| | |
|---|---|
| **ECE 498SMA: Principles of safe autonomy** | Spring 2020 |
| *Written by: Ted Liu, Qichao Gao, Sayan Mitra Course Website* | *MP 1: Lane detection* |
| | *Due Date: Feb. 7* |

# 1 Introduction

In this MP, you will implement a camera-based lane detection module. The module will take a video stream in a format that is used in autonomous systems as input, and it will produce annotated video with lane area. We will split the video into frames, and process it as an image in ROS(Robot Operating System). We will finish the functions shown in Figure 1, one by one, to create the lane detection module.

To start, you need to download the code from a gitlab repo . All the code you need to change should be inside the file `studentVision.py` and `line_fit.py` and you will have to write a brief report. You will have to use functions from the Robot Operating System (ROS) [1] and OpenCV library [**?**]. This document gives you the first steps to get started on implementing the lane detection module. You will have to take advantage of tutorials and documentations available online. Cite all resources in your report. **All the regulations for academic integrity and plagiarism spelled out in the student code apply.**

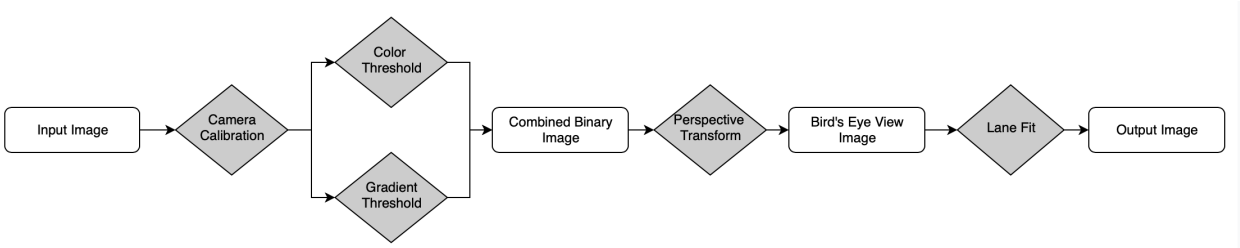


Figure 1: Lane detection module overview.

**Learning objectives**

- Working with rosbags
- Basic computer vision
- Working with OpenCV libraries

**System requirements**

- Ubuntu 16
- ROS Kinetic

## 2 Module architecture

The building-block functions of a typical lane detection module are shown in Figure 1 and also listed below. However, in this MP you only need to implement some of the functions. The functions marked by * are not *required* for you to implement, but you can experiment with them. Lane detection is a hard problem with lots of ongoing research. This could be a component to explore more deeply in your project.

**Camera calibration*** Since all camera lenses will introduce some level of distortions in images, camera calibration is needed before we starting process the image.

**Perspective transform** Convert the image into *Bird's Eye View*.

**Color threshold** Convert the image from RGB space to HLS space, and threshold the S channel to find lane pixels.

**Gradient threshold** Run an edge detection on the image by applying a Sobel filter.

**Combined binary image** Combine the gradient threshold and color threshold image to get lane image. We suggest to apply a *Region of Interest* mask to get rid of irrelevant background pixels and apply morphology function to remove noise.

**Lane fitting** Extract the coordinates of the centers of right and left lane from binary *Bird's Eye View* image. Fit the coordinates into two second order polynomials that represent right and left line.

## 3 Development instructions

### 3.1 Start ROS

Robot Operating System(ROS) [1] is a robotics middleware that significantly simplifies software development for robotics and embedded systems. Several autonomous driving companies use ROS or some modified version thereof, in their products. We will have a more thorough introduction to ROS in future lectures.

All 7 machines in our lab (ECEB5072) have ROS installed in root directory, so there is no need to install it again. However, when a user logs in to a machine the first time, he or she needs to set up some environment variables. Copy and run the following two lines in the command console (Application->System Tools->MATE Terminal).

```
echo "source␣/opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Then before we do anything on ROS, we need to start the ROS master node:

```
roscore
```

### 3.2 Reading input from rosbag file

Messages with images, sensor data, etc., in ROS [1] can be recorded in a *rosbag file*. This is convenient as the rosbags can be later replayed for testing and debugging the software modules we will create. Within a rosbag, the data from different sources is organized by different *rostopics*.

For this MP, our input images are published by the rostopic *camera/image_raw*. We can play it back using the commands `rosbag info` and `rosbag play`.

We have stored the rosbags the root directory on each machine. You can navigate to the folder by:

```
cd ~/../../var/www/ECE498SM/bag/
```

Inside the folder there are two ROS bag files. First, execute the following command from the bag files directory to see the type of contents in the rosbag.

```
rosbag info <your bagfile>
```

You should see something like:

```
path:        0011_sync.bag
version:     2.0
duration:    7.3s
start:       Dec 31 1969 18:00:00.00 (0.00)
end:         Dec 31 1969 18:00:07.30 (7.30)
size:        98.6 MB
messages:    74
compression: none [74/74 chunks]
types:       sensor_msgs/Image [060021388200f6f0f447d0fcd9c64743]
topics:      camera/image_raw    74 msgs    : sensor_msgs/Image
```

This tells us topic names and types as well as the number (count) of each message topic contained in the bag file.

The next step is to replay the bag file. In a terminal window run the following command in the directory where the original bag files are stored:

```
rosbag play -l <your bagfile>
```

When playing the rosbag, the video will be converted to individual images that are sent out at a certian frequency. We have provide a callback function. Every time a new image is published, the callback function will be triggered, convert the image from ROS message format to OpenCV format and pass the image into our pipeline.

### 3.3 Gradient threshold

In this section, we will implement a function which uses gradient threshold to detect interesting features (e.g. lanes) in the image. The input image comes from callback function. The output shall be a binary image that highlights the edges.

```
def gradient_threshold(self, img, thresh_min= 25, thresh_max= 100):
    """
    Apply sobel edge detection on input image in x, y direction
    """
    return binary_output
```

First we need to convert the colored image into grey scale image. For a gray scale image, each pixel is represented by a uint8 number (0 to 255). The image gradient could emphasize the edges easily and hence segregate the objects in the image from the background.

Figure 2: Image gradient. *Left:* Original image. *Right:* Gradient image.

Then we need to get the gradient on both on x axis and y axis. Recall that we can use the Sobel operator to approximate the first order derivative. The gradient is the result of a 2 dimensional convolution between Sobel operators and original image.

$$\nabla(f) = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}] \tag{1}$$

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \; S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2}$$

$$G_x = S_x * I, \; G_y = S_y * I \tag{3}$$

Finally we need to convert each pixel into unit8, then apply the threshold to get binary image. If a pixel has value between minimum and maximum threshold, we set the value of that pixel to be 1. Otherwise set it to 0. The resulting binary image will be combined with the result from color threshold function and passed to perspective transform.

### 3.4 Color threshold

In this section, we will implement a function which uses color threshold to detect interesting features (e.g. lanes) in the image. The input image comes from callback function. The output shall be a binary image that highlights the white and yellow color.

```
```

```
def color_threshold(self, img, thresh=(100, 255)):
    """
    Convert RGB to HSL and threshold to binary image using S channel
    """

    return binary_output
```

Besides gradient method, we can also utilize the information that lane markings in United States are normally white or yellow. You can just filter out the white and yellow pixels and they are likely to be on the lanes. You may use different color space(RGB, LAB, HSV, HSL, YUV...), different channels and compare the effect. Some of use prefer the RGB and HSL in the lane detection. Feel free to explore other color spaces.

You need to first convert the input image from callback function into the desired color space. Then apply threshold on certain channels to filter out pixels in white and yellow. In the final binary image, those pixels shall be set to 1 and the rest pixels shall be set to 0.

Figure 3: Color threshold. *Left:* Original image. *Right:* Color threshold image.

### 3.5 Perspective transform

In this section, we will implement a function which converts the image to *Bird's eye view* (looking down from the top). This will give a geometric representation of the lanes on the 2D plane and the relative position and heading of the vehicle between those lanes. This view is much more useful for controlling the vehicle, than the first-person view. In the *Bird's eye view*, the distance of pixels on the image will be proportional to the actual distance, thus simplifies calculations in control modules in the future labs.

The input comes from combined binary image from color and gradient threshold. The output is the image converted to *Bird's eye view*. M and Minv are transformation matrix and inverse transformation matrix. To get them, you need to choose 4 points on original image and map the same 4 points on *Bird's eye view* image. Then pass those coordinates into cv2.getPerspectiveTransform to get M and Minv.

```
def perspective_transform(self, img, verbose=False):
    """
    Get bird's eye view from input image
    """

    return warped_img, M, Minv
```

During the perspective transform we wish to preserves collinearity (i.e., all points lying on a line initially still lie on a line after transformation) and ratios of distances (e.g., the midpoint of a line segment remains the midpoint after transformation). Such transformation is also called Affine Transformation, which requires a 3-by-3 transformation matrix.

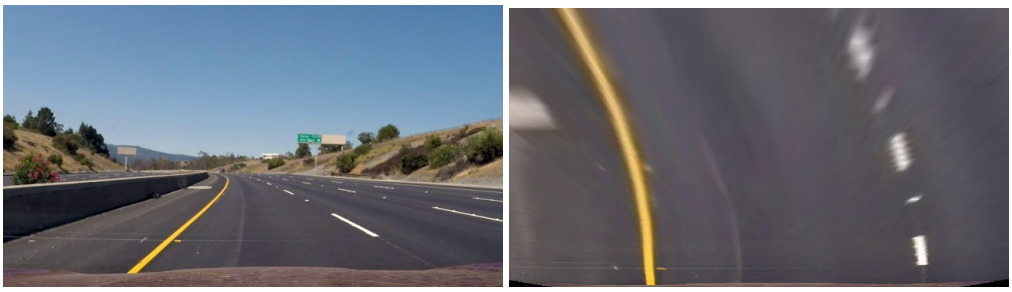

Figure 4: Perspective transformation. *Left:* Original image. *Right:* Bird's eye view image.

Here $(x, y)$ and $(u, v)$ are the coordinates of the same point in the coordinate systems of the original perspective and new perspective.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4}$$

To find the matrix, we need to find the location of 4 points on the original image and map the same 4 points on the *Bird's Eye View*. Any 3 of those 4 points should not be on the same line. Put those 2 groups of points into cv2.getPerspectiveTransform(), the output will be the transformation matrix. Pass the matrix into cv2.warpPerspective() and we will have the warped *Bird's Eye View* image.

### 3.6 Lane fitting

From the previous step, we get binary *bird's eye view* images that separates the possible lane pixels and background. We cut the image horizontally into several layers, and try to find the lane center on each of the layers. The histogram method is used in this step. We calculate the histogram of pixels in left half and right half of that horizontal layer. The place where possible pixels are most dense will be treated as centroid of each lane. Finally fit the coordinates of those center of lanes to second order polynomial using np.polyfit() function. Enhancement can be done by taking the average of N most recent frame polynomial coefficients.

### 3.7 Test results

When code is finished, just run:

```
python studentVision.py
```

Rviz is a tool that help us visualize the ROS messages of both the input and output of our module. It is included by default when you install ROS. To start it just run:

```
rviz
```

Then a window should pop out like in Figure 5.

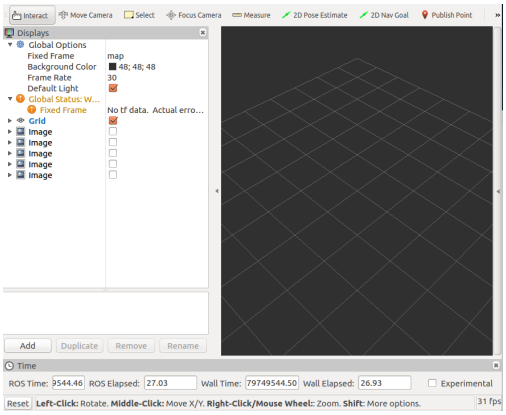


Figure 5: Rviz Window

You can find a "add" button on lower left corner. Click add -> By Topic. From the list of topics, choose "/lane detection -> /annotated image -> /Image" (in Figure 6). A small window should pop out that display the right and left line have been overlaid on the original video. Repeat the procedure to display "/lane detection -> /Birdseye -> /Image". The result should be like in Figure 7.

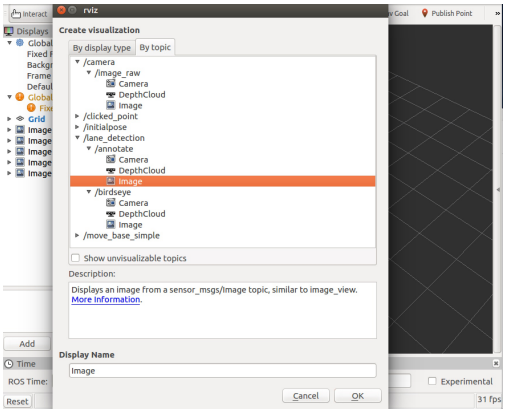


Figure 6: Choose Topics

# 4 Report

Each group should upload a short report (2 pages). First, all group members' names and netIds should be listed on the first page. Then, discuss the the following questions possibly with screenshots:

1. What are the interesting design decisions you considered and executed, in creating the different functions in your lane detection modules?

2. What is each member's contribution and the number of hours spent (this does not have to be equal)? Give a rough breakdown of the time spent.

3. Any feedback you have for SafeAutonomy498-team on the Lab and the MP?

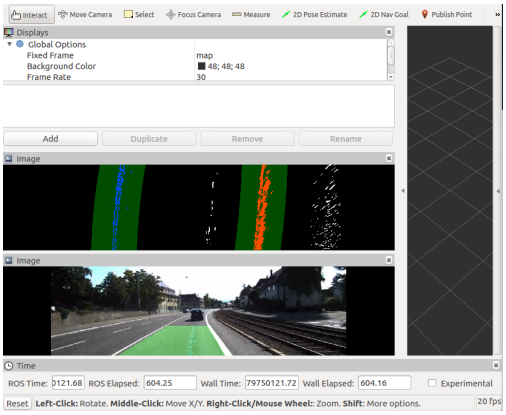


Figure 7: Result in Rviz

# 5 Submission instructions

Before the deadline, students need to take some short videos clearly prove that their code works on both videos. You can either use some screen recording software or just your smart phone. The videos and report only need to be submitted to Compass 2G by one of the members in your group (homework need to be submitted individually). Then in the following week, each group should come to one of the office hours and demo the code to the TAs. All members should show up and be prepared to answer some questions. The TA might pick a random person and ask that person to answer the question. So all students should be familiar with the content.

# 6 Grading rubric

- 25%: Video is submitted on Compass 2G quality of results on old and new tests
- 30%: The Rviz shows a completed annotated video during the demo
- 20%: TAs questions are answered properly during the demo.
- 25%: Report

# References

[1] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.