

## 1 Introduction

In this MP, you will apply computer vision techniques for lane detection. Similar to MP0, MP1 is also divided in 2 parts. In first part, you will answer 2 written questions related to concepts in computer vision. Then, 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's marked. We will split the video into frames, and process it as an image in ROS (Robot Operating System). You will finish the functions shown in Figure 3, one by one, to create the lane detection module.

For part 1, you will need to finish them individually and submit solutions in one file `mp1_1_netid.pdf`

For part 2, all your code should be in the file `studentVision.py` and `line_fit.py`, and you will have to write a brief report `mp1_2_groupnumber.pdf`. You will have to use functions from the Robot Operating System (ROS) [2] and you are encouraged to use the OpenCV library [1]. 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.**

### Learning objectives

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

### System requirements

- Ubuntu 16
- ROS Kinetic

## 2 Written Problems

These questions are related to course materials covered in lectures. Each member of the group will need to submit answers to these questions individually.

**Problem 1 Convolution (5 points)** Prove that convolution is commutative. That is given two functions (or arrays) show that  $f * g = g * f$ . You may consider the functions to be one or two dimensional. State your assumptions clearly.

**Problem 2 Filtering (10 points)** Write down a  $5 \times 5$  Gaussian derivative kernel in the  $x$ -direction. Does it find vertical or horizontal edges? Is this filter separable? Explain your answer.

### 3 Implementation

#### 3.1 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 are given with a video recorded on a real car moving on a high way.



Figure 1: Video recorded on a real car, resolution  $1242 \times 375$

In scenario 2, you will work with a GEM car model equipped with camera in Gazebo. The GEM car will be moving along a track.

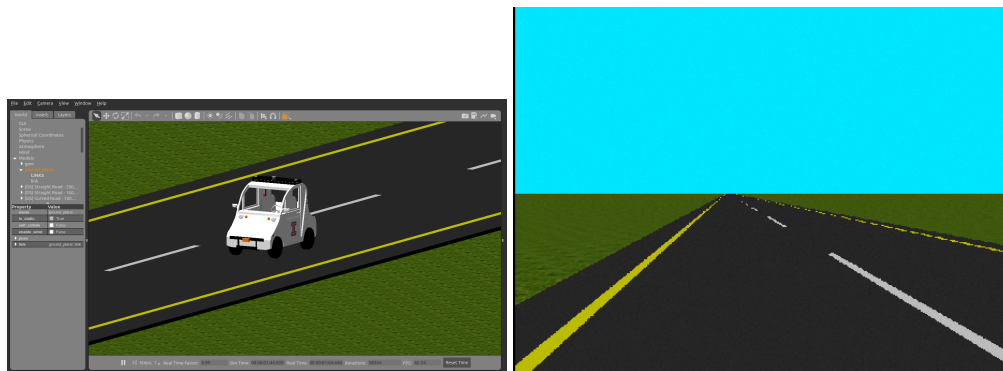


Figure 2: GEM car and its camera view in Gazebo

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

#### 3.2 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.

**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*.

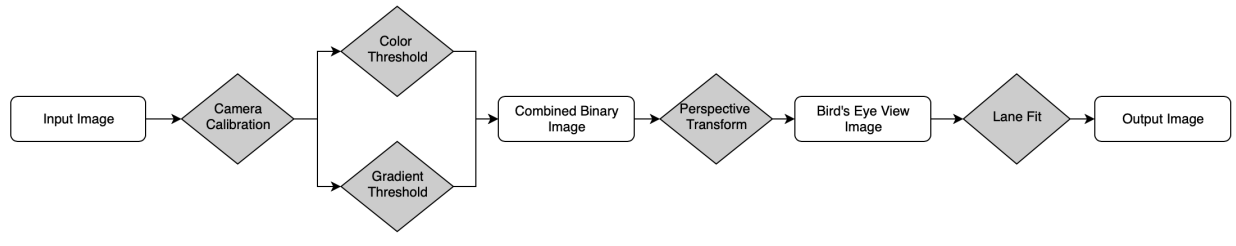


Figure 3: Lane detection module overview.

**Color threshold** Threshold on color channels to find lane pixels. A typical way to do this is to convert 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. 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.

### 3.3 Report

Each group should upload a short report (2 pages) with following questions answered

**Problem 3 (15 points)** What are some interesting design choices you considered and executed in creating the different functions in your lane detection modules? E.g. which color spaces do you use? how do you determine the source points and destination points for perspective transform?

**Problem 4 (25 points)** In order to detect the lanes in both scenarios, you will need to modify some parameters. Please list and compare all parameters you have to modify and explain why do you think altering them will be helpful?

**Problem 5 (40 points)** Record 2-3 short videos of Rviz window and Gazebo to show that your code works for both scenarios. You can either use screen recording software or smart phone to record. Also, report number of times the program prints "unable to detect lanes".

**Problem 6 (5 points)** Every member in the group is supposed to contribute roughly equally to every MP. Here you have to give an assessment of how much time each member of your group devoted to this MP each week in the following table format. Equal contribution does not necessarily mean equal number of hours. Give a rough breakdown of the time spent.

	Name 1	Hours 1	Name 2	Hours 2	Name 3	Hours 3
Week 1						
Week 2						

**Optional** Any feedback you have for SafeAutonomy498-team on the Lab and the MP?

## 4 Development Instructions

All the python script you will need to modify are `studentVision.py` and `line_fit.py` which are located in the following path:

```
cd [path-to-MPI-workspace]/src/gem_simulator/gem_gazebo/src
```

### 4.1 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_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 4: 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. For this part, you will find *Sobel* function in OpenCV to be useful.

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

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (2)$$

$$G_x = S_x * I, G_y = S_y * I \quad (3)$$

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.

## 4.2 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.

```
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 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.

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 5: Color threshold. *Left:* Original image. *Right:* Color threshold image.

## 4.3 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. You don't need to worry about them.

```
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 preserve 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.

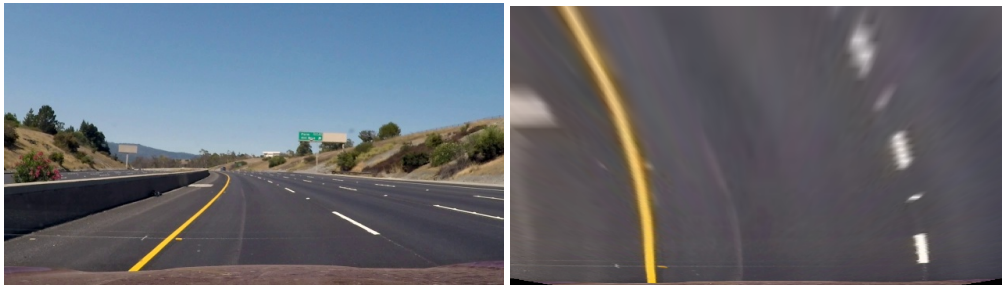


Figure 6: Perspective transformation. *Left*: Original image. *Right*: Bird's eye view image.

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

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

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.

#### 4.4 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. If the lane fit fails, meaning the polynomials are unable to be solved, the program will print error message: *unable to detect lanes*.

## 5 Testing Lane Detection

### 5.1 Video using rosbag

Messages with images, sensor data, etc., in ROS [2] 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:

```
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 2 rosbag the src directory: `0011_sync.bag` and `0056_sync.bag`. You can navigate to the directory by:

```
cd [path-to-MP1-workspace]/src/gem_simulator/gem_gazebo/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 `rosviz`, the video will be converted to individual images that are sent out at a certain 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.

## 5.2 GEM Model in Gazebo

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

Before launching GEM car in Gazebo, there are some dependencies you will need to install. You can do this by type following command in terminal:

```
sudo apt-get install ros-kinetic-effort-controllers
sudo apt-get install ros-kinetic-joint-state-controller
sudo apt-get install ros-kinetic-position-controllers
```

Compile and source the package

```
cd mp1-release-20fa
catkin_make
source devel/setup.bash
```

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

```
roslaunch gem_gazebo gem_vehicle.launch
```

To drive the vehicle along the track, execute python script:

```
python drive.py
```

To reset the vehicle to original position, execute python script:

```
python reset.py
```

## 5.3 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

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 7). 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 8.

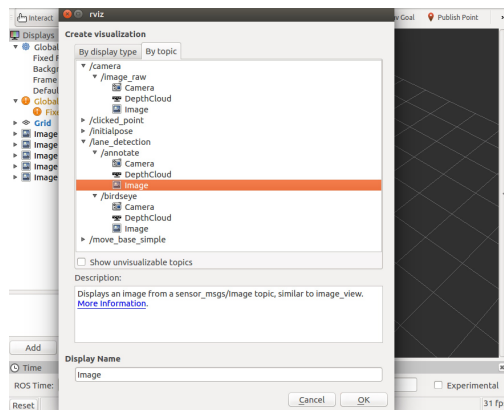


Figure 7: Choose Topics

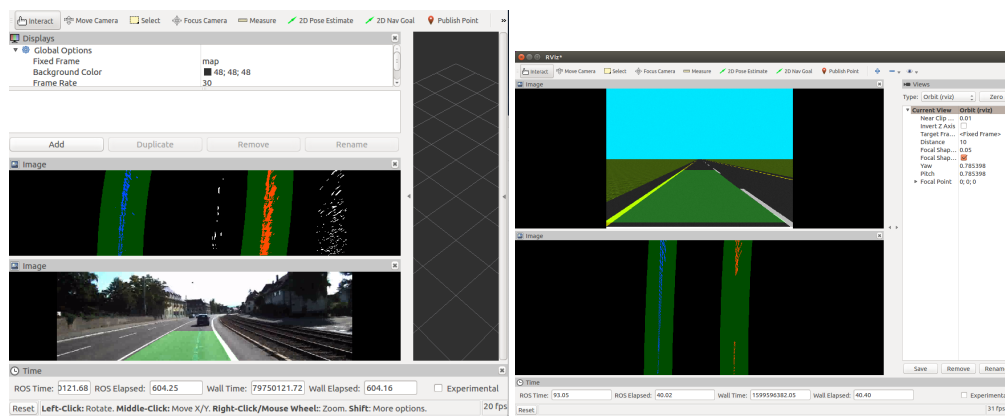


Figure 8: Result in Rviz; left for roscam scenario, right for GEM car scenario

## 6 Submission instructions

For the individual written problems 1-2, each student needs to submit their own solution in a file `mp1_1_netid.pdf`. This should be done **individually**, but you may discuss the problems following the tenets of academic integrity and collaboration. For the group project report (problems 3-6), students need to take 2-3 short videos clearly prove that their code works on both scenarios (2 `rosbag` videos and simulator environment) and answer other problems. Note that, for the simulation environment, your video must show that your implementation can detect lanes when car is moving. The videos can be uploaded to a website (YouTube, Vimeo, or Google Drive with public access) and you need to include the link in the report. One member of each team will upload the report `mp1_2_groupnumber.pdf`.

Both documents need to be uploaded to Gradescope. Also, please upload your code to Compass2G. Please only submit the `src` folder in a `.zip` file that contains code excluding `rosbags`. Rename the the `.zip` to `mp1_2_groupnumber.zip`

## References

- [1] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
- [2] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.