Gebze Technical University

Engineering Department

Computer Engineering

Line Detection

Ahmet Yigit – 200104004066

Özcan Akay – 2001040040XX

12/2024

CONTENTS

1. Abstraction

a. Background and Motivation

i. Line detection is a fundamental problem in computer vision and image processing, with applications ranging from autonomous vehicles (lane detection) to industrial automation and image analysis. This research aims to develop and evaluate a robust line detection algorithm that can handle various image conditions, including noise and challenging lighting. The ability to accurately detect lines is crucial for tasks such as road lane marking detection in self-driving cars, robotic vision, and optical character recognition (OCR).

b. Objective of the Study

i. The primary objective of this study is to develop an efficient algorithm for detecting lines in images and video sequences. The research aims to:

    1. Implement a line detection pipeline using image processing techniques.
    2. Evaluate the effectiveness of different edge detection methods, particularly Sobel filtering and Canny edge detection.
    3. Utilize the Hough Transform for accurate line detection and lane extraction.
    4. Compare the performance of line detection with and without Canny edge detection in terms of accuracy and processing time.

c. Methodology Overview

i. The proposed algorithm begins by preprocessing the image to enhance contrast and reduce noise. This is followed by edge detection using either the Sobel filter or the Canny edge detector, which is crucial for highlighting potential line features. The Hough Transform is then applied to detect lines based on the edge-detected image, with road-specific optimizations. Finally, the detected lines are grouped into left and right lanes, followed by the filling of the space between the lanes to highlight the region of interest in the image. Temporal smoothing is also incorporated for video sequences to reduce jitter and improve lane continuity.

ii. **Key steps involved in the methodology include:**

1. **Preprocessing**: Enhancing contrast and reducing noise.
2. **Edge Detection:** Using Sobel or Canny edge detection methods.
3. **Hough Transform:** Detecting lines in the edge-detected image.
4. **Lane Grouping**: Classifying lines into left and right lanes based on slope criteria.
5. **Smoothing**: Temporal smoothing of lane detection for video applications.

d. Test Cases and Results

i. To evaluate the performance of the algorithm, several test cases were used, including both static images and video sequences. Test cases were conducted with and without the application of the Canny edge detection method. The results demonstrated the accuracy of lane detection, with the Canny edge detection yielding more precise and robust results in noisy or dynamic environments.

ii. **Key findings from the test cases include:**

1. The Sobel filter effectively detects edges but is more susceptible to noise compared to the Canny edge detector.
2. Temporal smoothing improved the consistency of lane detection in video sequences.

e. Conclusion and Future Works

i. In conclusion, the algorithm successfully detected lanes and lines in images. The incorporation of Canny edge detection significantly improved the algorithm's robustness to noise and varying conditions, but improvement comes some disadvantages like changing of light may problem for it. Future work may focus on optimizing the algorithm for real-time performance, exploring alternative edge detection methods, and extending the approach to handle curved lanes or more complex road environments. Additionally, exploring the use of deep learning for line detection could further enhance the accuracy and adaptability of the system in diverse real-world scenarios.

2. Introduction

a. Background

i. Line detection is one of the core challenges in computer vision and image processing. In many real-world applications, the ability to identify and extract lines from images is crucial. One prominent application of line detection is in autonomous driving, where detecting road lane markings in real-time is essential for safe navigation. Other applications of line detection include industrial automation (for robot path planning), optical character recognition (OCR) for document scanning, and medical imaging (such as detecting veins or edges in X-ray images).

ii. The concept of line detection in an image is closely related to edge detection and pattern recognition. When an image is captured, it contains various features such as edges, textures, and colors. Lines, as geometric structures, are typically represented as strong gradients in intensity or color in images. Therefore, the challenge of line detection often begins with edge detection, which identifies significant changes in pixel intensity. Afterward, methods like the Hough Transform can be used to detect the geometric representation of these edges as lines.

iii. Several methods have been developed to detect lines, including Sobel edge detection, Canny edge detection, and Hough Transform, each of which has its strengths and weaknesses. Sobel edge detection, for instance, uses gradient-based methods to find edges in images, while the Canny edge detector provides a more sophisticated, multi-stage approach to edge detection with better noise suppression. The Hough Transform is particularly effective in line detection, as it can detect straight lines in an image, even if they are fragmented or partially obscured.

b. Objective of the Study

i. The primary goal of this study is to design, implement, and evaluate a line detection algorithm using image processing techniques. This includes preprocessing steps to enhance the image quality, edge detection methods to identify potential lines, and the use of the Hough Transform to extract lines from the detected edges. By developing and testing the algorithm, we aim to:

   1. **Develop a Robust Line Detection Pipeline:** The algorithm should handle real-world scenarios, such as noisy images or dynamic video sequences, where detecting lines may be challenging.

2. **Compare Edge Detection Methods:** We will compare the effectiveness of the Sobel edge detection method against the Canny edge detection method in terms of accuracy, noise reduction, and computational efficiency.

3. **Apply Hough Transform:** Implement the Hough Transform to detect lines in edge-detected images, specifically optimizing it for road lane detection, such as distinguishing left and right lanes in traffic scenarios.

c. Evaluate Performance:

i. Through various test cases, we aim to assess the performance of the algorithm in both static and dynamic environments. We will also compare the results of using Sobel edge detection versus Canny edge detection in the context of line detection accuracy and robustness.

ii. In essence, this study strives to offer an efficient, reliable, and accurate approach to line detection, which could be beneficial in a wide range of real-time applications, especially those requiring lane or road detection in autonomous systems.

d. Challenges

i. The task of line detection presents several challenges that need to be addressed in the design of the algorithm:

1. **Noise in the Image:** Real-world images often contain noise, which can obscure the edges and lines that need to be detected. Noise can come from various sources such as camera sensor limitations, environmental conditions, etc. Noise reduction method is crucial for accurate line detection.

2. **Lighting and Contrast Variations:** Images captured under different lighting conditions or with varying contrast may result in weak or indistinct edges. Preprocessing steps, such as contrast enhancement and selective region masking, can help improve line visibility in such cases.

3. **Edge Fragmentation:** In many cases, lines are not continuous but fragmented due to occlusions, partial visibility, or imperfections in the image (such as road wear or dirt). The Hough Transform is used here to detect these fragmented lines by mapping edge points to a parameter space and then identifying the lines that best fit these edge points.

4. **Real-time Performance:** Line detection is often required in real-time applications, such as autonomous driving or video surveillance. Achieving high accuracy while maintaining processing speed is a significant challenge, especially in video processing, where multiple frames need to be analyzed per second.

5. **Lane Detection in Complex Road Environments:** For autonomous vehicles, road lanes may not always be straight and may involve curves or intersections. Developing an algorithm that can adapt to these variations while maintaining accuracy and robustness is a complex task.

e. Contribution

i. This study makes several contributions to the field of line detection and image processing:

1. **Implementation of a Novel Line Detection Pipeline:** The proposed algorithm combines several well-known images processing techniques, including Sobel and Canny edge detection, the Hough Transform, and lane grouping, into a unified pipeline designed for real-time video processing applications.

2. **Comparison of Edge Detection Methods:** A significant contribution of this work is the comparison between two popular edge detection techniques: Sobel and Canny. While Sobel is simpler and computationally less intensive, Canny is more sophisticated and capable of handling noise better. By conducting tests on various images and video sequences, this study evaluates their respective performances in terms of robustness and accuracy.

3. **Application to Lane Detection:** The algorithm is particularly optimized for road lane detection, a key application in autonomous driving. By grouping detected lines into left and right lanes and smoothing them over time, the algorithm can detect lanes even in noisy or dynamic environments such as video sequences.

f. In conclusion, this study provides both a theoretical and practical contribution to line detection research, presenting an effective method for detecting lines in images and videos, with potential applications in autonomous driving and other fields of computer vision.

3. Functions of Line Detection

a. Noise Reduction

i. Gaussian Blur

1. Gaussian Blur is a widely used technique in image processing to reduce noise and smooth out details in an image. This technique is particularly important for ensuring that subsequent operations like edge detection and feature extraction can be performed more effectively and accurately. Gaussian Blur is applied using a **Gaussian function**, which is the core of this smoothing process.

2. What is Gaussian Blur?
    a. Gaussian Blur is a process where each pixel in an image is modified by the weighted average of its neighboring pixels. The weights for the averaging process are determined based on the Gaussian distribution, meaning pixels closer to the center have more influence than those farther away. Mathematically, the function used for the Gaussian Blur process is defined by the **Gaussian function** as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \cdot \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

Gaussian Function (eq. 3.1.1)

    b. Where: G (x, y) is the value of the Gaussian function at the pixel coordinates (x, y).
    c. σ is the standard deviation of the Gaussian distribution, which controls the amount of blurring. A higher σ results in more blur.
    d. x and y are the pixel coordinates relative to the center of the kernel.
    e. This function (eq. 3.1.1) is used to calculate the weights for each pixel's neighboring pixels. Then, each pixel's new value is computed as a weighted average of its neighbors, with the weights determined by the Gaussian function.

3. Steps of Applying Gaussian Blur
    a. **Defining the Kernel:** The kernel is defined using the Gaussian function. The kernel is typically represented as a matrix.
        i. $$\begin{bmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{bmatrix}$$

    Gaussian Example Kernel Matrix (fig. 3.1.1)

    b. In this kernel (fig 3.1.1), each element represents the weight of the corresponding pixel. The values are derived from the Gaussian function, where the central pixel has the highest weight, and the weights decrease as the distance from the center increases.

    c. **Applying the Kernel to the Image:** The kernel is applied to each pixel in the image. For each pixel, the kernel is centered on that pixel, and the weighted average of the surrounding pixels is calculated. The result of this operation replaces the original pixel value.

    d. **Edge Handling:** When applying the Gaussian kernel at the edges of the image (where there aren't enough neighboring pixels), different strategies can be employed to handle the borders.

4. Effect and Importance of Gaussian Blur
    a. The Gaussian Blur smooths the image and reduces high-frequency noise, making it particularly useful in edge detection algorithms where detecting sharp changes in intensity is crucial. Without noise reduction, edge detection might produce spurious results due to random variations in pixel intensity, which are often caused by noise.
    b. By using the Gaussian Blur to remove noise, we can achieve more accurate and stable results.

b. Edge Detection

i. Sobel Filter

1. The Sobel Filter is a discrete differentiation operator that computes an approximation of the gradient of the image intensity function. It uses convolution with two kernels to determine the intensity gradients in the horizontal (Gx) and vertical (Gy) directions. These gradients are then combined to calculate the overall gradient magnitude, which highlights edges in the image.

2. Mathematical Functions
    a. The Sobel Filter uses two 3X3 convolution kernels, one for detecting horizontal changes (Gx) and the other for vertical changes (Gy):

    b. Horizontal Kernel

    i.
$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

Sobel Horizontal Kernel (Fig. 3.2.1)

    c. Vertical Kernel

    i.
$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Vertical Horizontal Kernel (Fig. 3.2.2)

3. Gradient Magnitude and Direction
    a. After computing Gx (Fig 3.2.1) and Gy (Fig 3.2.2), the magnitude of the gradient is calculated to measure the strength of the edge at each pixel. The gradient magnitude is given by:

i. $$G = \sqrt{G_x^2 + G_y^2}$$

Gradient Magnitude (Eq 3.2.3)

4. **Steps in Applying the Sobel Filter**
   a. **Preprocessing**
      i. If necessary, apply noise reduction techniques like Gaussian Blur to smooth the image and remove high-frequency noise that could interfere with edge detection.

   b. Convolution:
      i. Convolve the image with the Sobel kernels (Gx and Gy) [Eq 3.2.3] to compute the horizontal and vertical gradients.

   c. Gradient Calculation:
      i. Compute the gradient magnitude (G) [Eq 3.2.3] and, if needed, the gradient direction (θ).

   d. Thresholding (Optional):
      i. Apply a threshold to G [Eq 3.2.3] to retain only significant edges and suppress weak gradients.

5. **Importance**
   a. The Sobel Filter is computationally efficient and robust against noise compared to simple derivative operators.

ii. Canny Edge Detection

1. The primary goal of the Canny algorithm is to detect edges by finding regions of rapid intensity change in an image. This is achieved while ensuring:
   a. **Low Error Rate:** All significant edges should be detected.
   b. **Good Localization:** Detected edges should be as close as possible to the true edges.

    c. Single Response: Each edge should be detected only once.

2. **Steps in the Canny Edge Detection Algorithm**
    a. The Canny Edge Detection process involves five key steps:
        i. **Noise Reduction**
            1. Noise in an image can lead to false edge detection. To mitigate this, a Gaussian filter is applied to smooth the image and reduce high-frequency noise. The Gaussian kernel is defined as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Gaussian Kernel (Eq. 3.2.4)

        ii. **Gradient Calculation**
            1. The gradient intensity and direction are calculated using partial derivatives of the image. This is often achieved using Sobel filters.

        iii. **Non-Maximum Suppression**
            1. Non-maximum suppression is performed to thin out the edges and retain only the local maxima of the gradient magnitude. For each pixel, the algorithm examines its neighbors along the gradient direction.

        iv. **Double Thresholding**
            1. To identify strong and weak edges, the gradient magnitude is subjected to two thresholds:
                a. High Threshold: Pixels with gradient magnitude above the

high threshold are marked as strong edges.

b. Low Threshold: Pixels with gradient magnitude between the low and high thresholds are marked as weak edges.

2. **Edge Tracking by Hysteresis**
   a. In this step, weak edges that are connected to strong edges are retained, while the rest are discarded. This ensures that only edges that are part of continuous boundaries are detected.

3. Advantages of Canny Edge Detection

a. Noise Robustness: By smoothing the image, it reduces the impact of noise.

b. Accurate Edge Localization: Combines gradient direction and magnitude to pinpoint edge locations.

c. Single Response to Edges: Reduces the occurrence of false positives and double edges.

c. Hough Transform

i. The Hough Transform is a feature extraction technique used in image analysis, computer vision, and digital image processing to detect geometric shapes, particularly lines, circles, and ellipses. Originally introduced by Paul Hough in 1962, it has since been extended to various applications. The primary advantage of the Hough Transform is its robustness to noise and discontinuities in edges, making it effective for detecting shapes even in complex and noisy images.

ii. **Purpose and Overview**
   1. The Hough Transform maps points in the image space to a parameter space where potential geometric shapes can be identified through an accumulation process. For line

detection, it relies on the parametric representation of a line and converts it into a voting process in the parameter space.

$$r = x\cos\theta + y\sin\theta$$

Polar Representation (Eq. 3.3.1)

iii. **Steps in Hough Transform**

1. Edge Detection
    a. Before applying the Hough Transform, edges in the image are detected using techniques such as Sobel, Prewitt, or Canny edge detection. The edge detection step ensures that only significant boundaries are considered.

2. Mapping to Parameter Space
    a. For each edge point (x, y) in the image space, the corresponding r and θ values are calculated using Eq. 3.3.1. These values are mapped into a 2D accumulator array, often referred to as the Hough Space.
        i. The range of θ is typically chosen between 0 and 180◦ (or 0 and π radians).
        ii. The r range depends on the image dimensions, spanning [−d, +d] where d= sqrt {w^2 + h^2} d , and w, h are the image width and height.

3. Accumulator Array Voting
    a. Each edge point votes for all possible (r, θ) pairs that satisfy Eq. 3.3.1. This creates a peak in the accumulator array at (r, θ) corresponding to the line parameters.

4. Detecting Pekas in Hough Space
    a. The peaks in the accumulator array represent potential lines in the image. These peaks are detected using thresholding or non-maximum suppression to ensure only significant lines are identified.

iv. **Mathematical Representation**

1. Parameterization of a Line with using Eq 3.3.1.

2. Hough Space Mapping
    a. Each edge point (x, y) contributes to the accumulator array by voting for all (r, θ) combinations.

3. Peak Detection
    a. A peak in the accumulator array indicates a potential line in the image space.

$$A(r, \theta) \geq T$$

Peak Detection (Eq. 3.3.2)

    b. Where T is a threshold value.

v. **Advantages of Hough Transform**

1. Robustness to Noise
    a. It performs well in noisy images or with incomplete edge data.

2. Flexibility
    a. Can be extended to detect other shapes, such as circles or ellipses, by modifying the parameterization.

3. Global Detection
    a. Finds lines across the entire image, making it suitable for complex scenes.

d. Region of Interest (ROI)

i. The Region of Interest (ROI) is a fundamental concept in image processing and computer vision, focusing analysis on a specific area within an image. By isolating the ROI, computational resources are used more efficiently, and noise from irrelevant regions is minimized. This step is especially critical in applications such as edge detection, object tracking, and feature extraction, where processing the entire image may introduce unnecessary complexity and noise.

ii. **Purpose and Overview**
   1. The primary objective of defining an ROI is to restrict the analysis to regions of the image that are most relevant to the problem at hand. This reduces computational load and increases accuracy by ignoring irrelevant areas.

   2. For a 2D image (x, y), the ROI can be expressed as a subset of pixel coordinates:

$$\mathrm{ROI} = \{(x, y) \mid x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$$

ROI Equation (Eq. 3.4.1)

   3. Where?
      a. x1, x2: Horizontal boundaries of the ROI.
      b. y1, y2: Vertical boundaries of the ROI.

iii. **Steps in Defining and Using ROI**
   1. Defining the ROI
      a. ROIs can be defined in various ways, including:
         i. **Rectangular ROI**:
            1. A rectangle with specific dimensions and location.

         ii. **Polygonal ROI**:
            1. A custom-shaped area defined by a series of vertices.

         iii. **Circular ROI**:
            1. A region centered at a specific point with a defined radius.

$$M(x, y) = \begin{cases} 1 & \text{if } x_1 \leq x \leq x_2 \text{ and } y_1 \leq y \leq y_2, \\ 0 & \text{otherwise.} \end{cases}$$

Rectangular ROI (Eq. 3.4.2)

b. Applying the ROI

$$I_{\text{ROI}}(x, y) = I(x, y) \cdot M(x, y)$$

Filtered Image ROI (Eq. 3.4.3)

i. The mask M (x, y) is applied to the original image I (x, y) resulting in a filtered image IROI (x, y) [Eq. 3.4.3]

ii. This operation ensures that only the pixels within the ROI are retained for further processing.

iv. Mathematical Representation

1. Mask Application using Eq. 3.4.2.

2. Filtering Image with using Eq. 3.4.3.

$$\sum_{i=1}^{n} (x \cdot y_i - y \cdot x_i) \leq 0$$

Polygonal ROI (Eq 3.4.4)

3. Polygonal ROI (Line-Based Definition)
   a. A polygonal ROI can be mathematically described using Eq. 3.4.4. Where Xi and Yi are the vertices of the polygon.

v. Advantage of ROI

1. **Efficiency**:
   a. Reduces computational overhead by limiting processing to specific regions.

2. **Accuracy**:
   a. Eliminates irrelevant areas, improving detection and feature extraction.

3. **Scalability**:

a. Can be dynamically defined based on application requirements.

e. Line Detection

i. Purpose and Overview

1. Line detection identifies and represents linear features in an image mathematically. It often follows edge detection and uses algorithms like the Hough Transform to detect prominent lines.

$$\rho = x \cdot \cos\theta + y \cdot \sin\theta$$

Parametrical Line Equation (Eq. 3.5.1)

2. For an edge-detected image I(edge)[x,y], a line can be defined parametrically as using equations 3.5.1. Where p is a perpendicular distance from the origin to the line. Q is a angle between the line and the horizontal axis.

ii. Steps in Line Detection

1. Preprocessing
    a. Before line detection, the input image is processed to enhance line-like structures and reduce noise:
        i. **Edge Detection**: Detect edges using Sobel, Canny, or other filters.

        ii. **ROI Definition**: Focus detection within a specific Region of Interest (ROI).

2. Hough Transform for Line Detection
    a. The Hough Transform maps edge pixels into a parameter space to identify line candidates.

3. Thresholding
    a. After populating the accumulator, a threshold is applied.

iii. Mathematical Representation

$$x = \rho \cos\theta - y \sin\theta$$

Pixel Cordinates (Eq. 3.5.2)

1. For a detected line in the image, the pixel coordinates can found be using Eq. 3.5.2. Where p and Q are the line parameters obtained from the Hough Transform.

iv. Advantages of Line Detection

1. Precision:
   a. Accurately identifies linear structures.

2. Robustness:
   a. Effective under varying lighting and environmental conditions.

3. Simplicity:
   a. Well-supported by efficient algorithms like the Hough Transform**.**

f. Fill Between Lines

i. Purpose and Overview

$$L_1 : \rho_1 = x \cdot \cos\theta_1 + y \cdot \sin\theta_1$$

L1 point (Eq. 3.6.1)

$$L_2 : \rho_2 = x \cdot \cos\theta_2 + y \cdot \sin\theta_2$$

L2 point (Eq. 3.6.2)

$$\min(\rho_1, \rho_2) \leq x \cdot \cos\theta + y \cdot \sin\theta \leq \max(\rho_1, \rho_2)$$

Filling pixels equation (Eq. 3.6.3)

1. The goal of this function is to identify and visually highlight the region enclosed by two or more lines within an image. This is achieved by interpolating the space between the lines and assigning pixel values within the defined region.

2. Given two lines L1 and L2, defined as Eq. 3.6.1 and Eq. 3.6.2.

3. The area between these lines is determined by filling the pixels (x, y) that satisfy the constraints as Eq 3.6.3.

ii. Algorithm

$$x_1(y) = \frac{\rho_1 - y \cdot \sin\theta_1}{\cos\theta_1}$$

Interpolate x1 (Eq. 3.6.4)

$$x_2(y) = \frac{\rho_2 - y \cdot \sin\theta_2}{\cos\theta_2}$$

Interpolate x2 (Eq. 3.6.5)

$$I(x,y) = \begin{cases} I_{\text{fill}}, & \text{if } x_1(y) \le x \le x_2(y) \\ I(x,y), & \text{otherwise} \end{cases}$$

Fill Pixel of I(x,y) (Eq 3.6.6)

1. Detect Lines
    a. Use an edge detection algorithm and the Hough Transform to detect two or more prominent lines L1 and L2.
2. Define Region of Interest
    a. Identify the polygon formed by the intersection of the detected lines within a specific region of interest (ROI). The endpoints of the lines in the image frame determine the bounds of this region.

3. Interpolate Points
    a. For each scanline (row) y, interpolate the x-coordinates of the intersections of the lines L1 and L2 with the scanline with using Eq. 3.6.4 and 3.6.5.
4. Fill the Pixels
    a. For each yyy, fill the pixels between x1(y) and x2(y)) with a specified value Ifill with using Eq. 3.6.6.

iii. Mathematical Representation of Filling

$$R(x, y) = \{(x, y) \mid \min(x_1(y), x_2(y)) \leq x \leq \max(x_1(y), x_2(y))\}$$

General Filling Between Lines Equation (Eq. 3.6.7)

1. Given a bounded region defined by x∈[xmin,xmax] and y∈[ymin,ymax], the filling function can be generalized as Eq. 3.6.7.

iv. Advantages

1. Highlighting Features:
    a. Enhances visual interpretation by emphasizing specific areas.

2. Flexibility:
    a. Can be applied to arbitrary line configurations.

3. Integration:
    a. Easily incorporated into pipelines for lane detection, segmentation, and other tasks.

4. Methodology

a. Without Canny Edge Detection

i. Preprocessing the Input Frame

1. The preprocessing stage refines the input grayscale frame by applying a **Region of Interest (ROI)** mask and enhancing contrast. The key steps include:

2. Defining a polygonal region representing the lanes (Equation 3.4.1).

3. Applying a mask to focus computations only on the ROI.

4. Contrast enhancement using percentile-based clipping to normalize pixel intensities in the ROI.

ii. Edge Detection using an Enhanced Sobel Operator

1. To detect edges without using the Canny algorithm, the Sobel operator is applied to the preprocessed image after a Gaussian smoothing step:

    a. **Gaussian Smoothing**: A convolution operation is performed using a 5×5 kernel to reduce noise while retaining structural details (Equation 3.1.1).

2. **Gradient Calculation**: Horizontal (Gx) and vertical (Gy) gradients are computed using Sobel kernels (Equations 3.2.1 and 3.2.2).

3. **Gradient Magnitude**: The final edge intensity is calculated using the gradient magnitude (Equation 3.2.3).

4. **Dynamic Thresholding**: A threshold is determined based on the mean and standard deviation of gradient magnitudes to filter noise.

iii. Hough Transform for Line Detection

   1. Using the edge-detected image, the Hough Transform identifies lines within the ROI:

      a. Accumulator space is parameterized by (ρ,θ), where ρoρ is the perpendicular distance from the origin, and θ is the angle (Equation 3.3.1).

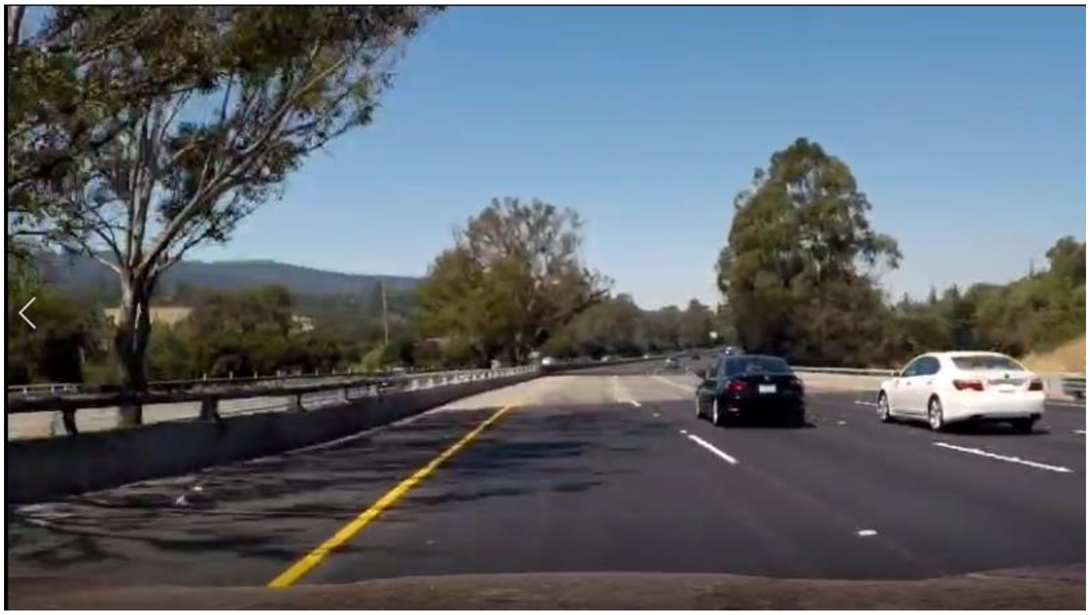
   2. The implementation optimizes the range of angles to focus on those commonly associated with road lanes

iv. Lane Grouping and Visualization

   1. Detected lines are grouped into left and right lanes based on their slopes, using outlier rejection and robust averaging to refine their positions. Finally, the region between the lanes is filled with a transparent color to visualize the detected area.

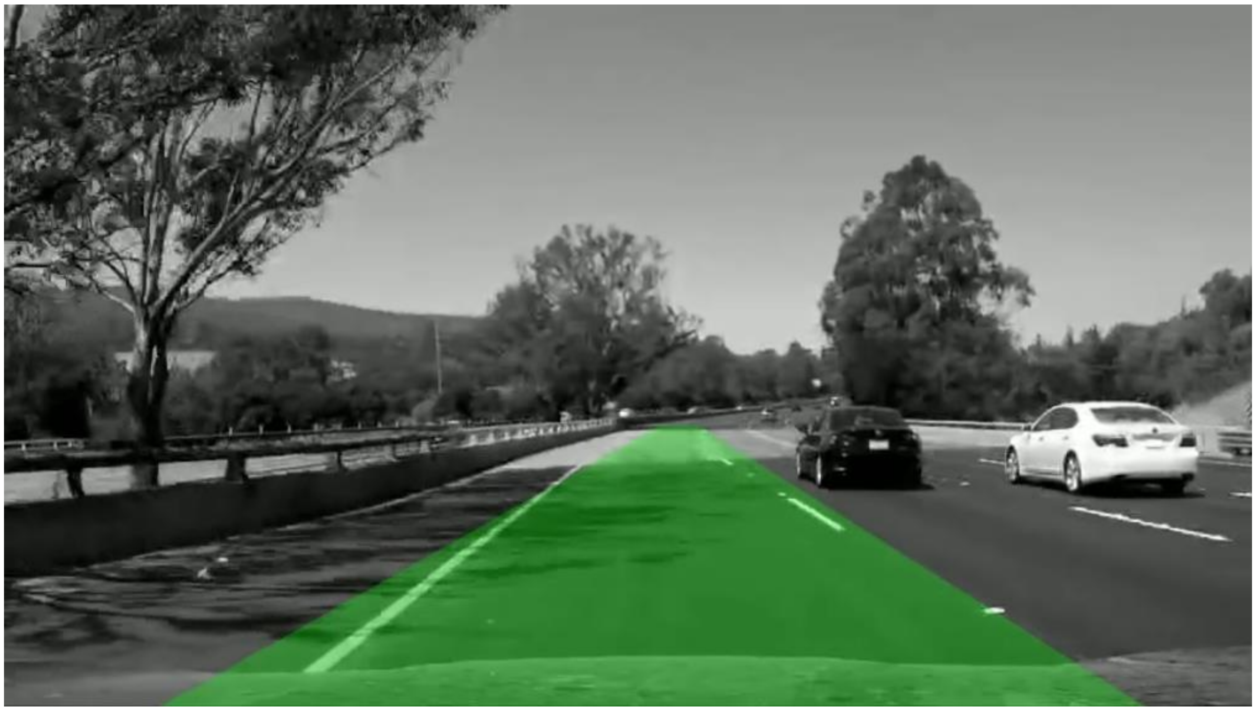
v. Test Case – 1

   1. Frame – 1
      a. Original Image



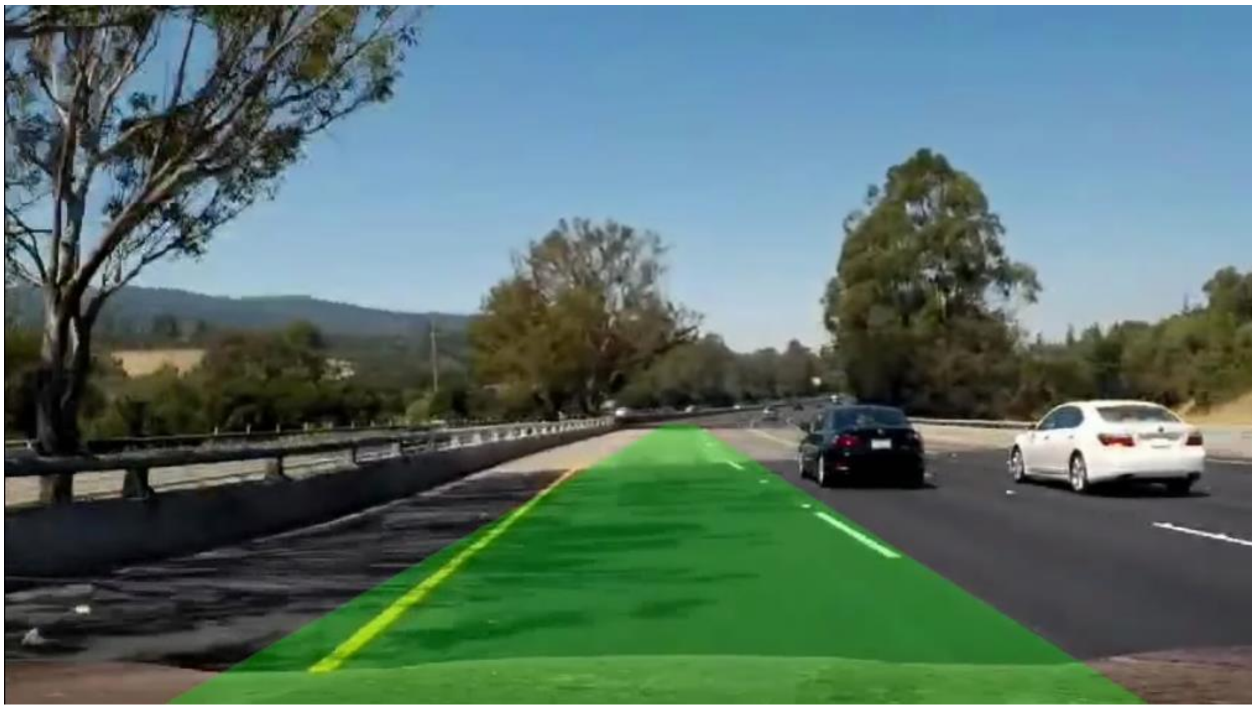
Original Road Image (Fig 4.2.1.1)

b. Processed Image



Processed GrayScale Road Image (Fig. 4.2.1.2)

a. Processed Image

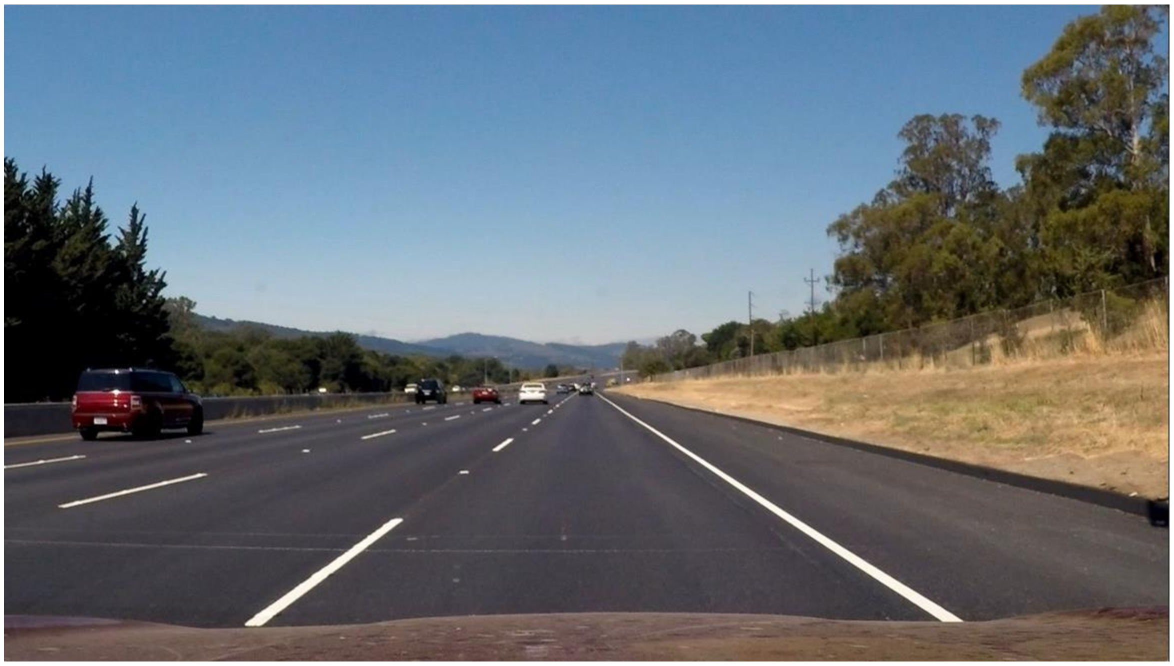


Processed Colored Road Image (Fig. 4.2.1.3)

b. The second image (fig. 4.2.1.2) demonstrates an advanced line detection algorithm that effectively identifies lanes on the road, even under shadowy conditions, by marking them with a green overlay. In contrast. This comparison highlights the robustness of the detection algorithm in enhancing visibility and maintaining accuracy in challenging lighting scenarios.

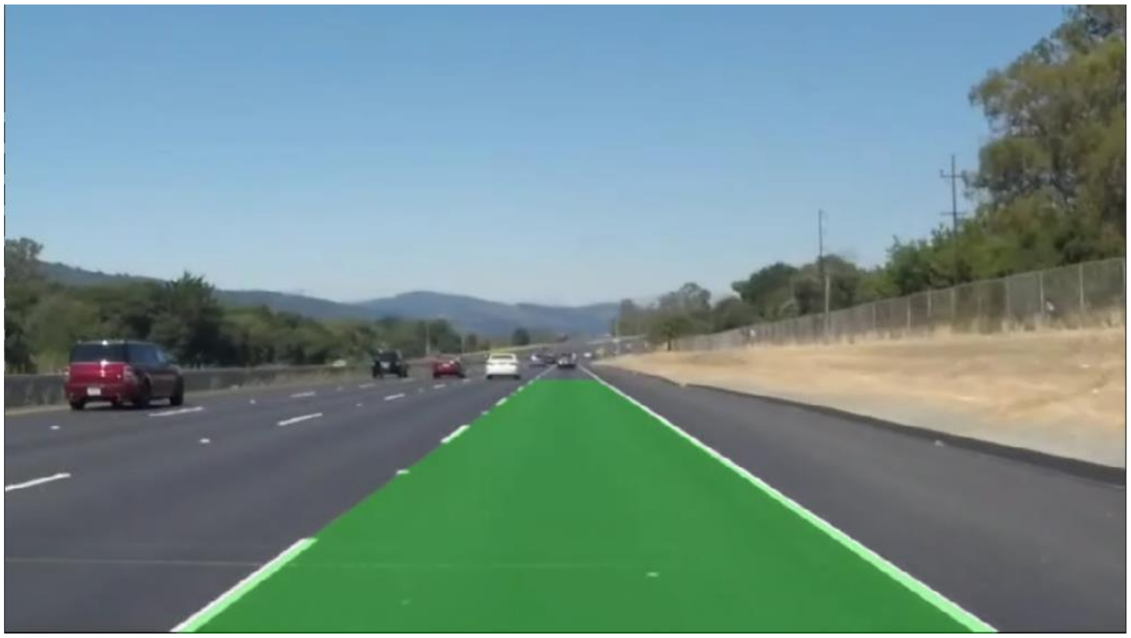
vi. Test Case – 2

1. Original Image



Original Road Image (Fig 4.2.2.1)

2. Processed Image


Processed Road Image (Fig 4.2.2.2)

b. Canny Edge Detection

1. Color Conversion to Grayscale

a. The cvtColor function performs a manual conversion of an RGB image to grayscale. This is done by calculating the grayscale intensity for each pixel.

b.

c. The function iterates over every pixel in the image, extracts its RGB components, and computes the corresponding grayscale value, which is then stored in a new grayscale image.

2. Gaussian Blur

a. Gaussian blur is applied through the gaussianBlur function. It utilizes a Gaussian kernel, which is calculated using the gaussian_kernel function. This kernel is applied to the image using convolution. The kernel is designed to smooth the image, reducing noise and details. The function implements padding and sliding window mechanisms to apply the kernel to every pixel in the image.

3. Region of Interest (ROI) Masking
    a. The region_of_interest function defines a polygonal mask that selects a specific region within the image. The mask is applied using a custom function, fill_poly_without_cv2, which fills the polygon area with a color value. The result is an image that retains only the region inside the polygon, effectively "masking" other regions.

4. Edge Detection Using Canny's Algorithm
    a. This steps are covered in 3.b.b.ii .

5. Line Detection Using Hough Transform
    a. The detect_lines function employs the Hough Line Transform to detect straight lines in an image. This method is particularly useful for detecting edges that correspond to significant geometric features, such as road markings or structural edges in images. It returns an array of detected lines.

6. Image Blending
    a. The add_weighted_images function is used to blend two images by assigning different weights to each and combining them. This is done by using the equation 4.2.1.

$$\text{Result Image} = \alpha \times \text{Image1} + \beta \times \text{Image2} + \gamma$$

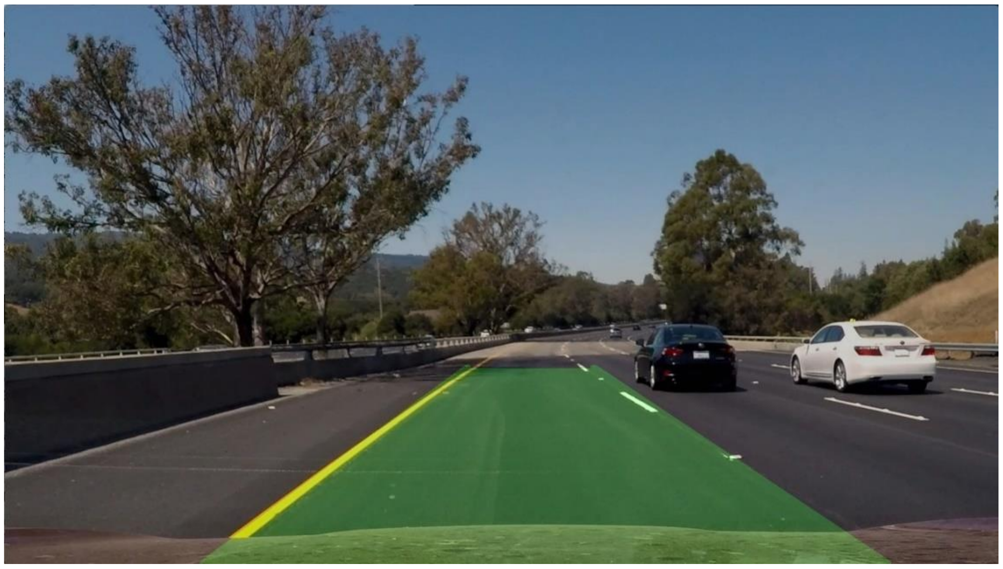
Image Blending (Eq. 4.2.1)

    b. The function ensures that the resulting image has pixel values in the range [0, 255] by clipping the values.

7. Extracting Frames from Video
    a. The extract_images_from_video function extracts frames from a video file at regular intervals. The frame rate is considered, and a specified number of frames per second are captured. This function allows for the extraction of still images from a video sequence, which is useful in scenarios where video data needs to be processed frame by frame.

8. Test Case – 1
    a. Original Image

    b. Processed Image



5. Results

a. Methodology Compression

i. The comparative analysis of the two line detection algorithms revealed distinct characteristics in terms of their methodological approaches and computational efficiency. The Canny edge-based algorithm demonstrated superior edge preservation and precise boundary detection through its multi-stage process. This approach effectively suppresses noise while maintaining essential structural information, resulting in a more refined detection output. However, this comprehensive processing pipeline introduces additional computational overhead compared to the non-Canny implementation.

ii. In contrast, the alternative algorithm, which operates without Canny edge detection, exhibited lighter computational requirements and faster processing times. This efficiency stems from its streamlined approach to line detection, employing direct geometric analysis methods without the preliminary edge detection stage. The reduction in processing steps contributes to a more resource-efficient implementation, particularly beneficial for real-time applications or systems with limited computational resources.

b. Methodology Accuracy

i. The accuracy assessment of both algorithms revealed complementary strengths and limitations that significantly impact their practical applications. The Canny edge-based detector demonstrated exceptional precision in controlled environments, consistently identifying fine line structures with high accuracy. This method excels in:

1. Precise boundary delineation

2. Accurate detection of subtle line features

3. Robust performance under optimal lighting conditions

4. Superior noise suppression in well-controlled environments

ii. However, the Canny-based approach showed notable sensitivity to environmental variations. Performance degradation was observed under:
    1. Variable lighting conditions
    2. Presence of shadows and artifacts
    3. Complex texture patterns
    4. Inconsistent image quality

iii. The non-Canny algorithm, while generally less precise in its line detection, exhibited remarkable adaptability across diverse operational conditions. This implementation demonstrated:
    1. Consistent performance across varying environmental conditions
    2. Robust detection capabilities in challenging lighting scenarios
    3. Higher tolerance to image quality variations
    4. More generalized line detection characteristics

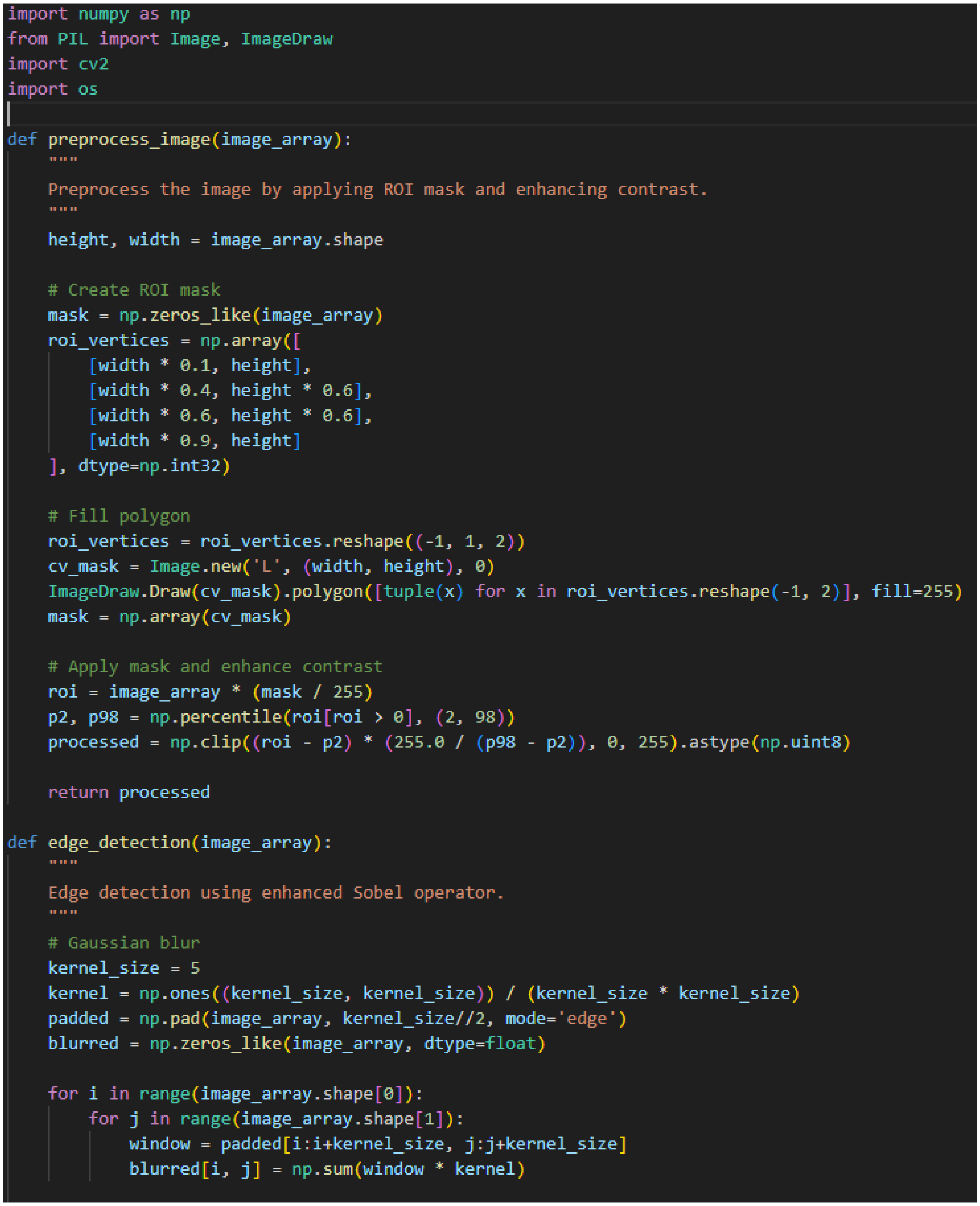
iv. Though this approach occasionally results in broader line detection boundaries compared to the Canny-based method, its stability across different operational conditions makes it particularly valuable for real-world applications where environmental control is limited.

v. The results indicate that each algorithm serves distinct use cases optimally. The Canny edge-based detector is more suitable for applications requiring high precision under controlled conditions, such as industrial quality control or medical imaging. Conversely, the non-Canny implementation proves more effective for general-purpose applications where environmental conditions are variable and robust operation takes precedence over absolute precision.

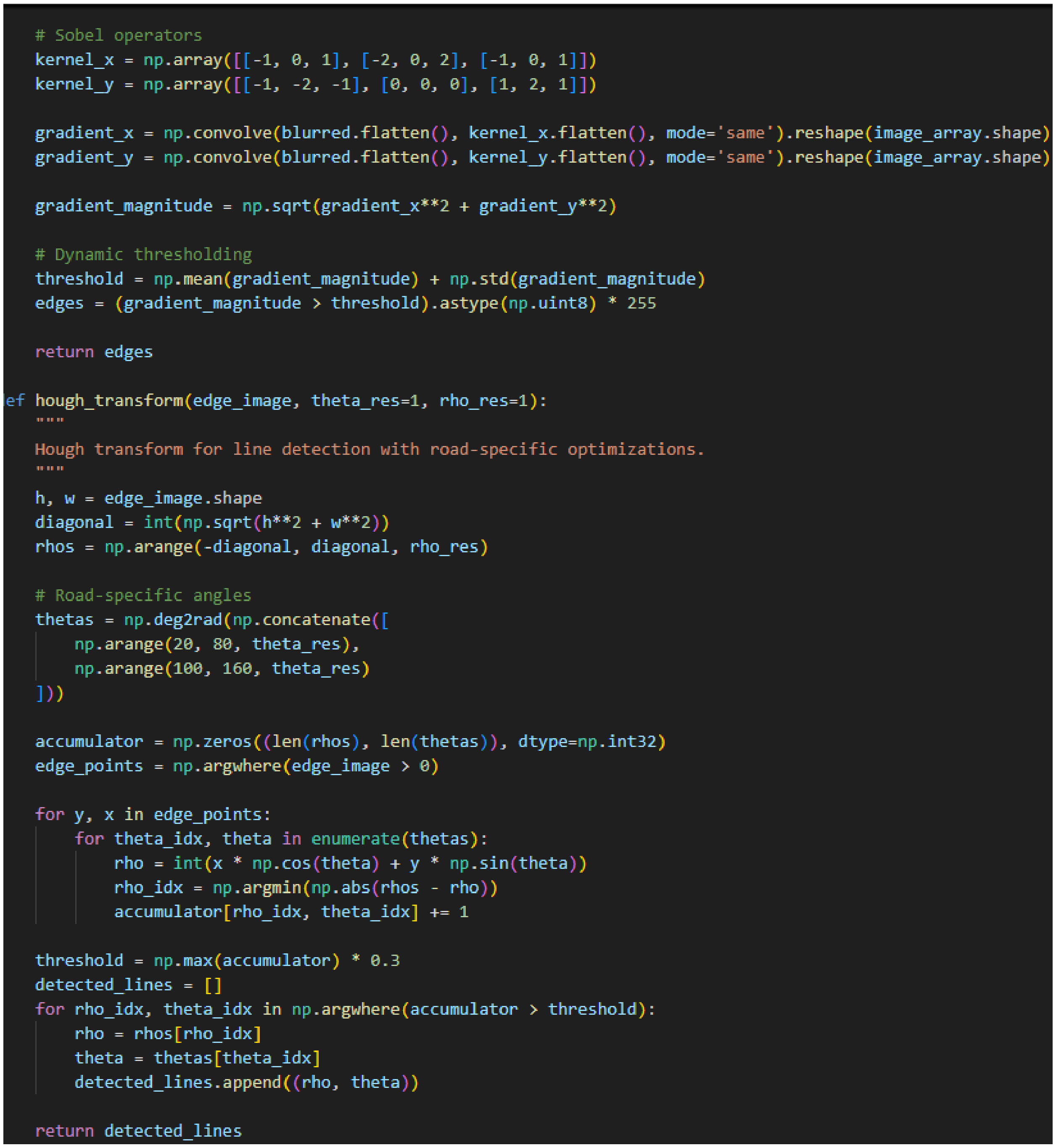
6. References

7. Appendix

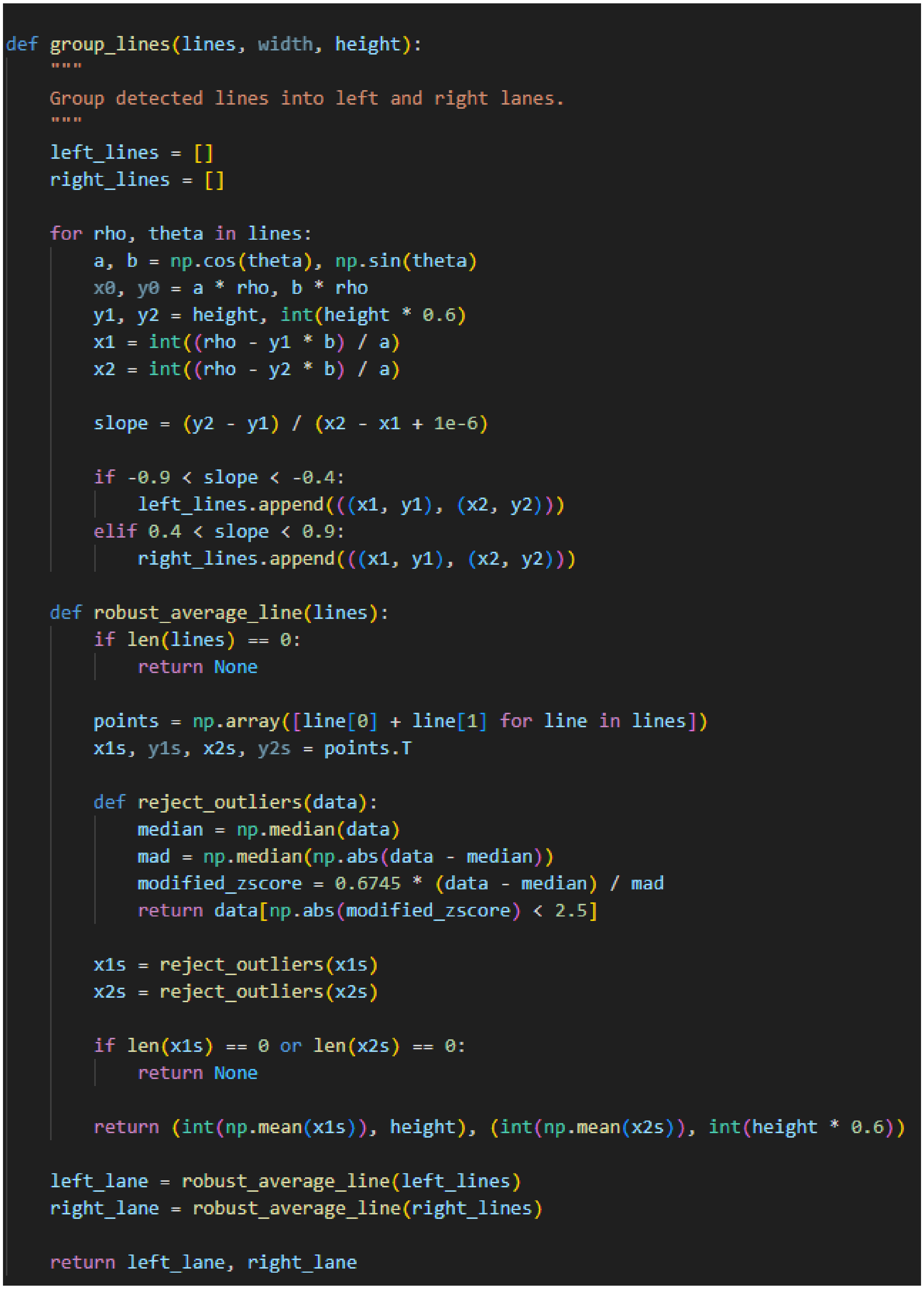
a. Non-Canny Edge Algorithm



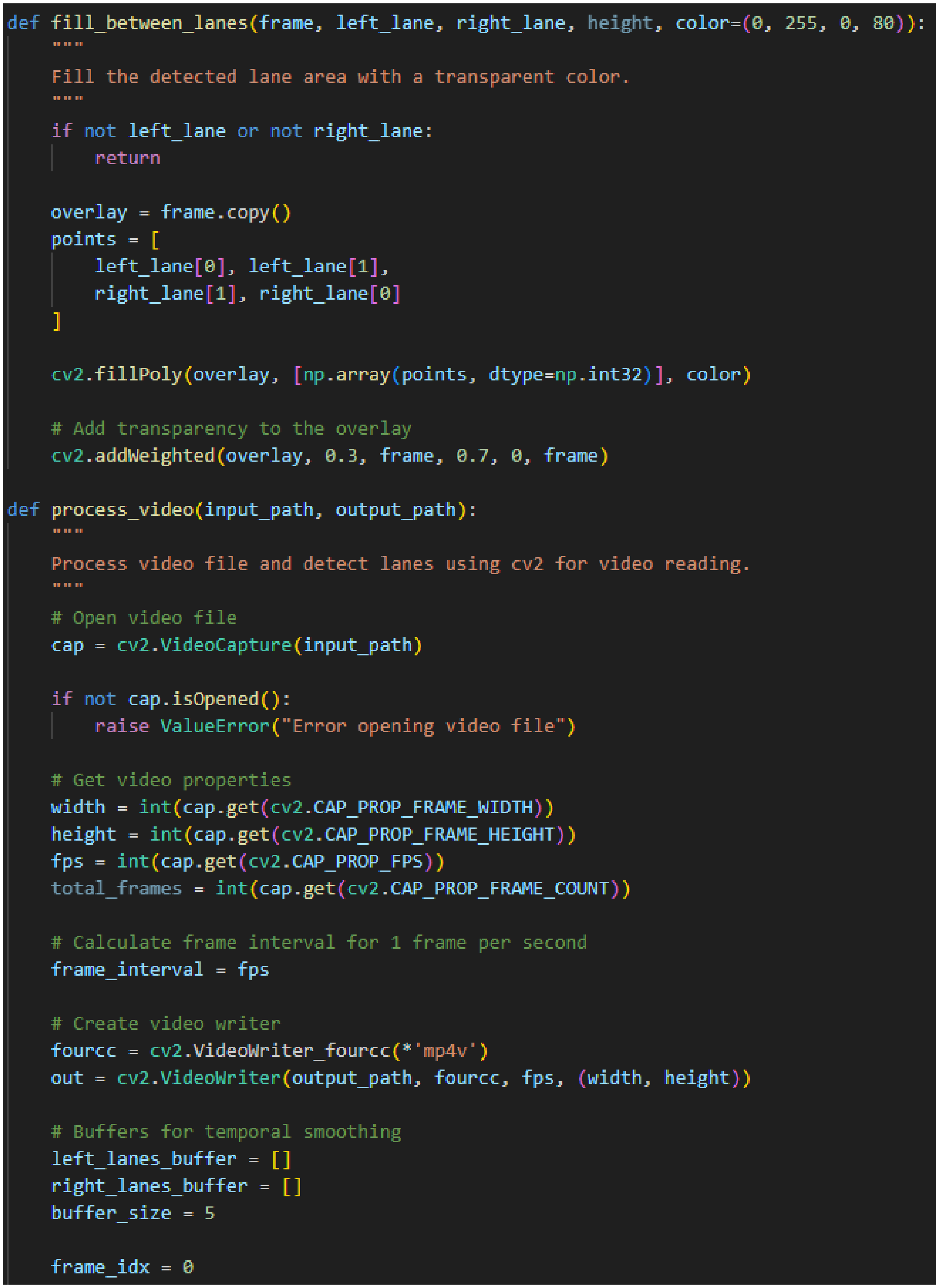
```python
import numpy as np
from PIL import Image, ImageDraw
import cv2
import os

def preprocess_image(image_array):
    """
    Preprocess the image by applying ROI mask and enhancing contrast.
    """
    height, width = image_array.shape

    # Create ROI mask
    mask = np.zeros_like(image_array)
    roi_vertices = np.array([
        [width * 0.1, height],
        [width * 0.4, height * 0.6],
        [width * 0.6, height * 0.6],
        [width * 0.9, height]
    ], dtype=np.int32)

    # Fill polygon
    roi_vertices = roi_vertices.reshape((-1, 1, 2))
    cv_mask = Image.new('L', (width, height), 0)
    ImageDraw.Draw(cv_mask).polygon([tuple(x) for x in roi_vertices.reshape(-1, 2)], fill=255)
    mask = np.array(cv_mask)

    # Apply mask and enhance contrast
    roi = image_array * (mask / 255)
    p2, p98 = np.percentile(roi[roi > 0], (2, 98))
    processed = np.clip((roi - p2) * (255.0 / (p98 - p2)), 0, 255).astype(np.uint8)

    return processed

def edge_detection(image_array):
    """
    Edge detection using enhanced Sobel operator.
    """
    # Gaussian blur
    kernel_size = 5
    kernel = np.ones((kernel_size, kernel_size)) / (kernel_size * kernel_size)
    padded = np.pad(image_array, kernel_size//2, mode='edge')
    blurred = np.zeros_like(image_array, dtype=float)

    for i in range(image_array.shape[0]):
        for j in range(image_array.shape[1]):
            window = padded[i:i+kernel_size, j:j+kernel_size]
            blurred[i, j] = np.sum(window * kernel)
```



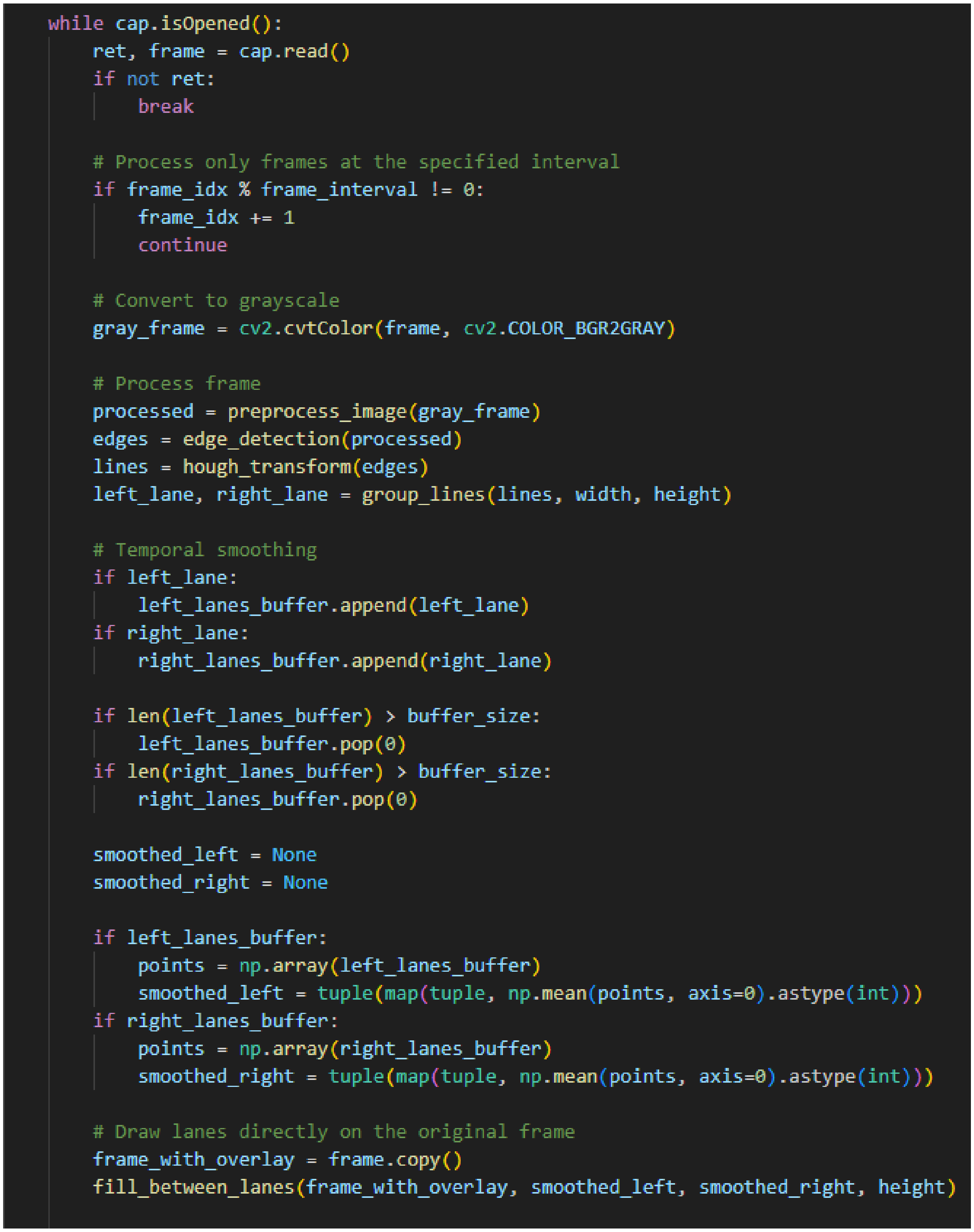
```
    # Sobel operators
    kernel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    kernel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

    gradient_x = np.convolve(blurred.flatten(), kernel_x.flatten(), mode='same').reshape(image_array.shape)
    gradient_y = np.convolve(blurred.flatten(), kernel_y.flatten(), mode='same').reshape(image_array.shape)

    gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)

    # Dynamic thresholding
    threshold = np.mean(gradient_magnitude) + np.std(gradient_magnitude)
    edges = (gradient_magnitude > threshold).astype(np.uint8) * 255

    return edges

ef hough_transform(edge_image, theta_res=1, rho_res=1):
    """
    Hough transform for line detection with road-specific optimizations.
    """
    h, w = edge_image.shape
    diagonal = int(np.sqrt(h**2 + w**2))
    rhos = np.arange(-diagonal, diagonal, rho_res)

    # Road-specific angles
    thetas = np.deg2rad(np.concatenate([
        np.arange(20, 80, theta_res),
        np.arange(100, 160, theta_res)
    ]))

    accumulator = np.zeros((len(rhos), len(thetas)), dtype=np.int32)
    edge_points = np.argwhere(edge_image > 0)

    for y, x in edge_points:
        for theta_idx, theta in enumerate(thetas):
            rho = int(x * np.cos(theta) + y * np.sin(theta))
            rho_idx = np.argmin(np.abs(rhos - rho))
            accumulator[rho_idx, theta_idx] += 1

    threshold = np.max(accumulator) * 0.3
    detected_lines = []
    for rho_idx, theta_idx in np.argwhere(accumulator > threshold):
        rho = rhos[rho_idx]
        theta = thetas[theta_idx]
        detected_lines.append((rho, theta))

    return detected_lines
```



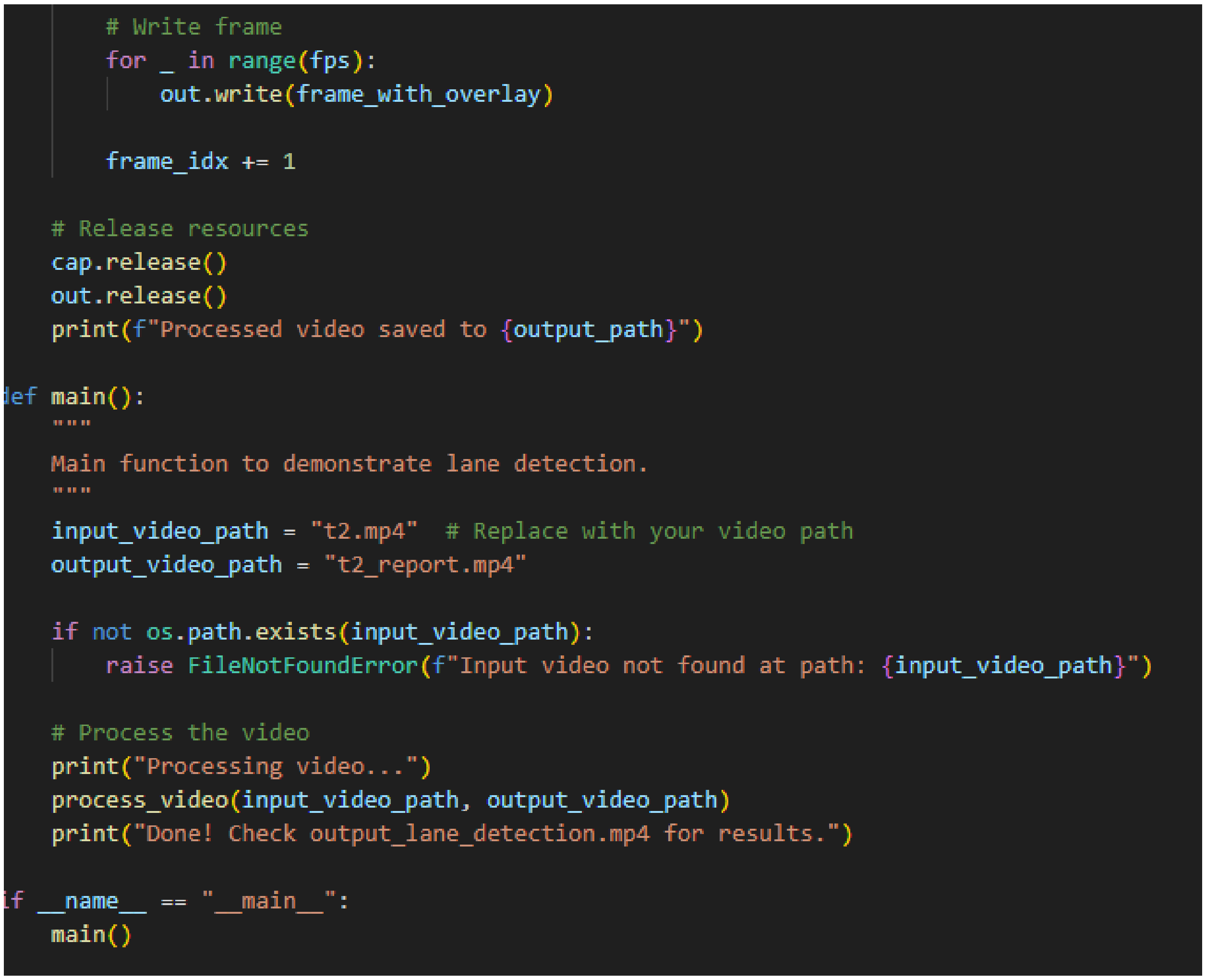
```python
def group_lines(lines, width, height):
    """
    Group detected lines into left and right lanes.
    """
    left_lines = []
    right_lines = []

    for rho, theta in lines:
        a, b = np.cos(theta), np.sin(theta)
        x0, y0 = a * rho, b * rho
        y1, y2 = height, int(height * 0.6)
        x1 = int((rho - y1 * b) / a)
        x2 = int((rho - y2 * b) / a)

        slope = (y2 - y1) / (x2 - x1 + 1e-6)

        if -0.9 < slope < -0.4:
            left_lines.append(((x1, y1), (x2, y2)))
        elif 0.4 < slope < 0.9:
            right_lines.append(((x1, y1), (x2, y2)))

    def robust_average_line(lines):
        if len(lines) == 0:
            return None

        points = np.array([line[0] + line[1] for line in lines])
        x1s, y1s, x2s, y2s = points.T

        def reject_outliers(data):
            median = np.median(data)
            mad = np.median(np.abs(data - median))
            modified_zscore = 0.6745 * (data - median) / mad
            return data[np.abs(modified_zscore) < 2.5]

        x1s = reject_outliers(x1s)
        x2s = reject_outliers(x2s)

        if len(x1s) == 0 or len(x2s) == 0:
            return None

        return (int(np.mean(x1s)), height), (int(np.mean(x2s)), int(height * 0.6))

    left_lane = robust_average_line(left_lines)
    right_lane = robust_average_line(right_lines)

    return left_lane, right_lane
```



```python
def fill_between_lanes(frame, left_lane, right_lane, height, color=(0, 255, 0, 80)):
    """
    Fill the detected lane area with a transparent color.
    """
    if not left_lane or not right_lane:
        return

    overlay = frame.copy()
    points = [
        left_lane[0], left_lane[1],
        right_lane[1], right_lane[0]
    ]

    cv2.fillPoly(overlay, [np.array(points, dtype=np.int32)], color)

    # Add transparency to the overlay
    cv2.addWeighted(overlay, 0.3, frame, 0.7, 0, frame)

def process_video(input_path, output_path):
    """
    Process video file and detect lanes using cv2 for video reading.
    """
    # Open video file
    cap = cv2.VideoCapture(input_path)

    if not cap.isOpened():
        raise ValueError("Error opening video file")

    # Get video properties
    width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    fps = int(cap.get(cv2.CAP_PROP_FPS))
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))

    # Calculate frame interval for 1 frame per second
    frame_interval = fps

    # Create video writer
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter(output_path, fourcc, fps, (width, height))

    # Buffers for temporal smoothing
    left_lanes_buffer = []
    right_lanes_buffer = []
    buffer_size = 5

    frame_idx = 0
```



```python
while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    # Process only frames at the specified interval
    if frame_idx % frame_interval != 0:
        frame_idx += 1
        continue

    # Convert to grayscale
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Process frame
    processed = preprocess_image(gray_frame)
    edges = edge_detection(processed)
    lines = hough_transform(edges)
    left_lane, right_lane = group_lines(lines, width, height)

    # Temporal smoothing
    if left_lane:
        left_lanes_buffer.append(left_lane)
    if right_lane:
        right_lanes_buffer.append(right_lane)

    if len(left_lanes_buffer) > buffer_size:
        left_lanes_buffer.pop(0)
    if len(right_lanes_buffer) > buffer_size:
        right_lanes_buffer.pop(0)

    smoothed_left = None
    smoothed_right = None

    if left_lanes_buffer:
        points = np.array(left_lanes_buffer)
        smoothed_left = tuple(map(tuple, np.mean(points, axis=0).astype(int)))
    if right_lanes_buffer:
        points = np.array(right_lanes_buffer)
        smoothed_right = tuple(map(tuple, np.mean(points, axis=0).astype(int)))

    # Draw lanes directly on the original frame
    frame_with_overlay = frame.copy()
    fill_between_lanes(frame_with_overlay, smoothed_left, smoothed_right, height)
```


```python
        # Write frame
        for _ in range(fps):
            out.write(frame_with_overlay)

        frame_idx += 1

    # Release resources
    cap.release()
    out.release()
    print(f"Processed video saved to {output_path}")

def main():
    """
    Main function to demonstrate lane detection.
    """
    input_video_path = "t2.mp4"  # Replace with your video path
    output_video_path = "t2_report.mp4"

    if not os.path.exists(input_video_path):
        raise FileNotFoundError(f"Input video not found at path: {input_video_path}")

    # Process the video
    print("Processing video...")
    process_video(input_video_path, output_video_path)
    print("Done! Check output_lane_detection.mp4 for results.")

if __name__ == "__main__":
    main()
```