ENPH 353 Final Report

Thomas Clarito
Farrandi Hernando

1. Introduction

ENPH 353 is a competition-based, software design course in which competitors develop software for an autonomous robot to detect license plates, parking spots, and obey traffic laws.

The outline of the competition can be found here:
https://docs.google.com/document/d/1gc6Xfro4eQ0ubFueHBWJ0QOLQ_KIGF5G2aT2bnyq9HA/edit

Our strategy after reading the rules was to simply drive around the outside road and read off the 6 cars parked outside. This would get us 41 points (5 from looping and 36 from detecting 6 plates).
Main components of our software:

- Drive
- Pedestrian detection
- License plate detection
- License and Parking spot Reading

2. Drive

For the script used to drive the robot, we used a simple PID algorithm. The main idea is to use the bottom 60 pixels of the robot's camera feed to create a mask for the road.

The error used in the PID algorithm is the distance from the centroid of the road mask to the center of the robot's camera feed. We used this error to calculate our proportional gain and derivative gain, which controlled to control the rotation of the robot.

The initial drive script had the robot moving forward at a constant velocity. This algorithm was passable at high real-time factors, but at lower real-time factors the robot's movement would become very sinuous. To deal with this issue we tried to have a mask at not only the bottom 60 pixels but the middle 60 pixels as well and average the error between the two images. This solution did improve the robot's movement when taking sharp turns, but overall made the movement even more sinuous.

Another solution and more effective solution, was altering the drive code to only move forward when it reaches a specific threshold level (see lines 46-49 in drive.py), and so we would have the robot move forward only when it is aligned well with the road. This made our driving smoother in all regards, although taking sharp turns would take more time. We decided that this would be adequate as taking more time during the turn would also mean that we would have more time to scan for the parking stall after the turn, which we had trouble with previous drive algorithms.

3. Pedestrian Detection:

The pedestrian detection is integrated into the drive script. The main method used to detect the pedestrian is by masking for the pedestrian pants in the middle 60 pixels of the robot’s view. The first algorithm for stopping when the pedestrian was detected ran like this:

- Mask for the pedestrian, mask for the road under the pedestrian in the same 60 pixels.
- If the pedestrian’s centroid was within the road mask plus or minus 20 pixels, the robot would stop. Otherwise it would keep going

The algorithm mentioned above worked pretty well for the most part, but there was one glaring issue. If the pedestrian was not crossing but about to cross, our robot would still continue to move. Although we do not hit the pedestrian, the pedestrian would hit us and in the Gazebo simulated world the pedestrian had a very high inertia. This would cause our robot to be pushed off the road or rotated 180 degrees, completely ruining the run. For this reason we decided to develop another algorithm.

The second algorithm uses a similar idea for detection, we mask for the pedestrian pants. But the algorithm for stopping was different. It goes as follows:

- Mask for the pedestrian, and also mask for the red line in front of the crosswalk. Note the pedestrian mask occurs in the middle 60 pixels of the robot’s camera feed, and the red line mask occurs at the bottom.
- The robot would then stop when it reaches the red line, and wait for the pedestrian to cross. How the robot would tell if it is crossing is by waiting until the pedestrian is within pixels from the center pixel of the robot’s camera feed. This would mean that the robot is in the middle of crossing the street.
- As soon as it detects the pedestrian within the bounds aforementioned, the robot would cross.

We had some initial problems with this code. First, the robot would always stop at the red line, and since there were two red lines at the crosswalk, the robot would stop when it did not need to. This was a simple fix by only stopping when both the red line and the pedestrian were detected. Another minor issue we had was the robot taking a long time before moving, this was due to the bounds of acceptance of when to run were extremely tight initially. In an attempt to solve this we tried another algorithm.

The third algorithm is a combination of both:

- Mask for the pedestrian, the road under the pedestrian, and the red line in front of the crosswalk.
- Check if the pedestrian’s centroid was within the road mask plus or minus 20 pixels, the robot would stop. Otherwise, it would keep moving.
- It would stop at the red line if the pedestrian was detected.
- It would then only move if the the pedestrian is within range as mentioned before in the second algorithm

This algorithm worked ok with high real-time factors, but when we switched to lower real-time factors it became completely unreliable and would just run into the pedestrian at times. For this reason we switched back to the **second algorithm**, which although was slower was very reliable.

4. License Plate Detection

For our license plate detection we went through several iterations and chose the most consistent algorithm. We made several different files for each detection method and we will go through each of that algorithm below.

4.1 First Iteration: Using Feature Matching (SIFT)

Our very first idea for license plate detection was to use feature matching. We planned to use the SIFT algorithm to compare a blank license plate with the camera image that the robot takes. Then we would use the similar points in the camera image then find its homography to perspective transform and form a crop around the “detected license plate” which we will feed to our reading algorithm.

SIFT works by finding key points in the query image, in our case the blank license plate, and the image where we want to find it - the camera image. We will then compare the descriptors of the key points from one picture to another and if it fits a condition we have then we will save those specific key points as “good points”. If the number of good points is above a certain set threshold then it means we would have detected the image and so we can crop then find the homography of the image.

This method did not work for us because it did not detect the blank license plate well, The license plate we want to detect is not empty and has big letters and numbers over it. We think it might be this that caused it to work poorly.

This algorithm can be found our file license_reader.py (last edit was in commit: `929d865`)

4.2 Second Iteration: Using Mask

Our second algorithm is in a new file called reader_mask.py (specifically in the method findPlate). Here we tried to mask the greys of the back of the car. We did this by converting the image from bgr to hsv. Then we used the function cv2.inRange to mask only the license plate. From this masked image we then used the function cv2.cornerHarris to find the edges of the mask and we would save the max and min heights and widths of these edges. Using this info we would then crop out the plate from the original unaltered image to get our license plate.

Some problems we ran into was that the grey colour behind the car would change depending on the car parking in different positions. This is due to the lighting in the simulation which causes the colour to be lighter or darker shades of grey. Another problem was that the roads and other things in the simulation were also grey, so we had to really fine tune our range for masking to do this properly.

This algorithm ended up working better than our first feature matching algorithm, but it still was not reliable. The range of grey that covers all the different shades of all the license plates would also pick up on some roads and other things so sometimes the cropped image we get would not just be the license plate and the parking number. Another problem we ran into was that since we cropped the image from using the max and min widths and heights, if the image was tilted we would still need to perspective transform it.

We thought this algorithm would work so we did several iterations with this so that we would more consistently crop out the license plate:

1. We tried masking the blue of the car instead of the gray on the back of the license plate, but this would cause us problems when we drive past a car and the side of the car (which is all blue) is in view. (commit : `8fdfecd`)
2. We then tried masking for blues first and then gray, which we thought would help us narrow down the grays (not picking up random roads and other things). However, this did not work because there are a lot of blues in the image. Sometimes when we are trying to read a plate from one car, some other car is in view and so the mask would also pick it up. (commit : `63b0caeb`)

Overall, this method was still very inconsistent but it would successfully detect the plates on rare occasions.

4.3 Third Iteration: Edge detection

Most of this code was inspired from https://medium.com/programming-fever/license-plate-recognition-using-opencv-python-7611f85cdd6c. Things to note is that how this algorithm works is by:

1. Making the image to grayscale
2. Using canny for edge detection - changes the image to all lines
3. Look for contours in the image
4. Sort out the contours and filter them to find rectangles

The main problem with this is that there are a lot of rectangles in our image, however this code was more consistent than the others. We found out that since the back of the car was gray and the road was gray, canny sometimes would not draw a line in between the 2 and so there would be no rectangles on the area we want to crop out. This algorithm is in the file: reader_canny.py

4.4 Final Iteration: Mask and contours

We decided to combine our previous iterations and it turned out really well. So in this algorithm we used the masks of our second iteration with the contours in our edge detection. The algorithm is:

1. Converting the image from bgr to hsv
2. Masking out the the greys of the back of the car (note that the grey of the back of the car and the license plate are different shades, and the shade of the license plate is more similar to the road)
3. Using contours to trace the edges of the mask
4. Run through the contours and approximating them to save space and for future filtering

5. Filter out the contours using 2 conditions: having exactly 4 edges and having an area greater than 12 000 unit^2.
6. Then crop out the license plate using the contours, here since our mask actually detects the back of the car, we extended the crop to include the license plate by adding 32 pixels below the 2 bottom points to get a picture of both the parking number (e.g. P1) and the license plate.
7. Arrange our contour points into top left, bottom left, bottom right, top right
8. Step 7 was so that we could perspective transform the image into our desired image size as we would need to feed the CNN pictures of the same size.

This algorithm worked the best. We had to play with the values to crop out the license plate and the minimum area needed. Once we found that, this algorithm would consistently crop out the parking number and license plate from the camera image. This algorithm was what we ended up using in the competition and it is in the file read_using_contours.py (commit :`b1571850`)

5. CNN Architecture:

We used 2 convolutional neural networks in our robot. One CNN to recognize the number for the parking spot, and another CNN to detect the characters in the license plate. Both CNNs use the same structure mentioned below. The CNN architecture is based on the previous labs and the following paper: http://cs231n.stanford.edu/reports/2015/pdfs/vikesh_final.pdf.

The structure is as follows:

- A 2D convolutional layer with 32 filters, kernel size of 3x3, and ReLu activation.
    - This layer detects the edges/transitions in the image.
- A Max Pooling layer with a kernel size of 2x2
    - This layer keeps only the most significant features in the image
- Another 2D convolutional layer with 64 filters, kernel size of 3x3, and ReLu activation
- Another Max Pooling layer with kernel size 2x2
- A Flatten layer
    - This squeezes the processed image in to a 1D array
- A Dropout layer, with rate 0.5
    - This randomly sets input values to 0 at a given rate
- A Dense layer, with ReLu activation and size 32.
    - This layer outputs the element wise activation of the input and a weighted matrix determined by the layer. The output’s size is the given input size.
- Another dense layer, with Softmax activation.

Initially, we only had one 2D convolutional layer and one max pooling layer, but we noticed that our CNN had trouble differentiating certain letters and numbers from each other during testing. For example, we had problems with 7 and T, S and 5, G and 0. For this reason we added another 2D convolutional layer with more filters and another max pooling layer. This would help us better detect the edges and transitions in the images.

6. Training the CNN

We had 2 CNNs to train but the process we did was very similar. We first needed to have every single letter of the alphabet from the license plate. To generate our dataset, we initially made 500 random license plates from a blank plate template, and cropped each character to create a dataset of 2000 images. We soon realized that this would not be an appropriate dataset for the competition given that our dataset would not match the images we get from the simulated environment.

We then generated data by running our drive and license plate detection algorithm around several times and saving the license plates. Then using these license plates we would crop out the parking spaces and save them to a dataset for parking, and also crop out each letter and alphabet from the license plate and do the same.

From this we would get every letter and alphabet needed and so we manually cropped out each letter slightly differently to get around 1600 random data (300 of which was used for validation). The pictures were also masked so that it would be black with only the letter/number being white. We ran it through our CNN several times to see the graphs of value loss in our validation data set, and we decided that 4 epochs was enough for our parking CNN and 8 epochs were enough for our license plate generator.

During the competition, our parking CNN ended up correctly identifying each license plate. However, our license reading CNN failed to do so. After consulting with our instructor, we found out that it was due to our dataset. We should have used a keras function ImageDataGenerator to generate the images and it would have made our CNN more robust.