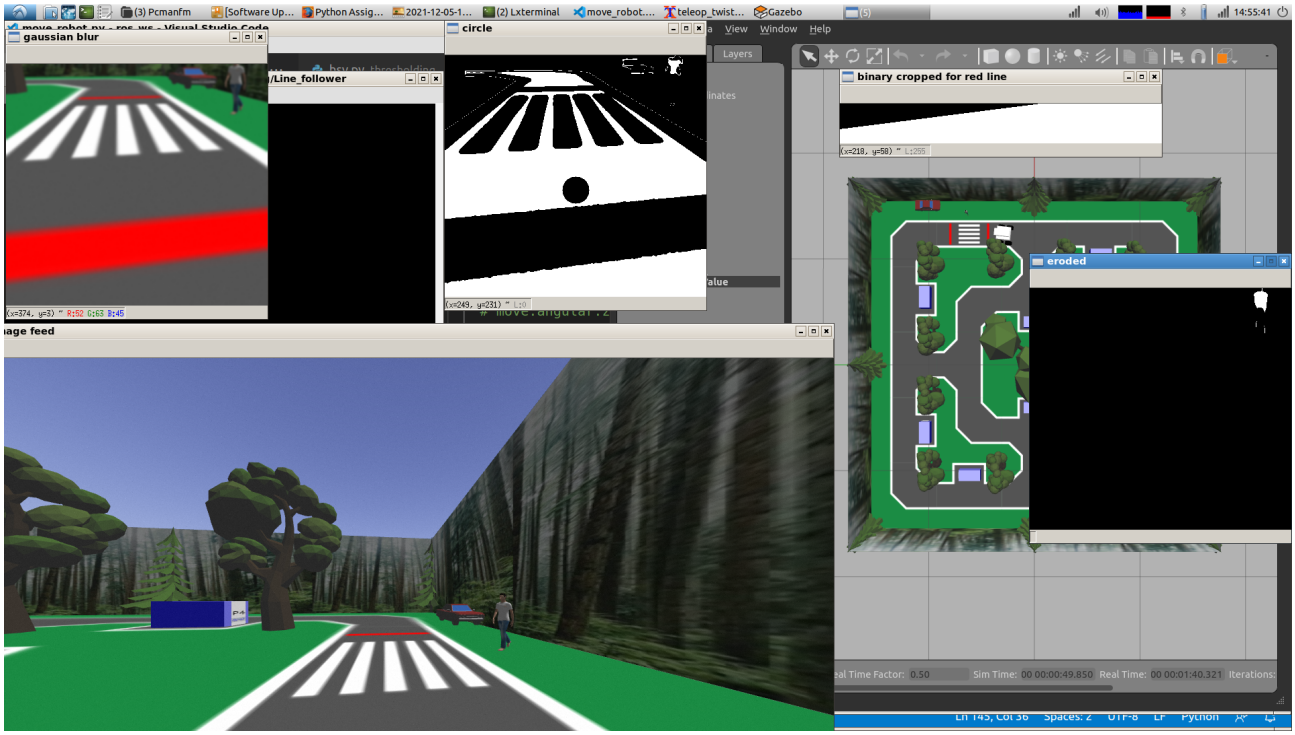

Figure 5: Debugging windows from the simulation showing the different views (as described in Figure 4's description).

Observe that in the above image, the pedestrian's image (white pixels in the "circle" window) have moved to the right since the pedestrian moved. The change in the position and number of these white pixels was used to detect pedestrians and their motion on the road.

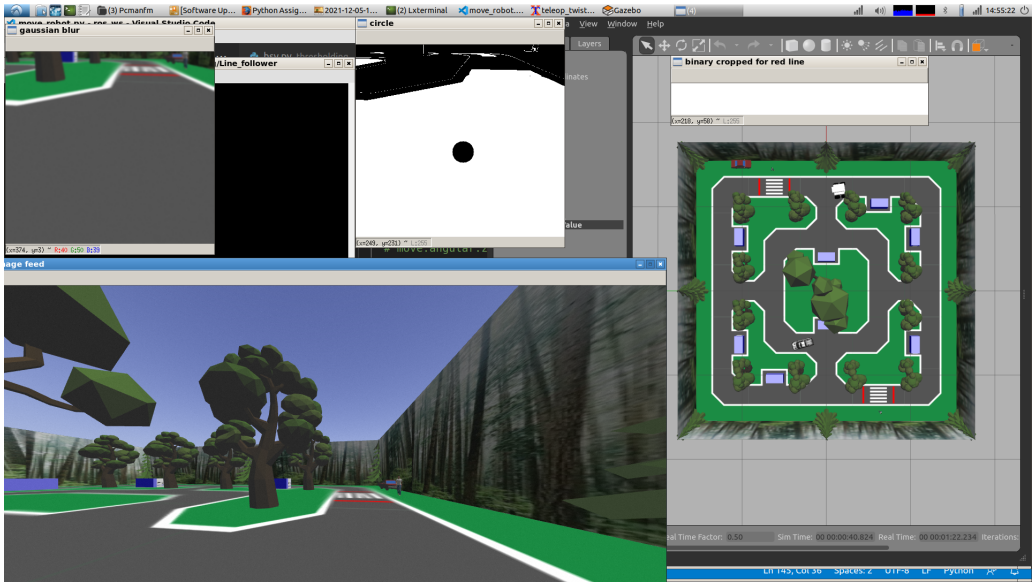

Figure 6: Image windows from the simulation

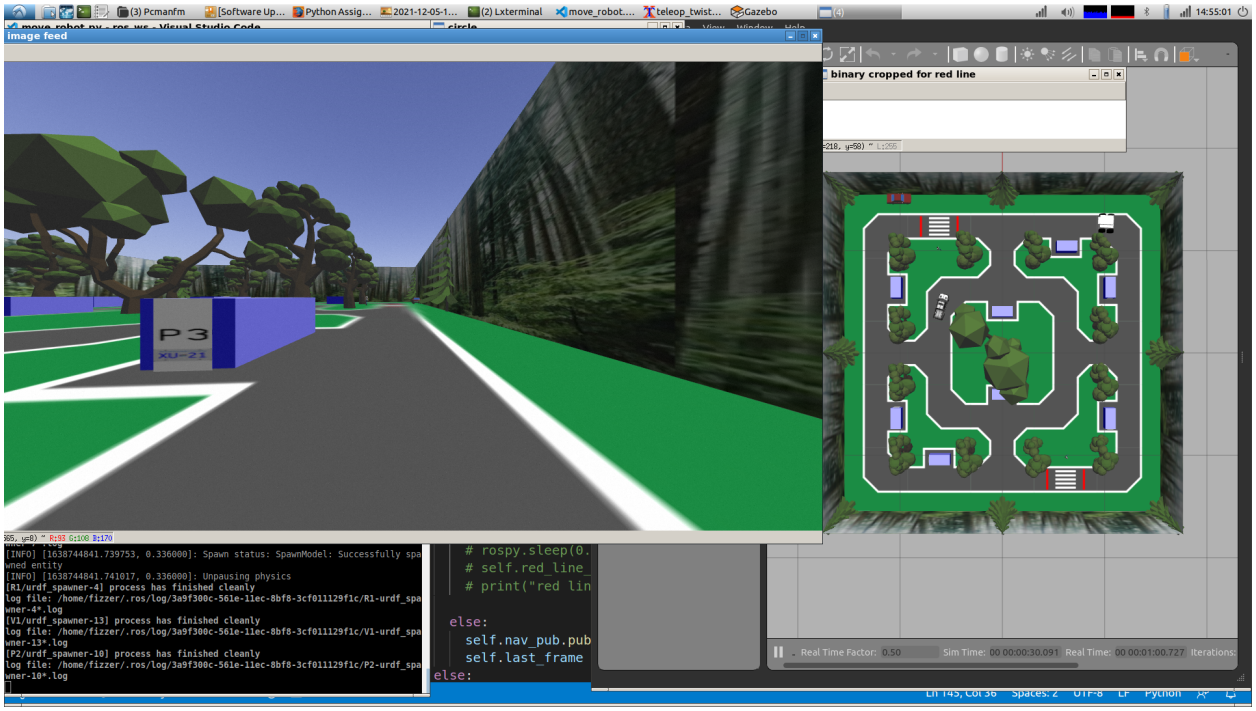

Figure 7: Camera feed showing how the parked cars look in the robot's moving frame.

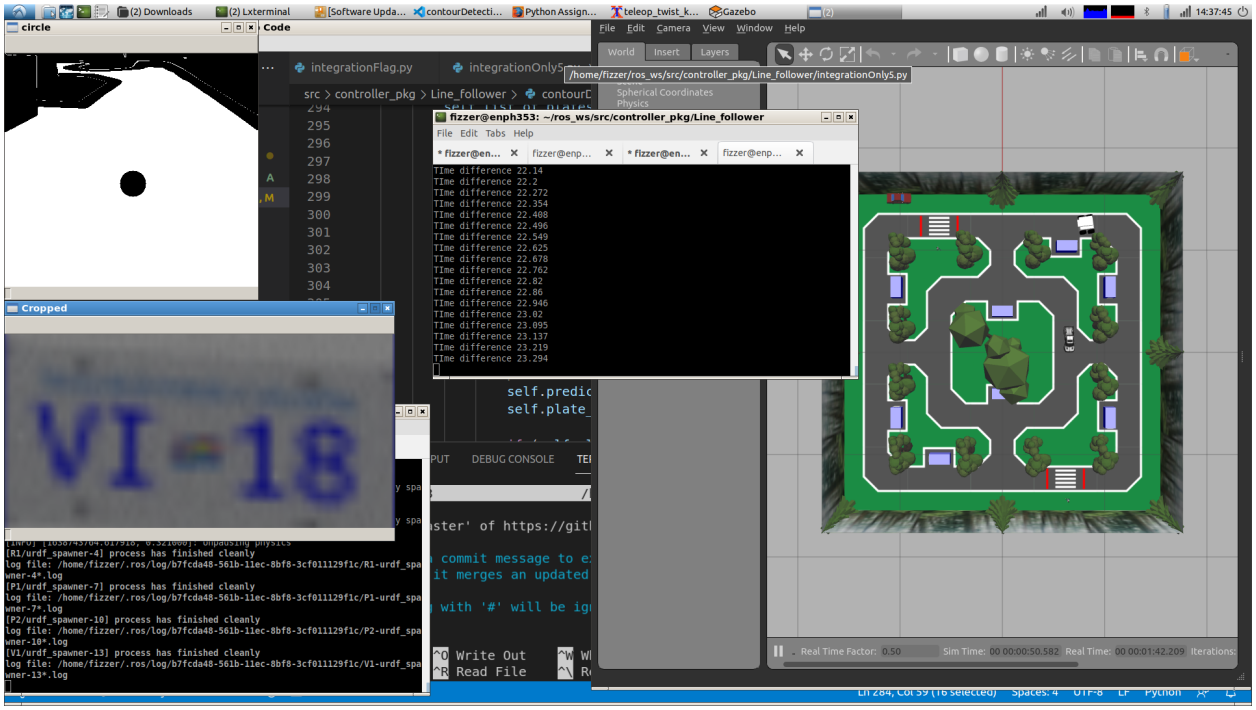

Figure 8: License plate for P3. The robot detected P3 and cropped the image feed in order to extract only the license plate.