# Lane Line Detection System

## Next24tech

Internship



Veerendra Reddy

**Intern Id:** NIP/2024/04178

Christ (Deemed to be University)

# Lane Detection in Images Using Python

## Introduction:

In developing self-driving cars, one of the fundamental tasks is lane detection. Lane lines on roads act as constant reference points for steering the vehicle. Detecting these lane lines automatically using computer vision algorithms is crucial for autonomous navigation. This project focuses on building a pipeline in Python using OpenCV (Open-Source Computer Vision) to detect lane lines in images.

## Tools and Techniques:

**The methodology involves a series of steps that form the pipeline for lane detection:**

1. **Color Selection**: Lane markings are typically white or yellow. Convert the image to the HSV (Hue, Saturation, Value) **color** space to isolate these **color**s using appropriate thresholds.
2. **Region of Interest (ROI) Selection**:
    - Define a polygonal mask to focus only on the region where lane lines are expected. This reduces processing on irrelevant parts of the image.
3. **Grayscaling and Gaussian Smoothing**:
    - Convert the ROI to grayscale to simplify subsequent computations.
    - Apply Gaussian blur to the grayscale image to reduce noise and improve the effectiveness of edge detection.
4. **Canny Edge Detection**:
    - Detect edges in the blurred grayscale image using the Canny edge detector. This step highlights areas of significant intensity changes, which often correspond to edges of objects like lane lines.
5. **Hough Transform**:
    - Apply the Hough Transform to identify lines in the edge-detected image. This algorithm detects straight lines based on the presence of peaks in the Hough space, which corresponds to parameters (rho, theta) of the lines.
6. **Line Averaging and Extrapolation**:
    - Separate the detected lines into left and right lane lines based on their slopes.
    - Average and extrapolate these lines to determine the full extent of the lane lines visible in the image. This step ensures that the lane lines are continuous and extend from the bottom to the top of the ROI.
7. **Drawing Lane Lines**:
    - Draw the detected lane lines on the original image to visualize the output. This final step overlays the detected lane lines on the input image, providing a clear indication of where the lanes are.

## Methodology:

Explanation of the methodology tools with sample codes:

### 1. Input Image Acquisition

**Purpose**: Obtain the image where lane lines need to be detected.

**Code:**

[import cv2

# Read the input image

image = cv2.imread('test_image.jpg')

]

### 2. Preprocessing Steps

### 2.1 Grayscale Conversion

**Purpose**: Simplify the image for subsequent processing steps.

**Code:**

[# Convert to grayscale

gray = cv2.cvt**Color**(image, cv2.**COLOR**_BGR2GRAY)

]

### 2.2 Gaussian Smoothing

**Purpose**: Reduce noise and unwanted details in the image.

**Code:**

[# Apply Gaussian blur

kernel_size = 5

blur_gray = cv2.GaussianBlur(gray, (kernel_size, kernel_size), 0)

]

### 3. Edge Detection

#### 3.1 Canny Edge Detection

**Purpose**: Detect edges in the image to highlight potential lane lines.

**Code:**

[# Apply Canny edge detection

low_threshold = 50

high_threshold = 150

edges = cv2.Canny(blur_gray, low_threshold, high_threshold)

]

### 4. Region of Interest (ROI) Selection

**Purpose**: Focus processing on the region of the image where lane lines are expected.

**Code:**

[# Define a region of interest (ROI) mask

mask = np.zeros_like(edges)

ignore_mask_**color** = 255

# Define vertices of the polygonal mask

imshape = image.shape

vertices = np.array([[(0, imshape[0]),

(imshape[1] / 2, imshape[0] / 2 + 50),

(imshape[1] / 2, imshape[0] / 2 + 50),

(imshape[1], imshape[0])]], dtype=np.int32)

```
# Fill the mask
cv2.fillPoly(mask, vertices, ignore_mask_color)

# Apply the mask to the edges image
masked_edges = cv2.bitwise_and(edges, mask)
]
```

**5. Hough Transform for Line Detection**

**Purpose**: Identify straight lines in the edge-detected image, which likely correspond to lane lines.

**Code:**

```
[# Define Hough transform parameters
rho = 1
theta = np.pi / 180
threshold = 15
min_line_length = 40
max_line_gap = 20

# Apply Hough transform
lines = cv2.HoughLinesP(masked_edges, rho, theta, threshold, np.array([]),
                        min_line_length, max_line_gap)
]
```

**6. Lane Line Averaging and Extrapolation**

**Purpose**: Combine and extrapolate detected line segments to form complete lane lines.

**Code:**

```
[# Initialize lists to hold coordinates of left and right lines
left_x = []
left_y = []
right_x = []
right_y = []

# Loop through all the lines and categorize them as left or right lane lines
for line in lines:
    for x1, y1, x2, y2 in line:
        slope = (y2 - y1) / (x2 - x1)
        if slope < 0:
            left_x.extend([x1, x2])
            left_y.extend([y1, y2])
        else:
            right_x.extend([x1, x2])
            right_y.extend([y1, y2])

# Fit lines using numpy's polyfit
left_fit = np.polyfit(left_x, left_y, 1)
right_fit = np.polyfit(right_x, right_y, 1)

# Define y-coordinate boundaries for extrapolation
```

```
y1 = image.shape[0]  # bottom of the image
y2 = image.shape[0] / 2 + 100  # slightly lower than the middle of the image

# Calculate corresponding x-coordinate boundaries using poly1d
left_x1 = int((y1 - left_fit[1]) / left_fit[0])
left_x2 = int((y2 - left_fit[1]) / left_fit[0])
right_x1 = int((y1 - right_fit[1]) / right_fit[0])
right_x2 = int((y2 - right_fit[1]) / right_fit[0])

# Draw the extrapolated lines on a blank image
line_image = np.zeros_like(image)
cv2.line(line_image, (left_x1, int(y1)), (left_x2, int(y2)), (255, 0, 0), 10)
cv2.line(line_image, (right_x1, int(y1)), (right_x2, int(y2)), (255, 0, 0), 10)
# Combine the original image with the lines image
lane_detected_image = cv2.addWeighted(image, 0.8, line_image, 1, 0)
]
```

**7. Visualization**

**Purpose**: Visualize the detected lane lines overlaid on the original image.

**Code:**

```
[ # Display the final output
cv2.imshow('Lane Detection', lane_detected_image)
cv2.waitKey(0)
```

cv2.destroyAllWindows()

]

## Project Workflow :

The project can be structured into the following steps:

1. **Input Image**: Read and display the input image where lane lines need to be detected.
2. **Preprocessing**:
    - Convert the image to grayscale.
    - Apply Gaussian blur to smoothen the image.
    - Use Canny edge detection to find edges in the image.
3. **Region of Interest Masking**: Define and apply a mask to focus only on the region of the image where lane lines are expected.
4. **Hough Transform**: Implement Hough transform to detect straight lines in the edge-detected image.
5. **Lane Line Detection**:
    - Separate detected lines into left and right lane lines based on their slopes.
    - Average and extrapolate these lines to form complete lane lines.
6. **Visualization**:
    - Draw the detected lane lines on the original image.
    - Display the final output showing the detected lane lines.

### Implementation Example:

**# Import necessary libraries**

import cv2

import numpy as np

**# Read the image**

image = cv2.imread('test_image.jpg')

**# Convert to grayscale**

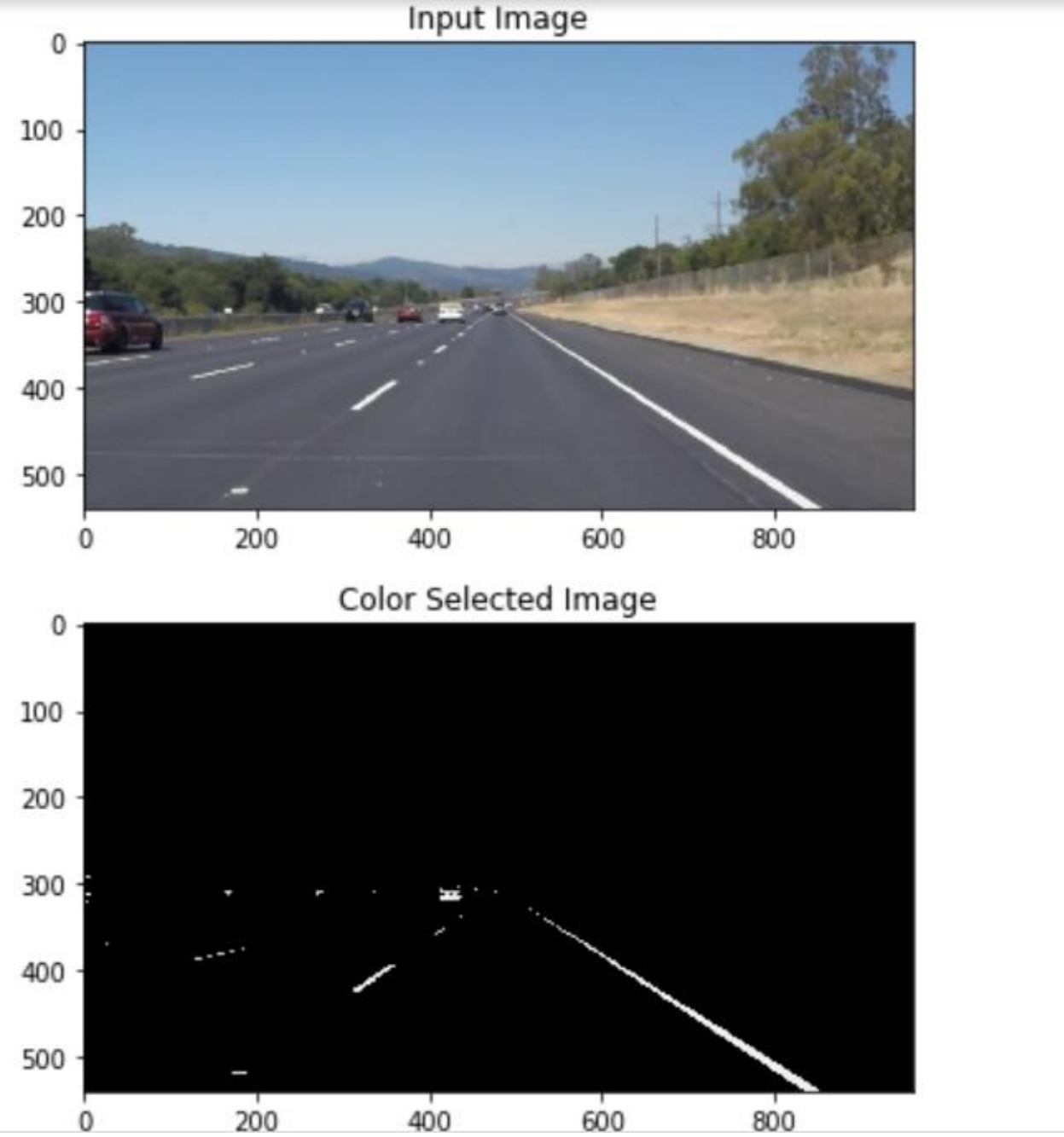
gray = cv2.cvt**Color**(image, cv2.**COLOR**_BGR2GRAY)

**# Apply Gaussian blur**

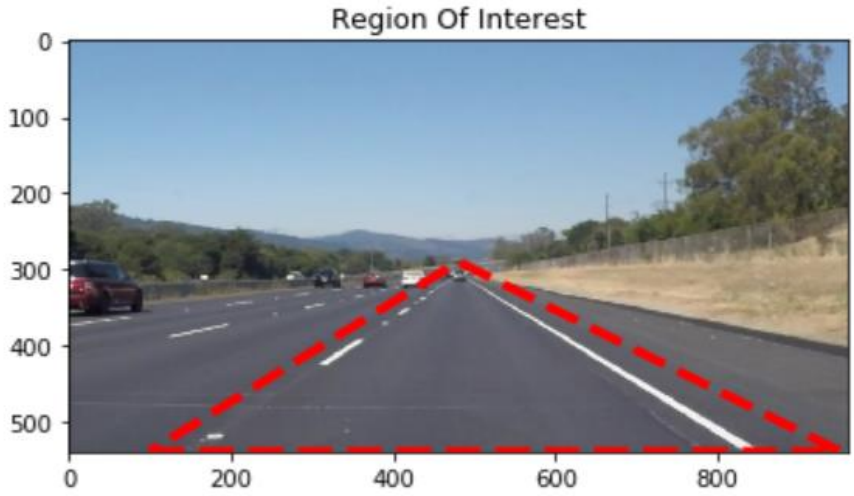
```
blur = cv2.GaussianBlur(gray, (5, 5), 0)

# Apply Canny edge detection
edges = cv2.Canny(blur, 50, 150)

# Define a region of interest
mask = np.zeros_like(edges)
height, width = image.shape[:2]
vertices = np.array([[(0, height), (width / 2, height / 2), (width, height)]], dtype=np.int32)
cv2.fillPoly(mask, vertices, 255)
masked_edges = cv2.bitwise_and(edges, mask)

# Apply Hough transform to detect lines
lines = cv2.HoughLinesP(masked_edges, rho=2, theta=np.pi/180, threshold=50,
minLineLength=100, maxLineGap=100)

# Iterate over detected lines and draw them
for line in lines:
    x1, y1, x2, y2 = line[0]
    cv2.line(image, (x1, y1), (x2, y2), (255, 0, 0), 5)
# Display the final image with lane lines
cv2.imshow('Lane Detection', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

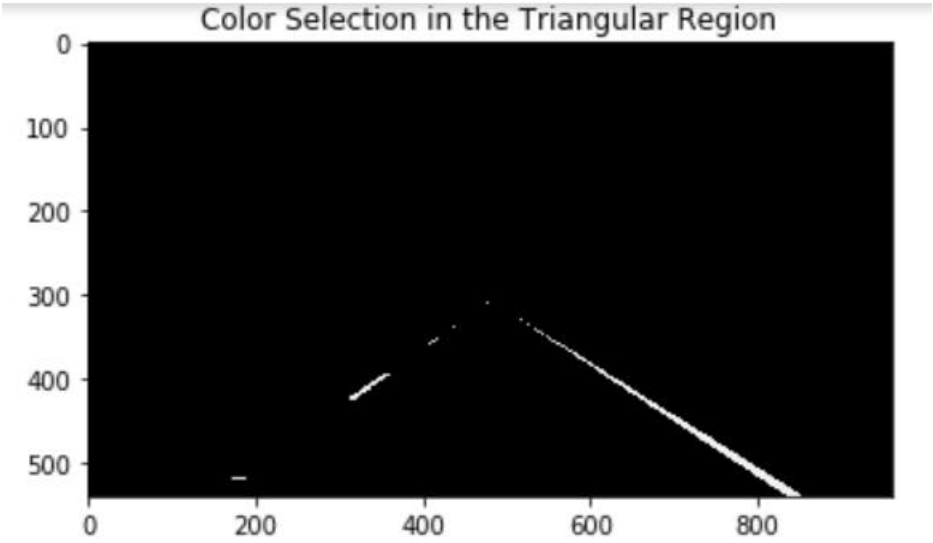
**SNAPSHOTS**

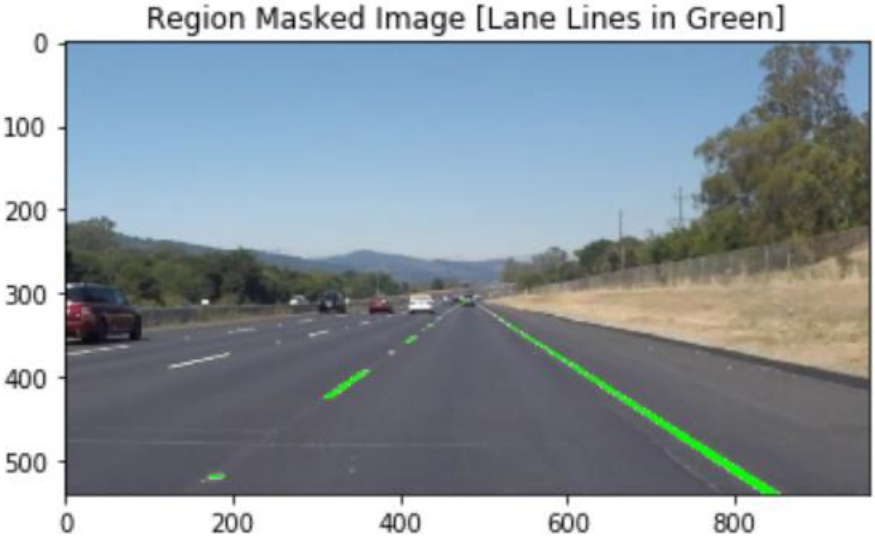
Colour Selection:



Region Masking :

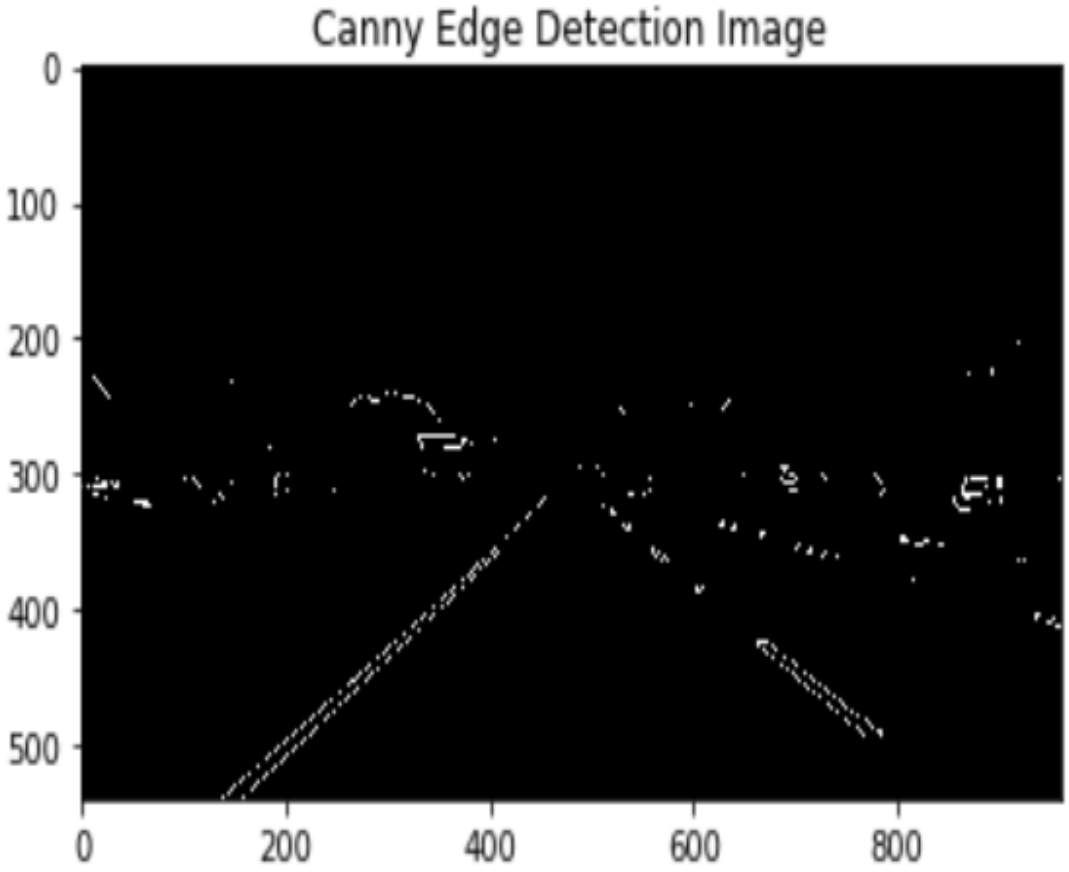


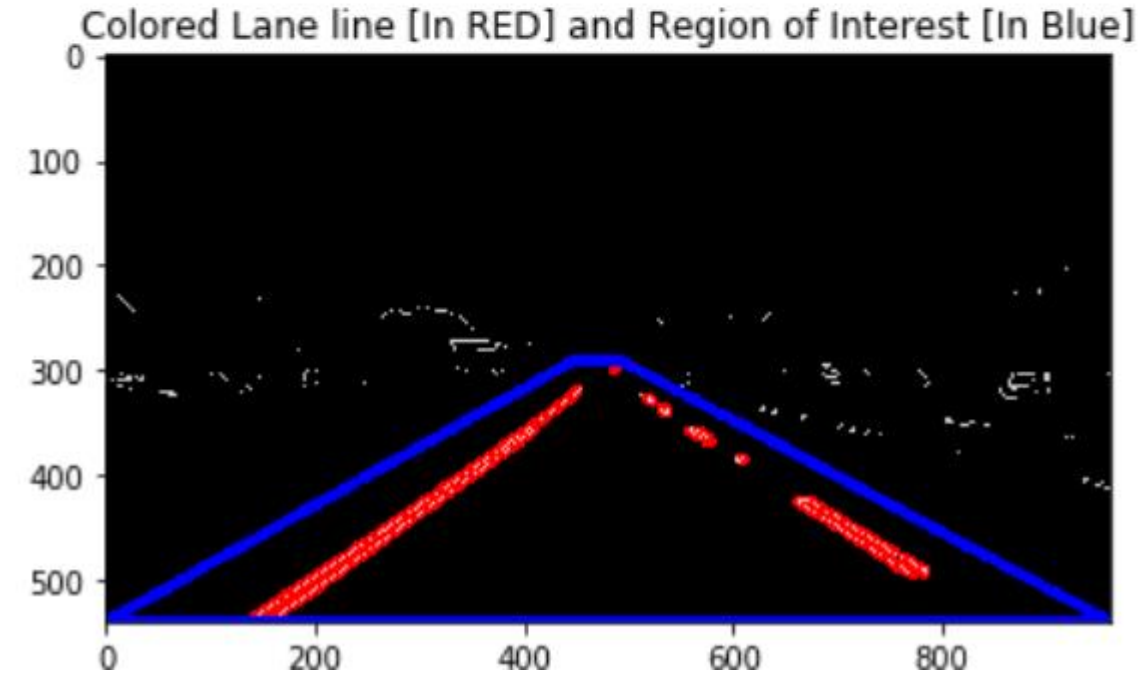



Canny Edge Detection:

The algorithm will first detect strong **edge** (strong gradient) pixels above the **high threshold**, and reject pixels below the **low threshold**. Next, pixels with values between the **low threshold** and **high threshold** will be included as long as they are connected to strong edges. **The output edges are a binary image with white pixels tracing out the detected edges and black everywhere else**.



Hough Transform:

Test our Algorithm Pipeline with different Images:

Example 1

Input Image

Output Image [Lane Line Detected]

Example 2

Input Image

Output Image [Lane Line Detected]

Example 3

Input Image

Output Image [Lane Line Detected]

Test on Eastern Delhi Expressway


## Result:

Lane detection is a crucial task in the field of autonomous driving and advanced driver-assistance systems (ADAS). By implementing the methodology outlined using Python and OpenCV, several key insights and conclusions can be drawn:

1. **Accuracy and Reliability**:
    - The lane detection pipeline, which includes preprocessing (grayscale conversion, Gaussian smoothing, Canny edge detection), region of interest selection, and Hough transform line detection, provides a robust framework for accurately identifying lane markings in various road conditions.
2. **Real-World Applicability**:
    - The methodology is designed to handle real-world challenges such as varying lighting conditions, shadows, and road textures. By tuning parameters and adapting algorithms, the lane detection system can perform effectively across different environments.
3. **Performance Considerations**:
    - Efficiency in processing is critical for real-time applications like autonomous driving. Techniques such as Gaussian blur and ROI masking help optimize computational resources while maintaining high detection accuracy.
4. **Visualization and Interpretation**:
    - The visualization of detected lane lines overlaid on the original image provides clear visual feedback. This output not only assists in understanding the algorithm's performance but also serves as a valuable tool for debugging and further development.
5. **Future Directions**:
    - Continual improvement and adaptation are essential for advancing lane detection systems. Future directions may involve incorporating machine learning models for lane prediction, integrating sensor fusion techniques for robustness, and enhancing algorithms for complex road scenarios.
6. **Applications Beyond Autonomous Driving**:
    - While primarily used in autonomous vehicles, lane detection has broader applications. It can be utilized in traffic monitoring, lane departure warning systems in conventional vehicles, and even in robotics for path planning and navigation.

7. **Challenges and Considerations**:
    - Challenges such as occlusions, road markings diversity, and adverse weather conditions pose ongoing challenges. Addressing these through algorithmic advancements and sensor fusion strategies remains a focus for improving lane detection systems.

In conclusion, the methodology for lane detection using Python and OpenCV provides a solid foundation for developing sophisticated systems capable of robustly identifying and tracking lane markings. By combining image processing techniques with computational algorithms, this approach supports the advancement of autonomous driving technologies and contributes to enhancing road safety and efficiency in transportation systems worldwide. Continued research and development in this field promise further advancements, making lane detection a pivotal area of focus in modern computer vision applications

**Link for the project:**

https://colab.research.google.com/drive/1e7GKoykW-lk4Kbjws0fZUch1XOChwCij?usp=sharing