Code a Car Camp Summer 2022

Written by: ECE484 and Code A Car camp team *MP1: Lane detection*

Website

1 Introduction

In this assignment, you will apply computer vision techniques for lane detection. 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 the lane area marked. We will split the video into frames, and process it as an image in ROS (Robot Operating System). You will finish a subset of the functions shown in Figure 3, one by one, to create the lane detection module. The functions that you will have to implement are in the file `studentVision.py`.

Learning objectives

- Edge detection using gradients
- Working with rosbags, Gazebo and Rviz
- Basic computer vision
- Working with OpenCV libraries

System requirements

- Ubuntu 20.04
- ROS Noetic

2 Implementation

2.1 Problem Statement

For the implementation, you are going to implement a lane detection algorithm that will be applied to 2 scenarios. In scenario 1, you will work with a GEM car model equipped with camera in Gazebo. The GEM car will be moving along a track.

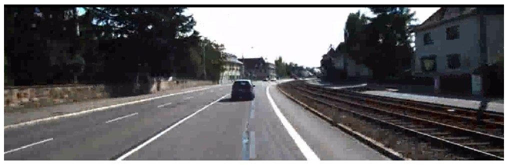


Figure 1: Video recorded on a real car, resolution 1242×375

In scenario 2, which is an extension activity, will run your lane detection algorithm on a video recorded on a real car moving on a high way.

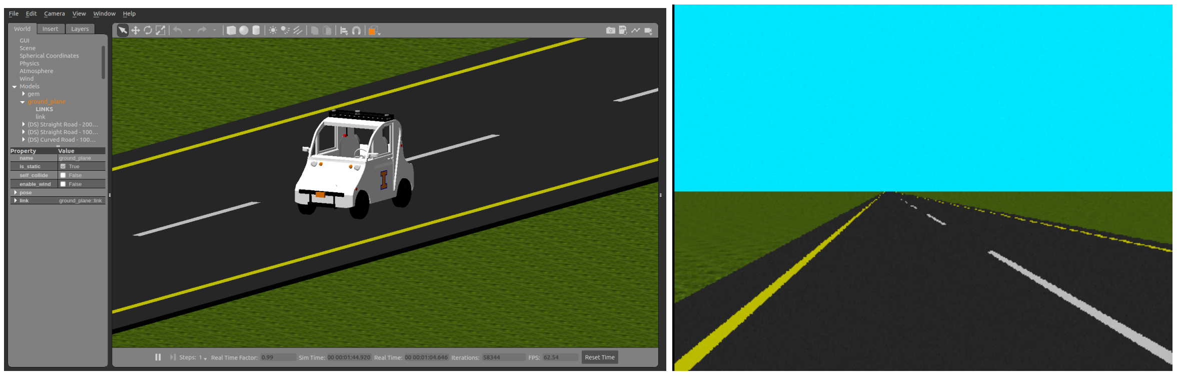


Figure 2: GEM car and its camera view in Gazebo

For both scenarios, your task is to use your algorithm to detect the lanes in the video or camera output. Please note that several parameters (e.g. color channels, threshold of Sobel filter, points for perspective transform) in the 2 scenarios are different, and you will need to figure them out separately.

2.2 Module Architecture

The building-block functions of a typical lane detection pipeline are shown in Figure 3 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.

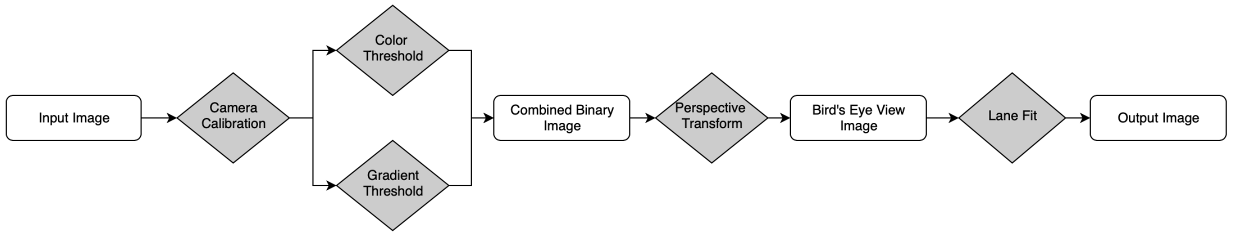


Figure 3: Lane detection module overview.

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

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

Color threshold Threshold on color channels to find lane pixels. A typical way to do this is to covert the image from RGB space to HLS space and threshold the S channel. There are also other ways to do this part.

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. You should also 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 lane.

3 Development Instructions

The functions you will have to modify are located in `studentVision.py` which is located in the following path:

```
cd ~/Desktop/CodeACar22/src/mp1/src
```

3.1 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_thresh(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.

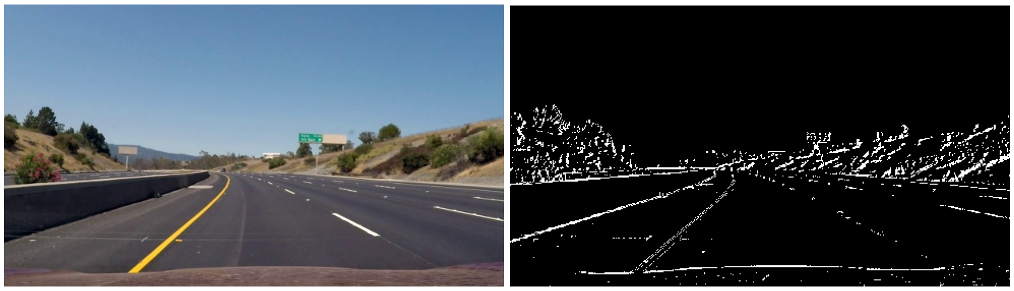


Figure 4: Image gradient. *Left:* Original image. *Right:* Gradient image.

Then we need to get the gradient on both 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. For this part, you will find *Sobel* function in OpenCV useful.

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.

3.2 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.

We pass a boolean flag, gem. You should implement your code such that when the gem flag is true, the color threshold parameters are used for the GEM environment. Likewise, when the gem flag is false, the color threshold parameters are used for the real world (which will be completed in the extension activity).

```
def color_thresh(img, gem, 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 pixels other than white and yellow and what's left 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 people prefer the RGB and HSL in the lane detection. Feel free to explore other color spaces.

HSL and HSV is another representation of the RGB color model, which matches human perception better. The color space is as follows:

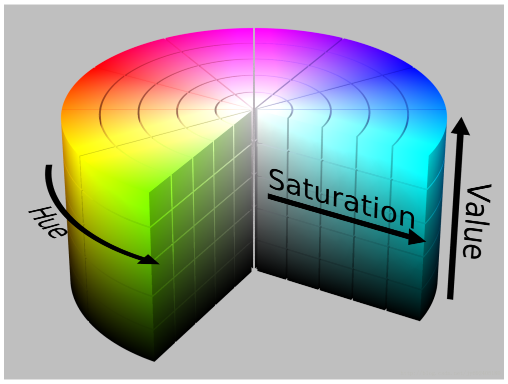


Figure 5: HSV Color Space

The H for both spaces is the set up around the circle. We can utilize this by taking a threshold between two different hues to extract all values of a specific color. For example, if you threshold H value between 0 and 12, you will get all colors between these two hues:

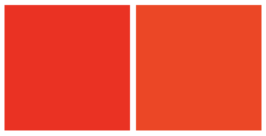


Figure 6: H Thresholding. *Left:* H: 0. *Right:* H: 12.

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 the final binary image, those pixels shall be set to 1 and the rest pixels shall be set to 0.

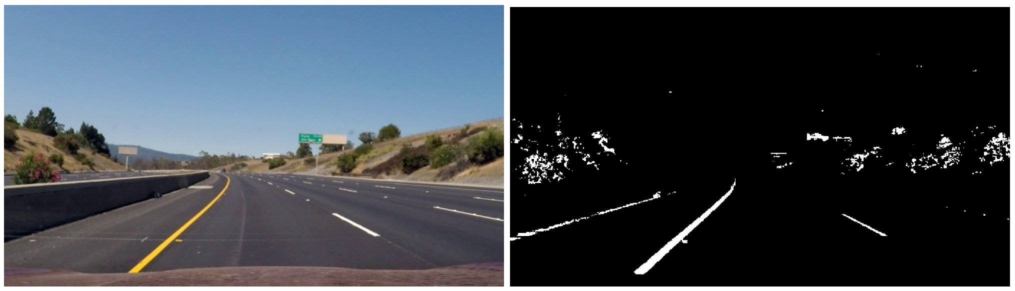


Figure 7: Color threshold. *Left:* Original image. *Right:* Color threshold image.

3.3 Combined binary image

In this section, we will combine the output of our *gradient_threshold* and *color_threshold* functions to create the final image that will be used in our perspective transform and lane fit modules. You may have to utilize bitwise operations for this function. Each pixel in your output image should only be a 1 if the respective pixel in the *gradient_threshold* image and *color_threshold* is also a 1.

```
def combinedBinaryImage(img, gem):
    """
    Get combined binary image from color filter and sobel filter
    """
    return binary_output
```

4 Testing Lane Detection

4.1 GEM Model in Gazebo

For this MP, we provide a simulation environment in Gazebo where a GEM car equipped with a camera is moving along a race track.

First, open the */Desktop/CodeACar22/user.txt* file and on the first line, put the team name. You can leave the second line as it is.

To compile and source the package, you need to open a new terminal and execute the following commands:

```
source /opt/ros/noetic/setup.bash
cd ~/Desktop/CodeACar22/
catkin_make
source devel/setup.bash
```

To launch Gazebo, you need to execute the following command in the same terminal:

```
roslaunch  mp1  mp1.launch
```

Two windows should open. There will be a window in Gazebo with the map rendered. You can minimize that window. There will also be an Rviz window. Rviz is a tool that helps us visualize the ROS messages of both the input and output of our module. This window will allow us to evaluate our lane detection pipeline.

To reset the vehicle to its original position and to drive the vehicle along the track, open a new terminal and execute python script:

```
source /opt/ros/noetic/setup.bash
source ~/Desktop/CodeACar22/devel/setup.bash
rosrun  mp1 main.py
```

To start your lane detection pipeline, open a new terminal and execute python script:

```
source /opt/ros/noetic/setup.bash
source ~/Desktop/CodeACar22/devel/setup.bash
rosrun mp1 lane_net_detector.py --gem
```

You can also pass the *–plot* flag as the following:

```
rosrun mp1 lane_net_detector.py --gem --plot
```

This will create a plot showing you the running average variance in the distance between the two lanes and the current variance. The goal is to make it as small as possible.

Both commands will also generate a video of the run which should be in the *CodeACar22/src/mp1/src* directory.

At the end of each run, you should run the *main.py* and *lane_net_detector.py* again. You don't need to *source* again.

4.2 Submission

For this mp, you are rewarded for having a consistent lane detection algorithm. The metric that will be used to evalute your algorithm will be the variance of the width of the lane detected.

When you are ready to submit to the leaderboard, run the *main.py* command, and then add the *upload* flag to the *lane_net_detector.py* command as the following:

```
rosrun mp1 lane_net_detector.py --gem --upload
```

4.3 Extension Activity: Video using rosbag

In this activity, you will run your lane detection algorithm on recorded videos from a real car. To do this, you will need to go back an modify your *color_thresh* function and add the the color threshold parameters are used for the real world.

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*.

Before we do anything on *rosbag*, we need to start the ROS master node. Open a new terminal and execute the following commands:

```
source /opt/ros/noetic/setup.bash
roscore
```

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 3 `rosbag` in the bags directory: `0011_sync.bag`, `0056_sync.bag`, and `0484_sync.bag`. Open a new terminal and navigate to the directory by:

```
source /opt/ros/noetic/setup.bash
cd ~/Desktop/CodeACar22/src/mp1/src/bags
```

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 into individual images that are sent out at a certian frequency. We have provided 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.

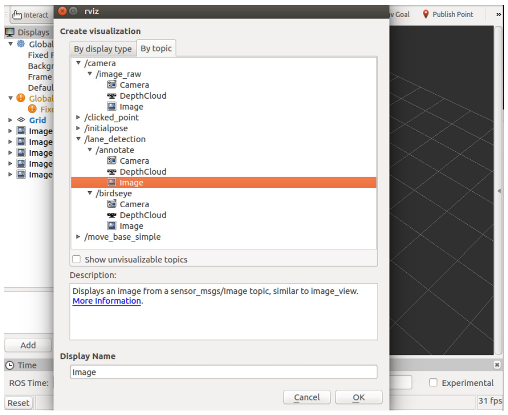
Now, to run our lane detection pipeline, execute the following commands in a new terminal:

```
source /opt/ros/noetic/setup.bash
rosrun  mp1 lane_net_detector.py
```

To analyze the result, you can use Rviz. In a new terminal, execute the following commands:

```
source /opt/ros/noetic/setup.bash
rviz
```

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 8). 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 9.



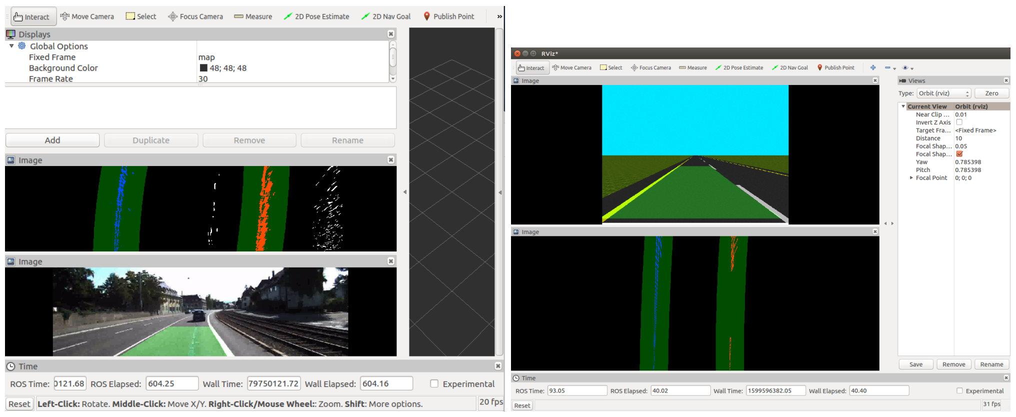
Displays an image from a sensor_msgs/Image topic, similar to image_view. More Information.

Display Name: Image

Figure 8: Choose Topics

References

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



Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30

ROS Time: 0121.68 ROS Elapsed: 604.25 Wall Time: 79750121.72 Wall Elapsed: 604.16 Experimental

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options. 20 fps

Type: Orbit (rviz)

Current View	Orbit (rviz)
Near Clip ...	0.01
Invert Z Axis	☐
Target Fra...	<Fixed Frame>
Distance	10
Focal Shap...	0.05
Focal Shap...	☑
Yaw	0.785398
Pitch	0.785398
Focal Point	0; 0; 0

ROS Time: 93.05 ROS Elapsed: 40.02 Wall Time: 1599596382.05 Wall Elapsed: 40.40 Experimental

31 fps

Figure 9: Result in Rviz; left for rosbag scenario, right for GEM car scenario