LAB Report Template

This is an example report template to help students write a concise and organized report. But you do not need to follow the exact format of this template, as long as you write a good quality report.

LAB: Lane Detection

Date: 2025-Apr-7

Author: Yechan Kim 22100153

Introduction

1. Objective

Goal: The purpose is to detect lanes that are identified in front while driving.

This task aims to detect lanes in two cases: when the car is in the center of the lane and when changing lanes.

2. Preparation

Software Installation

- OpenCV 4.9.0, Visual Studio 2022

Dataset

Dataset link:

Dataset 1

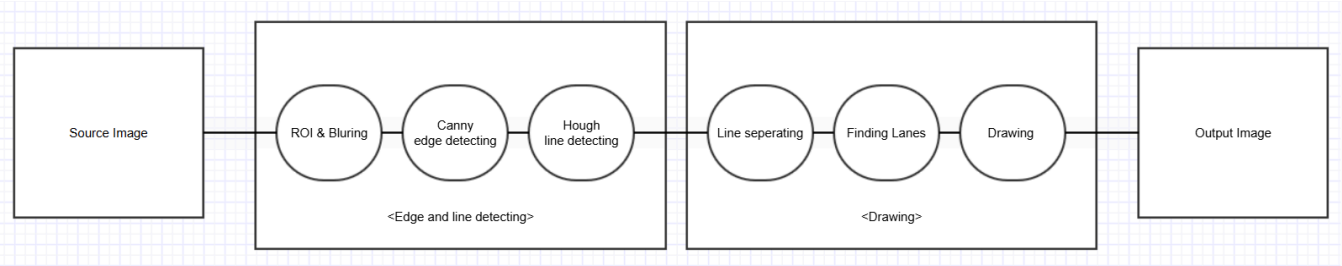
Dataset 2

Algorithm

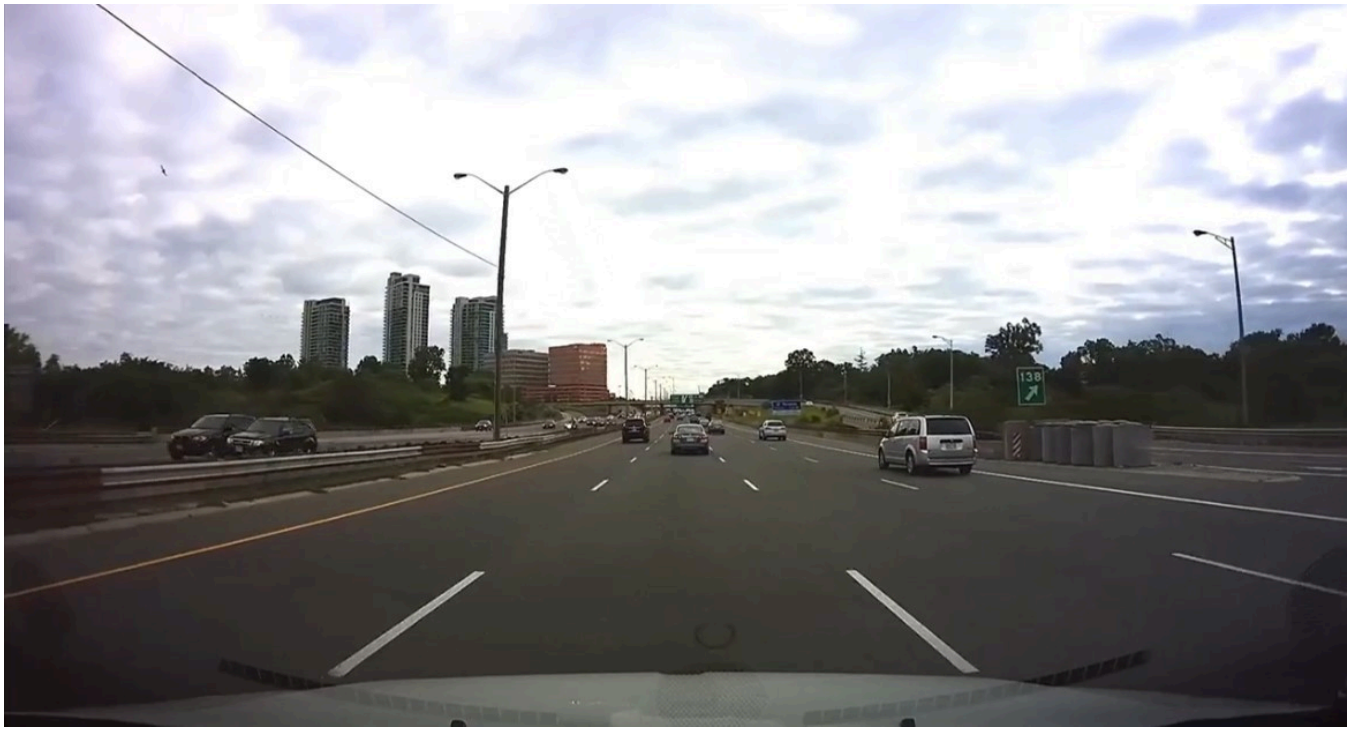
1. Overview

The flow of the entire algorithm is to detect the final lane by applying the HoughLinesP function after applying canny edge detection to the original image.

The original image is as follows:



Being on the center of the Lane



Changing the lane

2. Procedure

ROI & Bluring

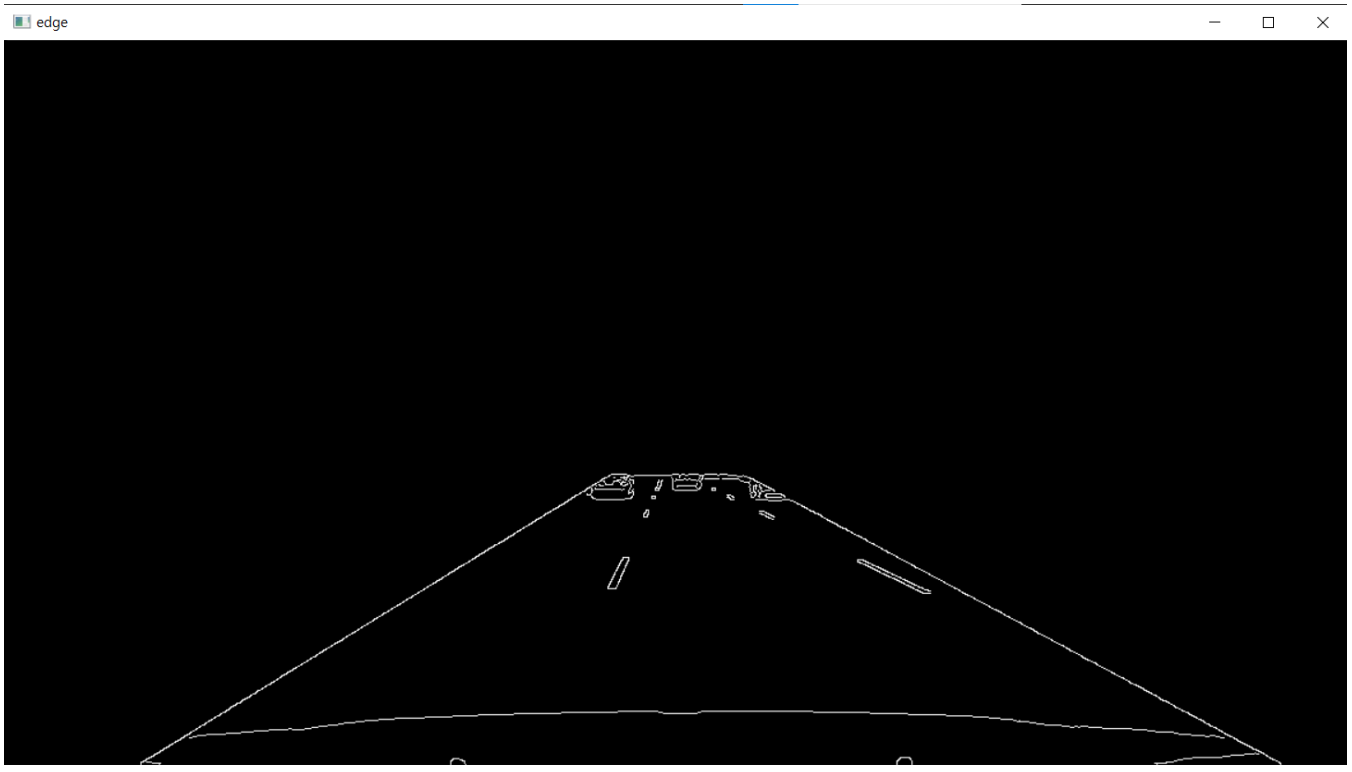
To detect only the lines of the road among the various lines in the image, we apply ROI in a trapezoidal shape around the road. In addition, blur filter was applied to reduce noise that interfered with line detection.

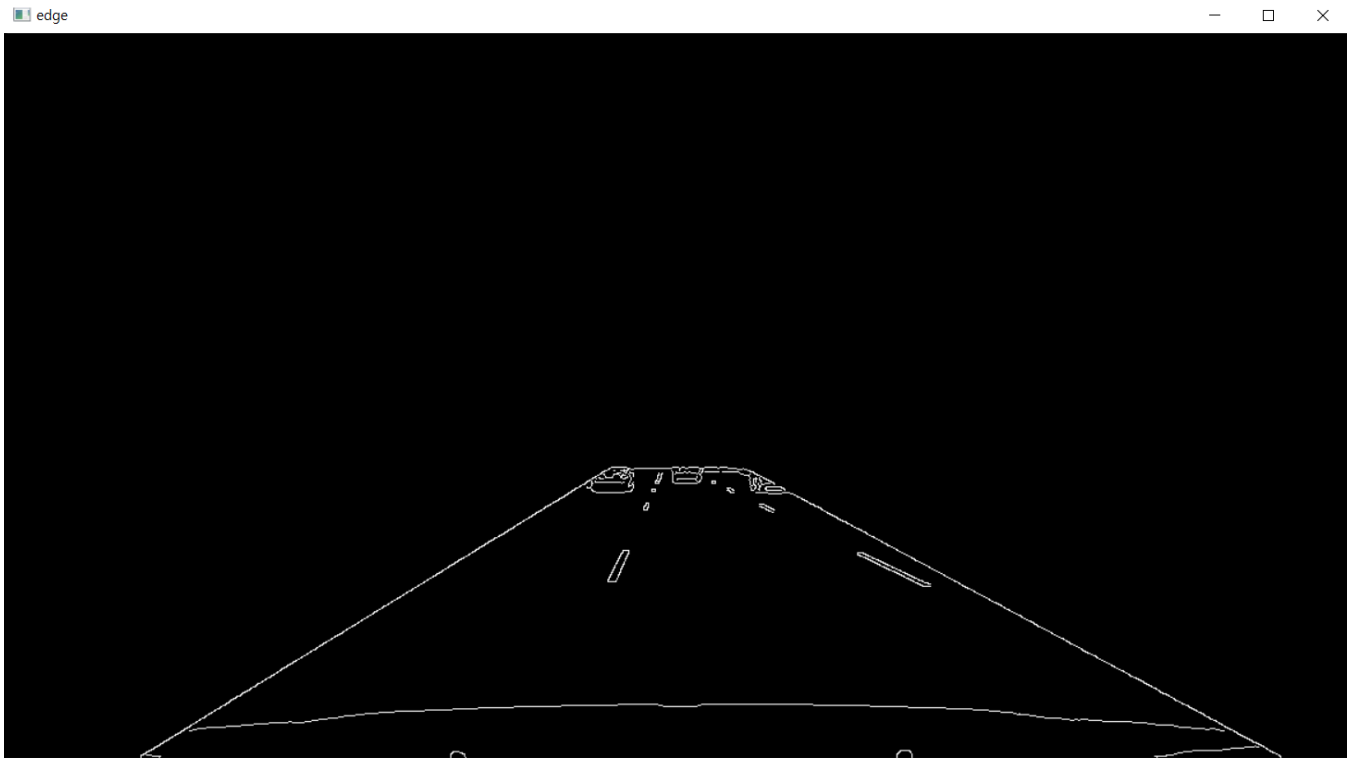

```
int width = src.cols;
int height = src.rows;
```

```
//도로 주변 사다리꼴 모양으로 roi를 딴다.
Point pts[4] = {
    Point(width * 0.1, height),              // 좌하단
    Point(width * 0.45, height * 0.6),       // 좌상단
    Point(width * 0.55, height * 0.6),       // 우상단
    Point(width * 0.95, height)               // 우하단
};
Mat mask = Mat::zeros(src_gray.size(), src_gray.type());
fillConvexPoly(mask, pts, 4, Scalar(255));
Mat roi_image;
bitwise_and(src_gray, mask, roi_image);


//roi 이미지에 대해 스무딩 필터를 적용한다.
blur(roi_image, roi_image, Size(3, 3));
```

Canny edge detecting

Canny edge detecting is used to detect the edge in the image. Set the two thresholds to 10 and 100, respectively.

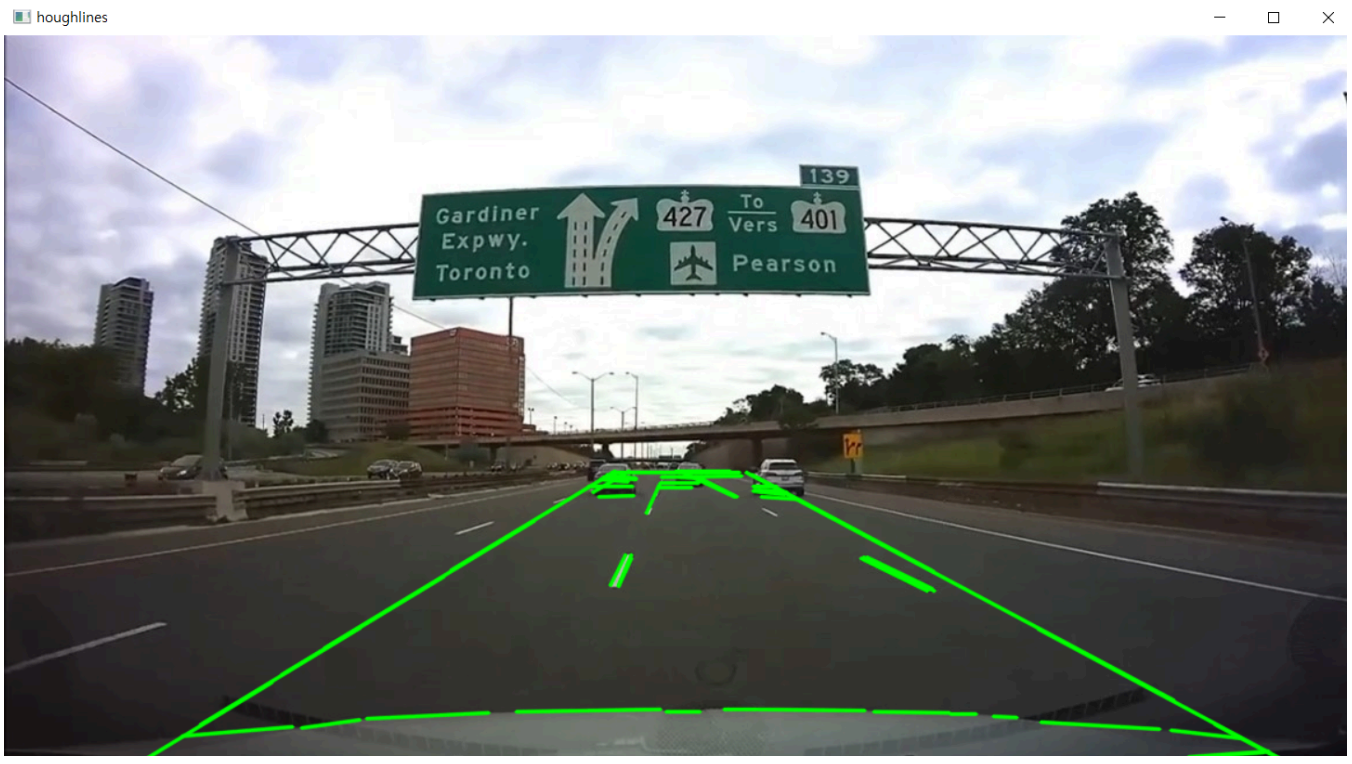





```
Mat edge;
Canny(roi_image, edge, 10, 100, 3);
```

Hough line detecting

Based on the detection of the edge, the line to the lane is detected using the HoughLinesP function.



```
vector<Vec4i> linesP;
HoughLinesP(edge, linesP, 1, CV_PI / 180, 20, 20, 15);
```

Line seperating

Separate the left lines and right lines.

Criteria:

1. Which side is based on the centerline of the image
2. the sign of the slope

```
    //중앙으로부터 어느쪽으로 떨어져있는지와 기울기의 부호를 기준으로 왼쪽 오른쪽 선을 분리한다.
    vector<Vec4i> left_lines, right_lines;

    for (auto& l : linesP) {
        double slope = double(l[3] - l[1]) / (l[2] - l[0] + 1e-6);
        int x_center = (l[0] + l[2]) / 2;

        if (slope < -0.3 && x_center < width / 2)
            left_lines.push_back(l);
        else if (slope > 0.3 && x_center > width / 2)
            right_lines.push_back(l);
    }
```

Finding Lanes

To distinguish which of the many lines is the lane , the final lane is obtained by placing the line closest to the center of the image as the lane in each direction.

```
//여러 개의 선 중 차선을 선택하여 저장할 변수를 선언하고 초기화한다.
Vec4i left_best=Vec4i(0, 0, 0, 0);
Vec4i right_best = Vec4i(0, 0, 0, 0);

//왼쪽 선, 오른쪽 선 모두 중앙과 가장 가까운 선을 차선으로 선정하여 left_best, right_best변수에 입력
한다.
int max_x_center = 0;
for (auto& l : left_lines) {
    int x_center = (l[0] + l[2]) / 2;
    if (x_center > max_x_center) {
        max_x_center = x_center;
        left_best = l;
    }
}

int min_x_center = width;
for (auto& l : right_lines) {
```

```
    int x_center = (l[0] + l[2]) / 2;
    if (x_center < min_x_center) {
        min_x_center = x_center;
        right_best = l;
    }
}
```

Drawing on the original image

1. **Draw two final lane lines and the vanishing point which is gotten using custom function on the original image.**

   Two lanes gotten from previous steps are too short, I extended lines using custom function as well.

```
//최종 결과물을 쓸 이미지를 만든다.
Mat output = src.clone();

//왼쪽 선과 오른쪽 선이 만나는 지점을 소실점으로 하여 이미지에 초록색 원으로 그린다.
Point vanish = getIntersection(left_best, right_best);
circle(output, vanish, 10, Scalar(0, 0, 255), 5);

//시선방향의 선을 파란색으로 그린다.
line(output, vanish, Point(vanish.x, height), Scalar(255, 0, 0), 1);

//최종적으로 검출한 왼쪽 차선과 오른쪽 차선을 연장하여 시각적으로 차선을 표현한다.
drawExtendedLine(output, left_best, Scalar(0, 0, 255), 2, height);
drawExtendedLine(output, right_best, Scalar(0, 255, 0), 2, height);
```

**Function getIntersection:

```
Point getIntersection(Vec4i l1, Vec4i l2) {
    Point2f a1(l1[0], l1[1]), a2(l1[2], l1[3]); // 선 1의 두 점
    Point2f b1(l2[0], l2[1]), b2(l2[2], l2[3]); // 선 2의 두 점

    float d = (a1.x - a2.x) * (b1.y - b2.y) - (a1.y - a2.y) * (b1.x - b2.x);
    if (d == 0) return Point(-1, -1); // 평행: 교차점 없음

    //크래머 공식 사용
    float px = ((a1.x * a2.y - a1.y * a2.x) * (b1.x - b2.x) - (a1.x - a2.x) * (b1.x * b2.y
- b1.y * b2.x)) / d;
    float py = ((a1.x * a2.y - a1.y * a2.x) * (b1.y - b2.y) - (a1.y - a2.y) * (b1.x * b2.y
- b1.y * b2.x)) / d;

    return Point(cvRound(px), cvRou\nd(py));
}
```

**Function drawExtendedLine:

```
//짧은 선을 같은 비율로 늘려주는 함수
void drawExtendedLine(Mat& img, Vec4i line, Scalar color, int thickness, int height) {
    double x1 = line[0], y1 = line[1];
    double x2 = line[2], y2 = line[3];

    double dx = x2 - x1;
    double dy = y2 - y1;

    // slope 안정성 체크
    if (abs(dx) < 1e-3) return;

    double slope = dy / dx;
    double intercept = y1 - slope * x1;

    // y = slope * x + intercept
    int y_top = int(height * 0.6);
    int y_bottom = height;

    int x_top = int((y_top - intercept) / slope);
    int x_bottom = int((y_bottom - intercept) / slope);

    // 유효 좌표 검사
    if (x_top < 0 || x_top > img.cols || x_bottom < 0 || x_bottom > img.cols)
        return;

    cv::line(img, Point(x_bottom, y_bottom), Point(x_top, y_top), color, thickness);
}
```

2. **fill the road with opacity**

To represents the drivable area of the road, fill the road with an opaque color.

```
//불투명하게 차선 내 도로를 표현하기 위한 행렬을 선언한다.
Mat overlay = output.clone();

// 연장된 왼쪽 선과 오른쪽 선의 아래쪽, 위쪽 점을 계산한다.
// y_top은 drawExtendedLine() 기준으로 0.6*height
int y_top = int(height * 0.6);
int y_bottom = height;

// 왼쪽 선의 위아래 점
double lx1 = left_best[0], ly1 = left_best[1];
double lx2 = left_best[2], ly2 = left_best[3];
double lslope = (ly2 - ly1) / (lx2 - lx1 + 1e-6);
double interceptL = ly1 - lslope * lx1;
int lxtop = int((y_top - interceptL) / lslope);
int lxbottom = int((y_bottom - interceptL) / lslope);

// 오른쪽 선의 위아래 점
double rx1 = right_best[0], ry1 = right_best[1];
double rx2 = right_best[2], ry2 = right_best[3];
```

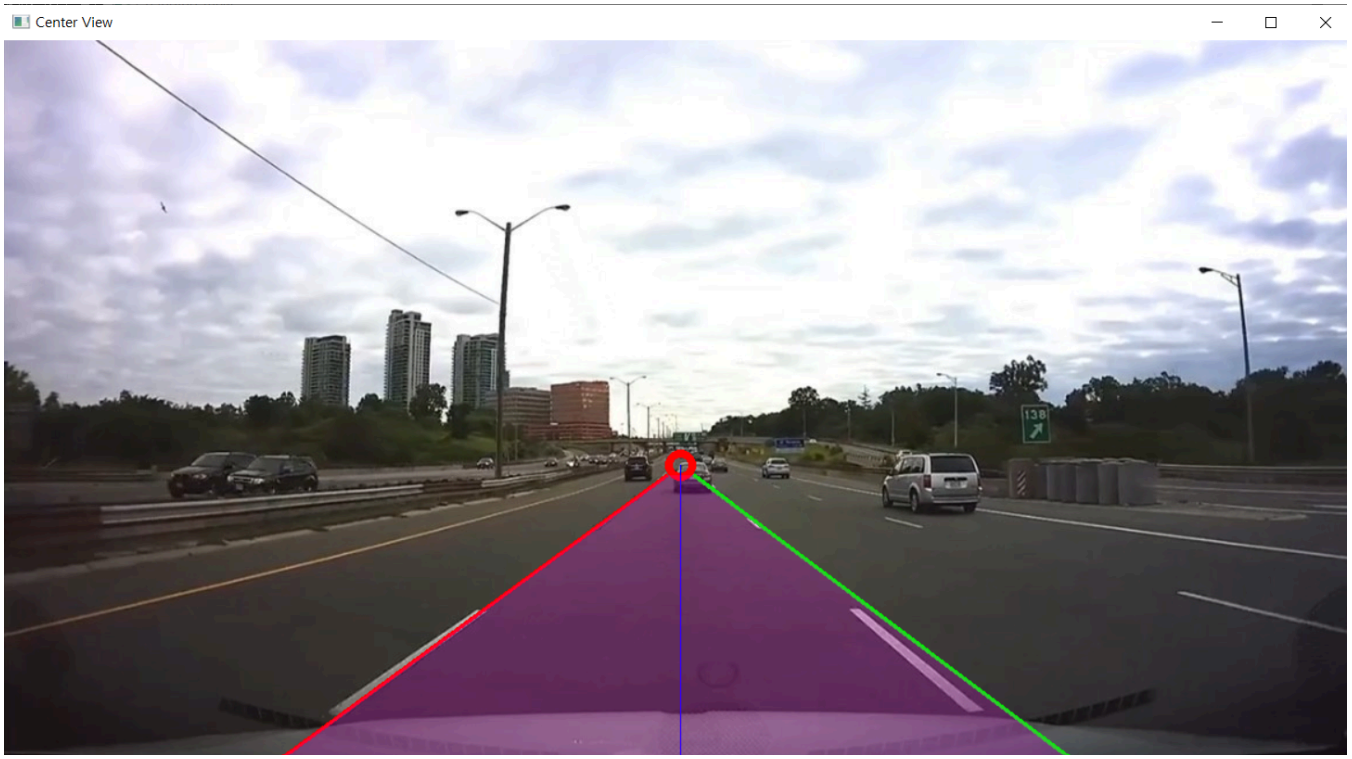
```
double rslope = (ry2 - ry1) / (rx2 - rx1 + 1e-6);
double interceptR = ry1 - rslope * rx1;
int rxtop = int((y_top - interceptR) / rslope);
int rxbottom = int((y_bottom - interceptR) / rslope);

// 도로 영역을 정의하는 사다리꼴 좌표
vector<Point> road_area = {
    Point(lxbottom, y_bottom),
    Point(lxtop, y_top),
    Point(rxtop, y_top),
    Point(rxbottom, y_bottom)
};

// 영역을 채운다. (보라색)
fillConvexPoly(overlay, road_area, Scalar(255, 0, 255));

// 반투명하게 합성한다.  (alpha blending)
double alpha = 0.2;
addWeighted(overlay, alpha, output, 1 - alpha, 0, output);
```

Result and Discussion

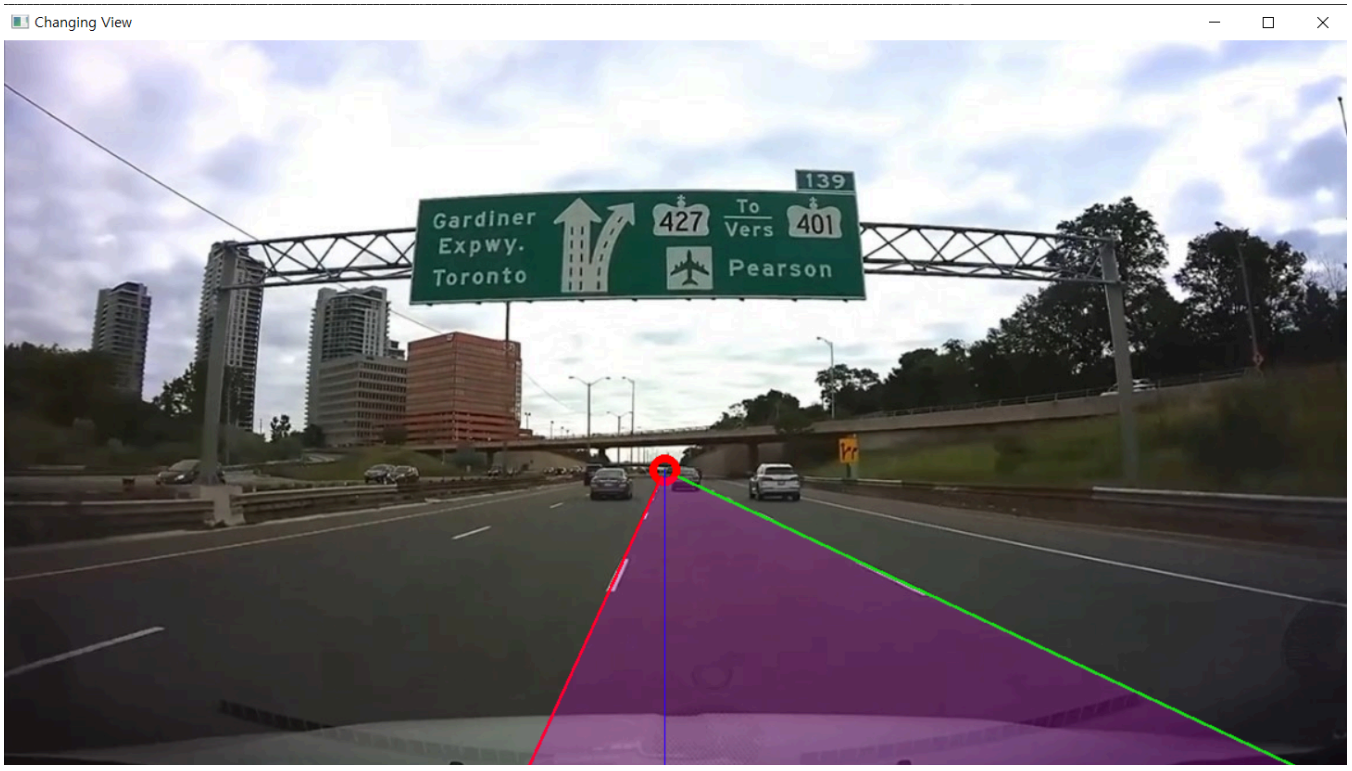
1. Final Result

Final results are as follows:

When the car is in the center of the lane



When the car is changing the lane



Canny edge detecting function

```
Canny(roi_image, edge, 10, 100, 3);
```

Threshold 1: 10

Threshold 2: 100

aperture size: 3

Hough line detecting function

```
HoughLinesP(edge, linesP, 1, CV_PI / 180, 20, 20, 15);
```

Threshold: 20

Minimum length: 20

Maximum line gap: 15

By applying these values to edge and line detection, the lane lines were successfully detected.

2. Discussion

In this project, Canny edge detection was combined with the HoughLinesP technique to effectively detect road lanes in images. In particular, a trapezoidal Region of Interest (ROI) was defined to eliminate noise from areas outside the lane, and a blurring filter was applied to enhance the accuracy of edge detection. Through this preprocessing process, meaningful lines could be reliably extracted even in complex road environments.

However, this algorithm has certain limitations. First, the line-based Hough Transform method may not work well on curved roads or roads where lane markings are faded. Second, the results of edge detection can vary depending on lighting conditions (such as nighttime or backlighting) and road surface conditions (such as snow or moisture). Lastly, since the ROI is fixed, the algorithm lacks flexibility when dealing with inclined roads or complex intersections.

This project revealed that the problems solvable by traditional image processing techniques are limited, and highlighted the need for incorporating advanced deep learning algorithms.

Conclusion

The goal of this project was to detect lane lines both when the car is centered in the lane and when it is changing lanes.

Using the Canny and Hough techniques, multiple lines were detected, and in both cases, the common feature was that the lane lines were the ones closest to the center line. By leveraging this characteristic, lane lines were successfully detected.

Appendix

```
#include <iostream>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
using namespace cv;
using namespace std;

Point getIntersection(Vec4i l1, Vec4i l2);
void drawExtendedLine(Mat& img, Vec4i line, Scalar color, int thickness, int height);
void lane_detecting(Mat& src, const string& window_name);


int main()
{
    //원본 이미지를 가져온다.
    Mat Lane_center = imread("Lane_center.jpg", IMREAD_COLOR);
    Mat Lane_changing = imread("Lane_changing.jpg", IMREAD_COLOR);

    //이미지를 불러오기 실패했을 경우 에러 메세지를 보낸다.
    if (Lane_center.empty() || Lane_changing.empty()) {
```

```
        cerr << "이미지를 불러오는 데 실패했습니다." << endl;
        return -1;
    }

    //두 가지 이미지에 대해서 차선을 검출한다.
    lane_detecting(Lane_center, "Center View");
    lane_detecting(Lane_changing, "Changing View");


    waitKey(0);
    return 0;
}

// 두 선의 교차지점을 알려주는 함수
Point getIntersection(Vec4i l1, Vec4i l2) {
    Point2f a1(l1[0], l1[1]), a2(l1[2], l1[3]); // 선 1의 두 점
    Point2f b1(l2[0], l2[1]), b2(l2[2], l2[3]); // 선 2의 두 점

    float d = (a1.x - a2.x) * (b1.y - b2.y) - (a1.y - a2.y) * (b1.x - b2.x);
    if (d == 0) return Point(-1, -1); // 평행: 교차점 없음

    //크래머 공식 사용
    float px = ((a1.x * a2.y - a1.y * a2.x) * (b1.x - b2.x) - (a1.x - a2.x) * (b1.x * b2.y
- b1.y * b2.x)) / d;
    float py = ((a1.x * a2.y - a1.y * a2.x) * (b1.y - b2.y) - (a1.y - a2.y) * (b1.x * b2.y
- b1.y * b2.x)) / d;

    return Point(cvRound(px), cvRound(py));
}



//짧은 선을 같은 비율로 늘려주는 함수
void drawExtendedLine(Mat& img, Vec4i line, Scalar color, int thickness, int height) {
    double x1 = line[0], y1 = line[1];
    double x2 = line[2], y2 = line[3];

    double dx = x2 - x1;
    double dy = y2 - y1;

    // slope 안정성 체크
    if (abs(dx) < 1e-3) return;

    double slope = dy / dx;
    double intercept = y1 - slope * x1;

    // y = slope * x + intercept
    int y_top = int(height * 0.6);
    int y_bottom = height;

    int x_top = int((y_top - intercept) / slope);
    int x_bottom = int((y_bottom - intercept) / slope);
```

```
    // 유효 좌표 검사
    if (x_top < 0 || x_top > img.cols || x_bottom < 0 || x_bottom > img.cols)
        return;

    cv::line(img, Point(x_bottom, y_bottom), Point(x_top, y_top), color, thickness);
}



void lane_detecting(Mat& src, const string& window_name)
{
    Mat src_gray;

    //가져온 이미지를 그레이스케일로 바꾼다.
    cvtColor(src, src_gray, cv::COLOR_BGR2GRAY);



    int width = src.cols;
    int height = src.rows;

    //도로 주변 사다리꼴 모양으로 roi를 딴다.
    Point pts[4] = {
        Point(width * 0.1, height),              // 좌하단
        Point(width * 0.45, height * 0.6),       // 좌상단
        Point(width * 0.55, height * 0.6),       // 우상단
        Point(width * 0.95, height)               // 우하단
    };
    Mat mask = Mat::zeros(src_gray.size(), src_gray.type());
    fillConvexPoly(mask, pts, 4, Scalar(255));
    Mat roi_image;
    bitwise_and(src_gray, mask, roi_image);


    //roi 이미지에 대해 스무딩 필터를 적용한다.
    blur(roi_image, roi_image, Size(3, 3));


    //스무딩필터를 적용한 roi이미지에 대해 canny edge detecting 기법을 사용하여 엣지를 검출한다.
    Mat edge;
    Canny(roi_image, edge, 10, 100, 3);



    //엣지에서 라인을 따 linesP에 정보를 저장한다.
    vector<Vec4i> linesP;
    HoughLinesP(edge, linesP, 1, CV_PI / 180, 20, 20, 15);



    //중앙으로부터 어느쪽으로 떨어져있는지와 기울기의 부호를 기준으로 왼쪽 오른쪽 선을 분리한다.
    vector<Vec4i> left_lines, right_lines;
```

```
    for (auto& l : linesP) {
        double slope = double(l[3] - l[1]) / (l[2] - l[0] + 1e-6);
        int x_center = (l[0] + l[2]) / 2;

        if (slope < -0.3 && x_center < width / 2)
            left_lines.push_back(l);
        else if (slope > 0.3 && x_center > width / 2)
            right_lines.push_back(l);
    }


    //여러 개의 선 중 차선을 선택하여 저장할 변수를 선언하고 초기화한다.
    Vec4i left_best=Vec4i(0, 0, 0, 0);
    Vec4i right_best = Vec4i(0, 0, 0, 0);

    //왼쪽 선, 오른쪽 선 모두 중앙과 가장 가까운 선을 차선으로 선정하여 left_best, right_best변수에
입력한다.
    int max_x_center = 0;
    for (auto& l : left_lines) {
        int x_center = (l[0] + l[2]) / 2;
        if (x_center > max_x_center) {
            max_x_center = x_center;
            left_best = l;
        }
    }

    int min_x_center = width;
    for (auto& l : right_lines) {
        int x_center = (l[0] + l[2]) / 2;
        if (x_center < min_x_center) {
            min_x_center = x_center;
            right_best = l;
        }
    }

    //최종 결과물을 쓸 이미지를 만든다.
    Mat output = src.clone();

    //왼쪽 선과 오른쪽 선이 만나는 지점을 소실점으로 하여 이미지에 초록색 원으로 그린다.
    Point vanish = getIntersection(left_best, right_best);
    circle(output, vanish, 10, Scalar(0, 0, 255), 5);

    //시선방향의 선을 파란색으로 그린다.
    line(output, vanish, Point(vanish.x, height), Scalar(255, 0, 0), 1);

    //최종적으로 검출한 왼쪽 차선과 오른쪽 차선을 연장하여 시각적으로 차선을 표현한다.
    drawExtendedLine(output, left_best, Scalar(0, 0, 255), 2, height);
    drawExtendedLine(output, right_best, Scalar(0, 255, 0), 2, height);

    //불투명하게 차선 내 도로를 표현하기 위한 행렬을 선언한다.
    Mat overlay = output.clone();

    // 연장된 왼쪽 선과 오른쪽 선의 아래쪽, 위쪽 점을 계산한다.
```

```
    // y_top은 drawExtendedLine() 기준으로 0.6*height
    int y_top = int(height * 0.6);
    int y_bottom = height;

    // 왼쪽 선의 위아래 점
    double lx1 = left_best[0], ly1 = left_best[1];
    double lx2 = left_best[2], ly2 = left_best[3];
    double lslope = (ly2 - ly1) / (lx2 - lx1 + 1e-6);
    double interceptL = ly1 - lslope * lx1;
    int lxtop = int((y_top - interceptL) / lslope);
    int lxbottom = int((y_bottom - interceptL) / lslope);

    // 오른쪽 선의 위아래 점
    double rx1 = right_best[0], ry1 = right_best[1];
    double rx2 = right_best[2], ry2 = right_best[3];
    double rslope = (ry2 - ry1) / (rx2 - rx1 + 1e-6);
    double interceptR = ry1 - rslope * rx1;
    int rxtop = int((y_top - interceptR) / rslope);
    int rxbottom = int((y_bottom - interceptR) / rslope);

    // 도로 영역을 정의하는 사다리꼴 좌표
    vector<Point> road_area = {
        Point(lxbottom, y_bottom),
        Point(lxtop, y_top),
        Point(rxtop, y_top),
        Point(rxbottom, y_bottom)
    };

    // 영역을 채운다. (보라색)
    fillConvexPoly(overlay, road_area, Scalar(255, 0, 255));

    // 반투명하게 합성한다.  (alpha blending)
    double alpha = 0.2;
    addWeighted(overlay, alpha, output, 1 - alpha, 0, output);

    //최종 결과물을 출력한다.

    namedWindow(window_name, cv::WINDOW_AUTOSIZE);
    imshow(window_name, output);


}
```