

**Lines in Focus**

a custom vision pipeline using

Canny Algorithm and Hough transform

*Diana Dinca, gr. 30233*

**Cuprins**

[1. Overview 3](#_Toc198294781)

[2. Project Plan 3](#_Toc198294782)

[3. Grayscale transformation 4](#_Toc198294783)

[4. Canny Algorithm 4](#_Toc198294784)

[4.1. Kernel calculator 4](#_Toc198294785)

[4.2. Gaussian blur 5](#_Toc198294786)

[4.3. Image high-pass filtering 6](#_Toc198294787)

[4.4. Non-maxima suppression 7](#_Toc198294788)

[4.5. Hysteresis Thresholding 9](#_Toc198294789)

[5. Hough Algorithm 11](#_Toc198294790)

[5.1. Hough Transform 11](#_Toc198294791)

[5.2. Draw Lines 11](#_Toc198294792)

[5.3. Display Hough Space 12](#_Toc198294793)

[6. Source Code 13](#_Toc198294794)

[7. Testing 14](#_Toc198294795)

[Test 1 – Geometric Shapes 14](#_Toc198294796)

[Test 2 – Noisy Synthetic Road Scene 15](#_Toc198294797)

[Test 3 – Real-World Road Scene 17](#_Toc198294798)

[Test 4 – Road Markings 19](#_Toc198294799)

[8. Conclusion 20](#_Toc198294800)

## **1. Overview**

The goal of this project is to detect lines in grayscale images using image processing techniques. The intuition behind this approach is that many real-world images contain linear structures, such as road markings, building edges, or object contours, which can be effectively identified using mathematical techniques.

The desired outcome is to extract edges from an input using the Canny edge detector, then apply the Hough Transform to detect significant lines and visualize them over the original grayscale image. The implementation follows a structured pipeline: preprocessing the image, applying edge detection, transforming the detected edges into Hough space, identifying and filtering long lines, and finally overlaying the detected lines onto the original image.

This project has practical applications in various domains, including road sign recognition, lane detection in autonomous vehicles, and image analysis. By implementing this approach, we aim to develop an efficient and accurate method for detecting prominent lines in images.

## **2. Project Plan**

To achieve accurate line detection, we need to follow a structured plan. Each step contributes to preparing the image for edge detection and line extraction. The expected deliverables include a set of functions, each responsible for a specific task, ensuring modularity and ease of integration into the pipeline. These functions will handle preprocessing, algorithms, filtering, and visualization, leading to a clear representation of detected lines in the image.

1. **Preprocessing the Image**- we make sure that the input image is optimized for the next stage, and we convert the input image to grayscale.
2. **Edge Detection using Canny Algorithm**- this step provides the edge map required for the Hough Transform to detect lines.

The steps of the Canny edge detection method are:

1. Noise filtering through a Gaussian kernel;
2. Computing the gradient’s module and direction using Sobel operator;
3. Non-maxima suppression of the gradient’s module;
4. Edge linking through adaptive hysteresis thresholding.

Because choosing inappropriate threshold values may lead to too few or too many detected edges, we can use automatic threshold selection techniques or test with different thresholds.

1. **Hough Transform for Line Detection**- this step transforms edge points into a parameter space where lines can be easily identified

Apply the Hough Transform to detect straight lines in the edge-detected image.

Adjust parameters such as the accumulator threshold and line resolution to fine-tune results.

1. **Overlaying Detected Lines on the Original Image-** we draw the detected lines onto the original grayscale image for visualization.

Because some lines might not align perfectly with real-world edges due to pixel approximation errors, we could Improve the precision of detected lines by refining the Hough transform parameters.

1. **Displaying Hough Space Representation**- this step lets us visualize the Hough space accumulator matrix to demonstrate how the transformation captures image features.
2. **Testing and Evaluation**- we have to run the algorithm on multiple images with varying complexity to evaluate performance.

## **3. Grayscale transformation**

The input image consists of three primary colors: Red, Green, and Blue. I am using the function *rgb\_2\_grayscale* to convert the colored image into grayscale. This is done by iterating through each pixel, summing the three color components, and dividing the result by 3, to obtain the corresponding gray value.

O imagine care conține text, captură de ecran, Font, software

Conținutul generat de inteligența artificială poate fi incorect.

Fig.1 Grayscale conversion code

## **4. Canny Algorithm**

### 4.1. Kernel calculator

The *applyKernel* is a generic 2D convolution function that plays a central role in this project. It creates an empty result matrix of the same size as the input image, and by looping through each pixel (i, j) of the source image, it applies the kernel by multiplying its values with the corresponding neighborhood values in the image. The function handles image borders using pixel replication (to prevent the kernel from going out of bounds), and the result of the convolution is stored in the result image at position (i, j).

This function is used both for applying the Gaussian blur—to reduce noise before edge detection—and for applying Sobel filters, which compute the image gradients (edge directions and magnitudes).

O imagine care conține text, captură de ecran, software

Conținutul generat de inteligența artificială poate fi incorect.

Fig.2.1 Apply Kernel code

### 4.2. Gaussian blur

The *gaussian\_blur* function uses a Gaussian kernel to apply a blur to a grayscale input image, reducing noise while preserving edges. The function uses a 5x5 Gaussian kernel to smooth the image, which helps in eliminating high-frequency noise, but it ensures that edges remain more defined than with other blur methods.

This kernel has higher weights in the center and smaller weights for the outer pixels, reflecting the nature of Gaussian distribution where the central pixel has the highest influence. The sum of all the kernel values is calculated and used to normalize the kernel values for the convolution, because we want the kernel to preserve the local average of the pixels and not amplify the image.

O imagine care conține text, captură de ecran, software

Conținutul generat de inteligența artificială poate fi incorect.

Fig.2.2 Grayscale code

O imagine care conține captură de ecran, astronomie, saturn

Conținutul generat de inteligența artificială poate fi incorect.

Fig2.3 Grayscale Image Fig 2.4 Blurred Image

### 4.3. Image high-pass filtering

The *high\_pass\_filter* function processes the grayscale blured input image and applies a Sobel filter to detect edges by calculating the gradient magnitude in both the horizontal and vertical directions. It uses convolution with two 3x3 kernels to highlight high-frequency components and suppress low-frequency ones (smooth regions).

The sobelX detects vertical edges, while sobelY detects horizontal edges.

O imagine care conține text, captură de ecran, software

Conținutul generat de inteligența artificială poate fi incorect.

Fig.3 Sobel Filter code

The function calculates the weighted sums of the neighboring pixel values using the kernels and the algorithm implemented in the function *applyKernel*. This convolution operation essentially detects how much the pixel intensity changes in both horizontal and vertical directions, which are used to compute the edge gradient. This gives the intensity of the edge at that particular pixel. The greater the magnitude, the stronger the edge.

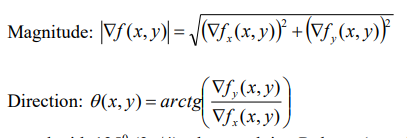


Fig.3.1 Magnitude and Gradient formula

### 4.4. Non-maxima suppression

After calculating the edge gradient magnitudes using the Sobel filter, we apply non-maxima suppression to thin the edges and preserve only the most relevant ones. This step is essential for producing a clear edge map before proceeding to the hysteresis thresholding stage.

The algorithm compares the magnitude of each pixel to the magnitude of its neighbors along the gradient direction. If the pixel is not the local maximum, it is suppressed (set to 0). This results in thinner, more defined edges and reduces the likelihood of detecting false or noisy lines.

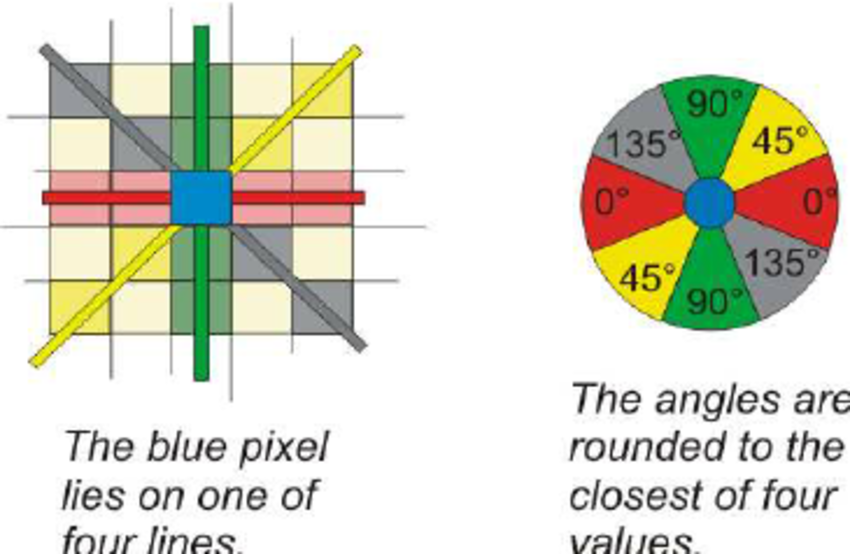


Fig.4 Angles for Suppression

To do this, the *non\_max\_suppression* function loops through each pixel (excluding the borders), reads its gradient direction, and maps it to one of four major angles: 0° (check left/right), 45° (check up-right/down-left), 90° (check up/down), or 135° (check up-left/down-right). These directions represent the possible orientations of edges in the image.

For each pixel, the function identifies the two neighboring pixels along the gradient direction and compares their intensity values (from the gradient magnitude). If the current pixel has a greater or equal value than both neighbors, it is considered a local maximum and is kept; otherwise, it is suppressed (set to 0).

O imagine care conține text, captură de ecran, software

Conținutul generat de inteligența artificială poate fi incorect.

Fig.4.1 Non-Maxima Suppresion code

O imagine care conține nevertebrate, ctenophora, captură de ecran

Conținutul generat de inteligența artificială poate fi incorect.

Fig.4.2 Gradient Magnitude normalized Fig 4.3. Non-Maxima Suppresion

### 4.5. Hysteresis Thresholding

The *hysteresis\_thresholding* function is responsible for finalizing the Canny Algorithm. It works by classifying pixels based on their gradient magnitude and linking edge segments.

This method converts the input (assumed to be the gradient magnitude, usually in CV\_32FC1) into an 8-bit grayscale image (CV\_8UC1), scaling values to the 0–255 range, builds a histogram of non-zero pixel intensities, and uses two threshold values: a low threshold and a high threshold.

O imagine care conține text, captură de ecran, Font

Conținutul generat de inteligența artificială poate fi incorect.

Fig.5 Threshlod code

Pixels with gradient magnitudes above the high threshold are immediately considered strong edges, while those below the low threshold are discarded as non-edges. Pixels with values between the two thresholds are treated as weak edges and are only retained if they are connected to a strong edge in their 8-neighborhood.

O imagine care conține text, captură de ecran, software

Conținutul generat de inteligența artificială poate fi incorect.

Fig.5.1 Edge clasification code

O imagine care conține captură de ecran

Conținutul generat de inteligența artificială poate fi incorect.

Fig.5.2 Final edges after Adaptive Binarization and Hysteresis

## **5. Hough Algorithm**

### 5.1. Hough Transform

The *hough\_transform* function implements the classical Hough Transform algorithm for detecting straight lines in a binary edge image. It works by mapping each white pixel (value 255) from the image space (x, y) into the Hough parameter space (ρ, θ), where each point votes for all possible lines it could belong to. In Hough Transform a line is represened in polar form: ρ=x⋅cos(θ)+y⋅sin(θ), where ρ is the distance between the origin of the image to the selected line, and θ is an angle in the range of [0°, 180°].

The function creates a 2D accumulator matrix where each cell represents a potential line with parameters (ρ, θ). For each edge pixel, the algorithm loops over a range of θ values, computes the corresponding ρ using the polar equation of a line, and increments the accumulator cell that corresponds to (ρ, θ). To handle negative ρ values, an offset (max\_rho) is added.

After filling the accumulator, the function scans it for cells with a number of votes above a defined threshold. These cells represent lines that are strongly supported by edge pixels.

O imagine care conține text, captură de ecran, software, Software multimedia

Conținutul generat de inteligența artificială poate fi incorect.

Fig.6 Threshlod code

### 5.2. Draw Lines

The *draw\_hough\_lines* function is responsible for visualizing the detected lines on the original image. It receives a list of (ρ, θ) pairs and the image on which the lines should be drawn.

For each line, the function converts the polar coordinates (ρ, θ) back into Cartesian points that can be used to draw the line in image space. Using basic trigonometry, it calculates two points located far enough in both directions along the line (to ensure it spans the image), and draws a red line using OpenCV’s line() function.

O imagine care conține captură de ecran, Color, astronomie

Conținutul generat de inteligența artificială poate fi incorect.

Fig.6.1. Hough Transform

### 5.3. Display Hough Space

The *display\_hough\_space* function is responsible for visualizing the Hough accumulator matrix generated during the Hough Transform process. This matrix, known as the accumulator, contains vote counts for each combination of distance (ρ) and angle (θ) that represents possible lines in the image.

To make this data viewable as an image, the function first converts the accumulator values into an OpenCV Mat of type CV\_32F (float matrix), while simultaneously finding the maximum vote value in the matrix. This maximum is then used to normalize the entire matrix so that the strongest lines (with the most votes) appear brightest when displayed. The conversion to CV\_8U (8-bit unsigned) ensures that pixel values range from 0 to 255, which is suitable for standard grayscale image display.

O imagine care conține text, captură de ecran, software, Font

Conținutul generat de inteligența artificială poate fi incorect.

Fig. 7 Hough Space code

O imagine care conține captură de ecran

Conținutul generat de inteligența artificială poate fi incorect.

Fig.7.1 Hough Space Image

## **6. Source Code**

The complete source code for this project is available in a public GitHub repository: [*https://github.com/Diana-Dinca/Lines-In-Focus*](https://github.com/Diana-Dinca/Lines-In-Focus)

## **7. Testing**

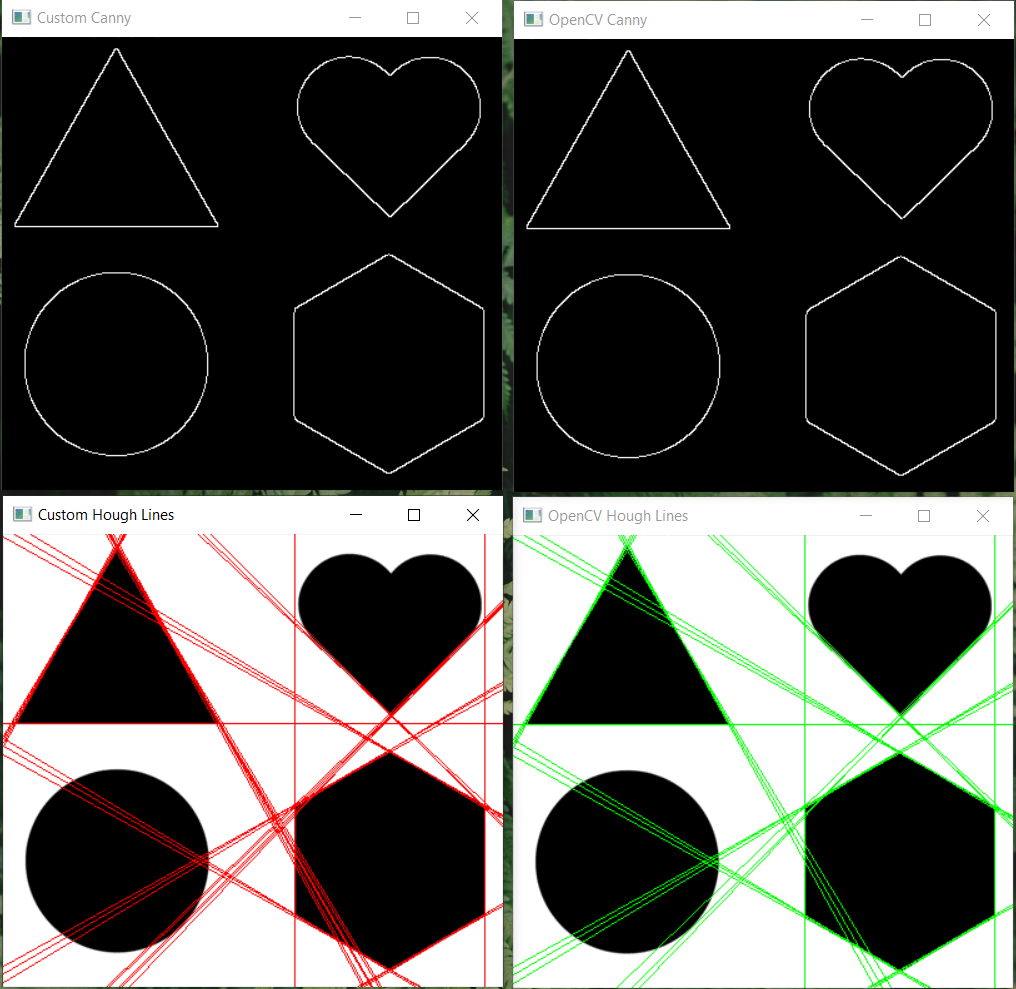
To evaluate the correctness and effectiveness of our custom implementations of the Canny edge detection and Hough transform algorithms, we conducted a comparative analysis against OpenCV’s built-in functions.

In order to ensure a fair and accurate comparison, we did not use arbitrary or hardcoded parameters for the OpenCV functions. Instead, we derived the thresholds for the OpenCV Canny function using the same histogram-based method we applied in our custom pipeline, and for the Hough transform, we used the same parameter values as in our custom method. Additionally, to support visual analysis, the results from both the custom and OpenCV methods were overlaid on the grayscale version of the input image, allowing us a direct side-by-side comparison.

Finally, for a detailed analysis, we measured the true positives (lines detected by both the custom and OpenCV implementations), false negatives (lines detected only by the custom implementation), and false positives (lines detected only by the OpenCV implementation). Based on these, we calculated the precision of the custom implementation and printed the results to the console.

### Test 1 – Geometric Shapes

This test compares the performance of the custom and OpenCV implementations of Canny edge detection and Hough transform on a synthetic image containing basic geometric shapes.



In the top row, the custom Canny implementation (left) and the OpenCV Canny (right) produce similar edge maps. In the bottom row, both implementations accurately detect and draw lines using the Hough transform. The evaluation metrics support the visual results, showing a precision of 100% with only 3 false negatives.

O imagine care conține text, Font, captură de ecran

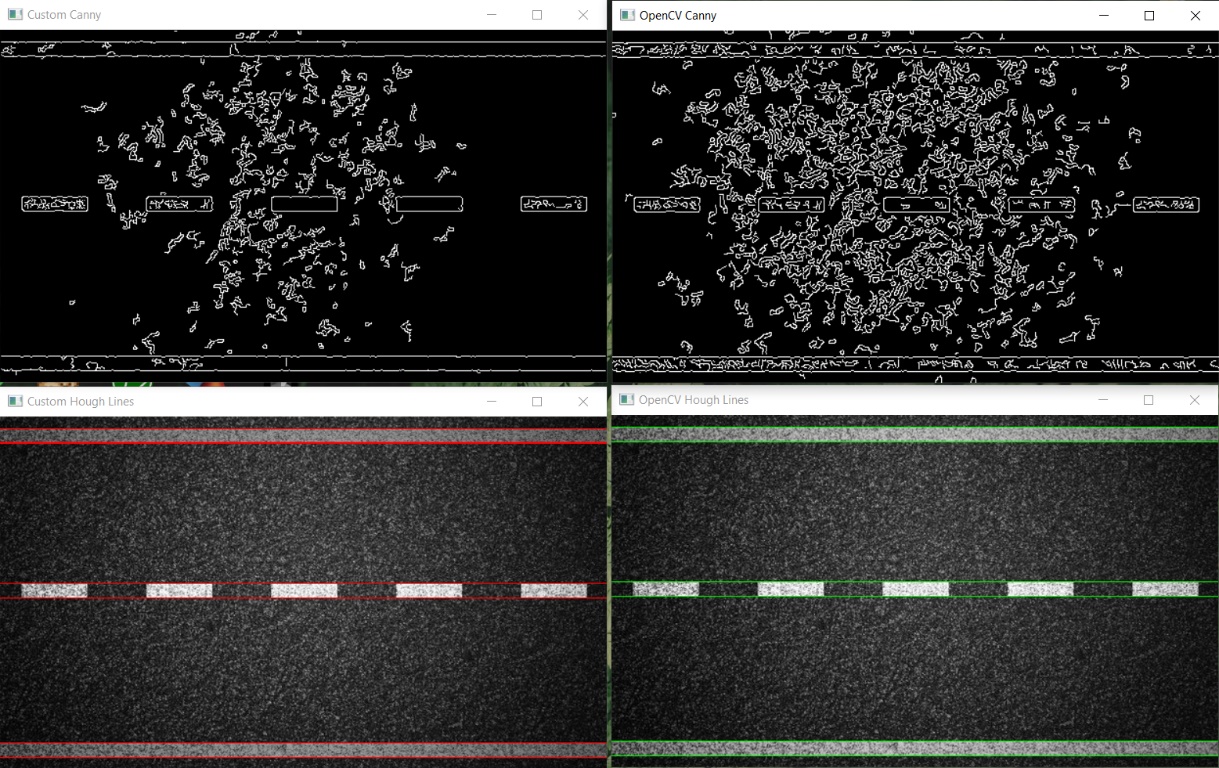
Conținutul generat de inteligența artificială poate fi incorect.

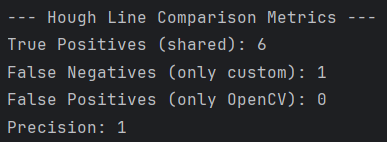
O imagine care conține captură de ecran, diagramă, Grafică, proiectare

Conținutul generat de inteligența artificială poate fi incorect.

### Test 2 – Noisy Synthetic Road Scene

This test evaluates the performance of both implementations on a challenging synthetic image: a textured, noisy background with lane-like dashed lines, simulating a noisy road environment.



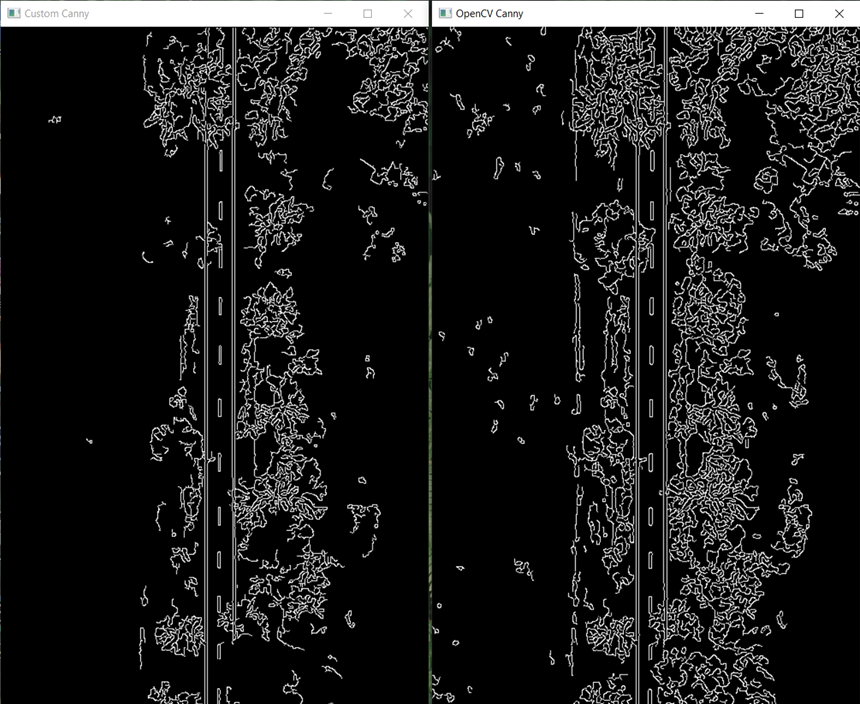
In the top row, the custom Canny implementation produces a noticeably cleaner edge map. It successfully isolates the dashed lines and horizontal boundaries with minimal noise amplification. The Hough line detection works similarly in both cases.

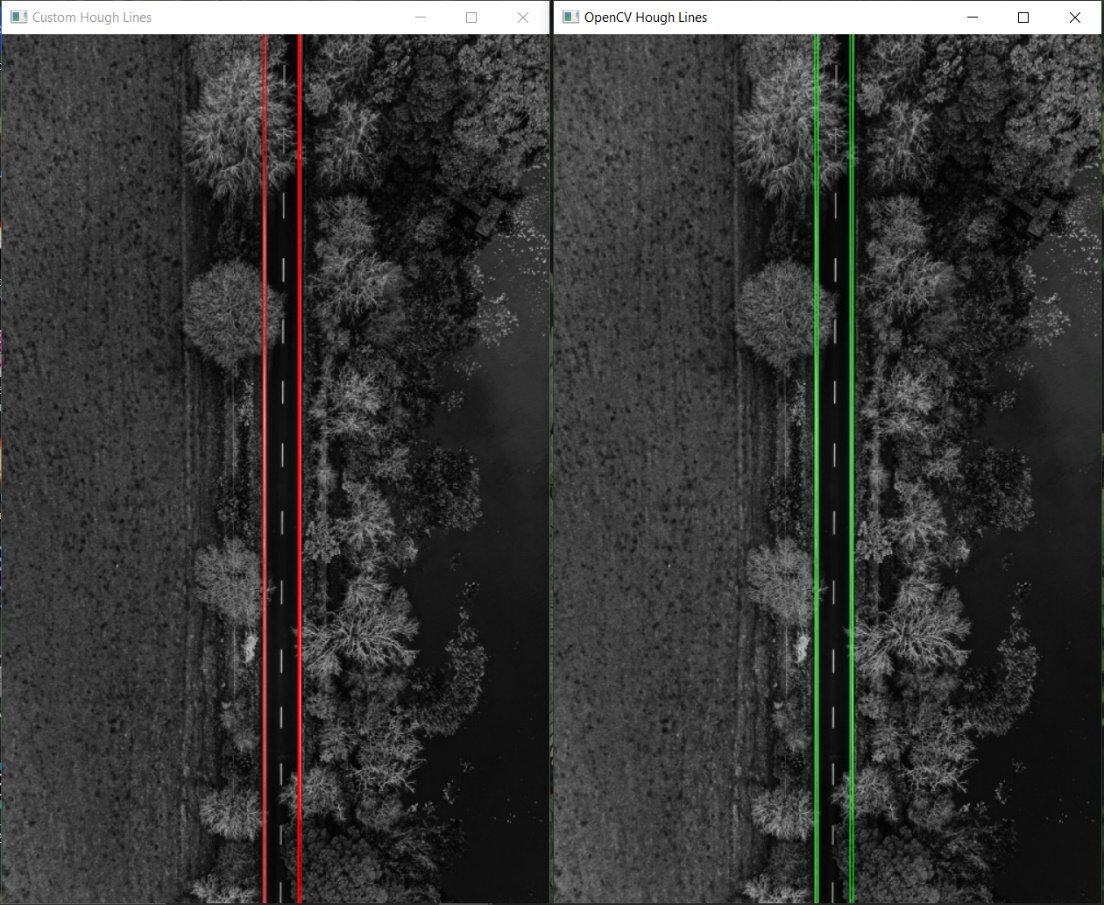
O imagine care conține captură de ecran, calculator, Dreptunghi, Software multimedia

Conținutul generat de inteligența artificială poate fi incorect.

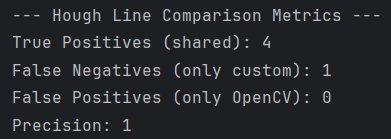
### Test 3 – Real-World Road Scene

This test uses a grayscale aerial image of a road surrounded by natural textures (trees, fields, and water) to assess the robustness of both the Canny edge detection and Hough line transform in real-world conditions.





In the top row, the custom version produces cleaner edges, with clearer lane markings and better separation from the surrounding vegetation, while the OpenCV’s version retains more fine texture and noise from the trees and road edges, which may not be relevant to structural detection but increase edge clutter. The Hough line transform detects the 2 main road lines with high precision and minimal dedundancy in both implementations.

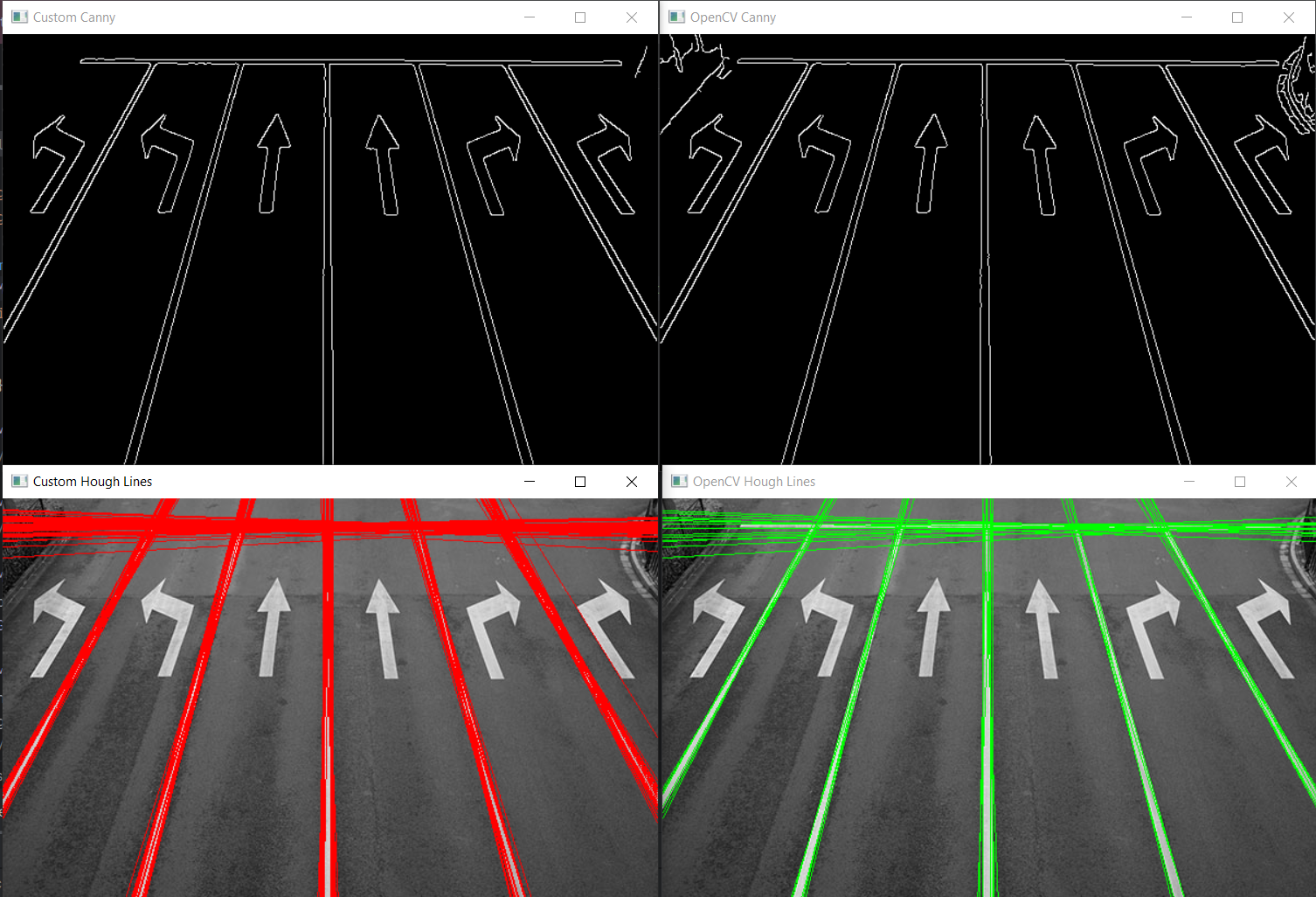


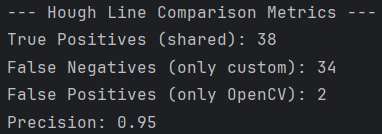
O imagine care conține captură de ecran, text, copac, în aer liber

Conținutul generat de inteligența artificială poate fi incorect.

### Test 4 – Road Markings

In this test, the image contains clean, high-contrast road markings including arrows and multiple straight lane dividers. It is a structurally rich but visually clean scenario.



In the top row, both Canny implemetations are accuretlly surprising the main lines, the custom implementation havig less details, it caputres the exential. In the bottom row, the custom implementation (left) detects many correct lines, but also introduces more false positives, including misaligned or duplicated lines, particularly in the upper area.

O imagine care conține captură de ecran, linie, Paralel, text

Conținutul generat de inteligența artificială poate fi incorect.

## **8. Conclusion**

The comparison revealed similar outputs between the custom implementations and OpenCV’s built-in functions. Our custom Canny edge detector produced cleaner results with fewer spurious edges, due to its more conservative non-maxima suppression and tailored hysteresis propagation. In contrast, OpenCV’s implementation, while faster and optimized for general-purpose use, was more permissive and retained more noise-like structures, particularly in high-detail regions.

Similarly, our custom Hough transform, while more computationally intensive, produces great results, but has edundant or closely spaced lines, which sometimes led to visual clutter.

Overall, the custom pipeline showed strong adaptability and interpretability, delivering results that were comparable in accuracy to OpenCV’s algorithms while offering greater flexibility and control over the intermediate processing steps. This makes it especially valuable in educational contexts, debugging, or applications that require fine-tuned edge and line detection beyond the scope of general-purpose implementations.