# Canny Lane Detection Algorithm

## Introduction

The purpose of this project is to give a computational approach to the Canny edge detector and the Hough line transform and use these to create a real-time lane-marking detection algorithm that can be used for autonomous driving and other machine vision applications. Edges are defined as rapid changes in the intensity of color values (which typically occur at object boundaries). The Canny edge detector is the process of identifying and locating these edges between different segments/objects in an image. Edge detectors are extremely useful as they serve to simplify the analysis of images by drastically reducing the amount of data that must be processed while preserving the structural information about the object's boundaries.

The Hough line transform is a technique used in computer vision and image processing to detect straight lines in an image. It works by representing these straight lines as points within its own space (called the Hough Space). These classified straight lines can then be determined to be the boundaries of lanes within a road. Using the Canny edge detector and the Hough line transform, we can find the boundaries of the road at any given moment, and thus determine whether the driver is within their lane or not.

## Background

### Canny Edge Detector

In typical Canny edge detectors, there are five main steps used to detect the edges of an image. The first step is to reduce the noise in the image, then the gradient magnitude and direction must be calculated. From there, non-maximum suppression must be performed before double thresholding and finally wrapping up the edge-detecting process with edge-tracking hysteresis. All these steps must be performed in this order for an effective and concise output.

### Gaussian Noise Reduction

One of Canny's biggest flaws is that the algorithm is sensitive to image noise [4]. This is because the Canny algorithm is a first derivative operator that relies on the gradient calculation of the image to locate local maxima (edges) in pixel intensity. This noise can lead to the detection of false edges (if local maxima are grouped) and thus false shape classifications. Therefore, it's essential that the first step of edge-detection would be to effectively reduce the amount of noise in the image. To do this, a Gaussian blur can be used to smooth the image by convolving the image with a Gaussian kernel. As seen in Equation 1 and Equation 2, the Gaussian kernel is defined as two square matrices (the size is user-inputted) that describe the vertical and horizontal distances from the origin of the kernel to each kernel's pixel location in the matrix.

$$x = \begin{matrix} -2 & -2 & -2 & -2 & -2 \\ -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{matrix}$$

*Equation 1: Gaussian Kernel in Horizontal Direction (x)*

$$y = \begin{matrix} -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \end{matrix}$$

*Equation 2: Gaussian Kernel in Vertical Direction (y)*

Once the Gaussian kernel has been calculated, the Gaussian filter must be defined. A filter is an operator that is applied to an image to enhance or extract information. As seen in Equation 3, the Gaussian filter is derived from the Gaussian impulse response which was determined to be the ideal operator.

$$G(x) = e^{\frac{-x^2}{2\sigma^2}}$$

*Equation 3: Gaussian Impulse Response*

The Gaussian filter was determined to be the optimal operator as it met the three edge-detection criteria the best. The first criterion is that the edge detector must have good detection, meaning there is a low probability of failing to mark real edge points and a low probability of falsely marking pseudo-edge points [2]. The first criterion can be measured by calculating the signal-to-noise ratio (SNR) using Equation 4.

$$SNR = \frac{\int_{-w}^{+w} G(-x) f(x) dx}{n_0 \int_{-w}^{+w} f^2(x)}$$

*Equation 4: First Criterion, SNR*

The numerator describes the signal of the image, and the denominator describes the noise of the image. *G(x)* represents the edge function (image), and *f(x)* is the Gaussian impulse response. The $n_o$ variable is the amplitude of the mean squared noise (average noise) per unit length and *±w* represents the bounds of the finite impulse response. As seen in Equation 5, the $n_o$ variable can be found by using the pixel intensity value $I$, the mean intensity value μ and the total number of pixels along the length $N$.

$$n_o = \frac{\sum_{i=i}^{N}(I_i - \mu)^2}{N}$$

*Equation 5: Mean Squared Noise per Unit Length*

The second criterion of the edge detector is that it must have good localization. This means that the points marked as edge points by the filter should be as close and accurate as possible to the center of the true edge of the image [1]. This criterion can also be measured using Equation 6.

$$Localization = \frac{\int_{-w}^{+w} G'(-x)f'(x)dx}{n_0 \int_{-w}^{+w} f'^2(x)}$$

*Equation 6: Second Criterion, Localization*

The final criterion is that there must only be one response to a single edge. As mentioned before, an edge is defined as a local maximum in the convolution between the image and the filter. If there are two or more responses to the same edge, then all but one of these responses would have been caused by unwanted noise. It is important to understand that an edge can only create one maximum when it is convolved with a filter [10]. This criterion will be handled using non-maximum suppression. Since we want to maximize the ratio of signal to noise as well as maximize the accuracy of the detected edges, the ideal filter (Gaussian filter) can be determined as the filter that maximizes the product of the first two criteria.

As seen in Equation 7, the Gaussian filter uses the Gaussian kernel along with a user-inputted standard deviation to calculate the weighted distribution of each element in the Gaussian kernel. The standard deviation is used to create a Gaussian distribution weighting system, meaning the closer the pixel is to the center of the matrix (the origin), the more weight it has in affecting the value of the pixel that is currently being processed.

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

*Equation 7: Gaussian Filter Equation*

Once the Gaussian filter has been derived, it must be convolved with each pixel in the image array to apply the Gaussian blur. This is done by placing each pixel in the image at the center of the kernel filter and convolving the two matrices. The average of this computation will then overwrite the current value of the image's pixel and thus blur that pixel to reduce its noise.

## Gradient Calculation

As seen in Equation 8, after blurring the image, the next step is to find the edge intensity and direction by calculating the gradient of the image using the Sobel edge detection filters.

$$K_x = \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$$

*Equation 8: Sobel Filters (Horizontal)*

$$K_y = \begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

*Equation 9: Sobel Filters (Vertical)*

As mentioned before, the edge simply corresponds to a significant change in the pixel's intensity. To highlight this change, the Sobel filters (in both the horizontal and vertical directions) can be convolved with the now blurred image. The filter will look at the intensity values of the neighboring pixels (to the currently processing pixel) and dot product the kernel size of those pixels with the Sobel filters [9] to return the gradient in both the horizontal ($6_x$)and vertical direction ($6_y$). The Sobel filters would then have to iteratively convolve through each pixel in the image to get the gradient of each pixel in the entire image. As seen in Equation 10 and Equation 11, the gradient in the horizontal and vertical directions can then be used to find the gradient magnitude and direction for each pixel.

$$|G| = \sqrt{{6_x}^2 + {6_y}^2}$$

*Equation 10: Gradient Magnitude*

$$\Theta(x,y) = \tan^{-1}(\frac{6_x}{6_y})$$

*Equation 11: Gradient Direction*

## Non-Maximum Suppression

Once gradient calculation has been performed, that marks the end of the Sobel edge detection algorithm and onto the Canny detection algorithm (improved version). The purpose of non-maximum suppression is to thin out the potentially numerous detected edges to a single maximum (since an edge can only create one maxima). This solves the final criterion of multiple responses that was mentioned earlier.

Furthermore, this process also refines the detected edges and produces "thin and well-defined edges" [1]. This thin (pixel-wide) continuous white line is much easier to visualize than a bunch of white lines with varying widths and lengths. To perform this computation, we must suppress the non-essential pixels within the image, leaving only the important edges standing.

As seen in Figure 1, the center pixel's gradient magnitude of the Gaussian Kernel is compared against the magnitude of its two adjacent pixels along its gradient direction. If either of the two adjacent pixels has a higher magnitude than the pixel being processed, the center pixel is set to a gradient magnitude of 0 (absolute black). If there are no pixels in the gradient direction having more intense values than the value of the current pixel, its gradient magnitude is kept as is. This algorithm ensures that the edge boundaries are pixel wide.

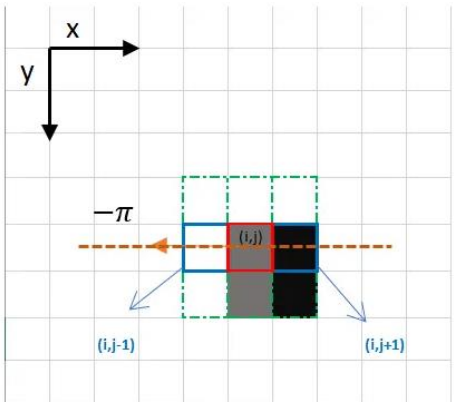


*Figure 1: Non-Maximum Suppression Diagram [8]*

## Double Threshold

The next part of the process is called double thresholding. The objective of this step is to identify three kinds of pixels in the image array: strong pixels, weak pixels, and non-relevant pixels. The main reason why double thresholding is important is that after computing the gradient, there will be various edges with different gradient magnitudes. Some edges will be well-defined, and others will be fainter, and harder to see. Double Thresholding ensures that when the final edge detected image is outputted, all the edges will have the same intensity value of 255 (absolute white).

The three types of pixels are determined by the user-inputted high-threshold and low-threshold values. If the pixel being processed has a magnitude higher than the high threshold, it's considered a strong pixel. If the pixel being processed is lower than the low threshold, it's considered a non-relevant pixel. Finally, if the pixel has a magnitude within both thresholds, it is flagged as a weak pixel that will be processed in the final step.

## Edge-Tracking by Hysteresis

The final step in the Canny edge-detecting process is to perform edge-tracking by hysteresis. The purpose of this step is to transform the weak pixels (that were flagged in the double thresholding) into strong pixels if and only if at least one neighbouring pixel has a higher gradient magnitude than the pixel being currently processed. The neighbouring pixels are defined as the nine pixels that are in direct contact with the pixel being processed. This final step is important in the algorithm as it addresses the issue of edge response to noisy data and helps in creating continuous edges by linking weak edges to strong edges. If this step was not performed, the image would output discontinuous edge segments, making it harder to identify the objects/segments of the image [2].

## Hough Line Transform

The purpose of the Hough line transform is to detect straight lines in an image. The transform can detect these straight lines even if the lines may be broken, partially obscured, or noisy [13]. The process of detecting straight lines has three main steps. The first step is to express the image using the polar coordinate system. The second step is to plot these coordinates and determine the point of intersection among curves. Finally, the last step is to use the plots to detect the straight lines in the image.

## Polar Coordinates

An image is normally typically expressed in Cartesian coordinates [12], however, to use the Hough transform, as seen in Equation 12 and Equation 13, the image coordinates must be translated to the Polar coordinate system.

$$r_\Theta = x_0 cos(\Theta) + y_0 sin(\Theta)$$

*Equation 12: Polar Coordinate Representation*

$$0 < r_\Theta, 0 < \Theta < 2\pi$$

*Equation 13: Polar Coordinate Bounds*

### Plotting Polar Coordinates

Once the image has been converted to polar coordinates, the next step is to plot the sinusoidal equations for all instances/pixels of $x_0$ and $y_0$. As seen in Figure 3, if the curves of multiple different instances intersect at a point $(\Theta, r_\Theta)$, then this intersection can be flagged.

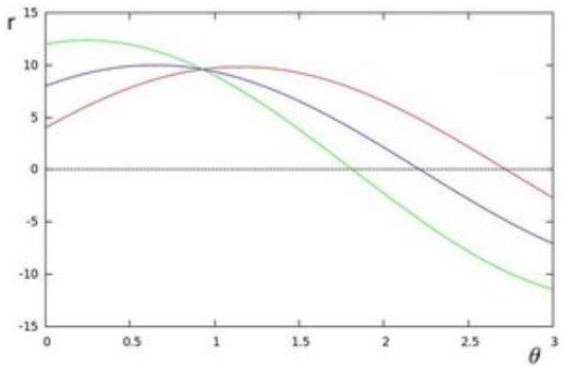


*Figure 2: Polar plot for instances of X0, Y0 [13]*

### Polar Intersections

The final step is to count the number of curves that create an intersection at a point $(\Theta, r_\Theta)$. If the number of intersections is above some user-inputted threshold, then it declares it as a line with the parameters. $(\Theta, r_\Theta)$.

## Approach

The purpose of this report is to solve the problem of lane detection in autonomous driving and computer vision applications. The primary objective is to accurately detect lane markings on the road to ensure safe and reliable autonomous navigation. When the program inputs a video of a car driving through an empty road, it should output the location of the center of the road which can be used to determine whether a driver is within their lane or not.

To solve this problem, the input image must first be pre-processed. This involves using first using the greyscale function to significantly reduce the amount of information that must be processed. Instead of a three-dimensional array in RGB, the output of the greyscale function is a black-and-white two-dimensional array. Once the image has been greyscaled, the image will undergo Gaussian blurring (using the Gaussian filter) to get rid of as much unwanted noise as possible.

Once pre-processing has been completed, the Canny edge detector algorithm is then used to isolate the edges of the blurred image. From there, the Hough line transform algorithm can be used to isolate the straight lines from the rest of the edges. Finally, the output of the Hough line transform can also be filtered by only taking straight lines which have a length greater than a user-inputted threshold. Since the Hough line transform returns the starting and ending point of each line segment, the equation for the distance of a straight line can be calculated using Equation 14.

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

*Equation 14: Line Segment Distance*

This extra filter is used because, there should only be two straight lines in a road, all the other ones would be due to unwanted noise. This final filter is used to reduce as many unwanted straight lines as possible, while still preserving the lines needed to detect the boundaries of the road. Once the boundaries of the road have been determined, the center of the lane can be found by taking the average of the leftmost and rightmost straight lines in the filtered image.

To test the algorithm, a video can be inputted into the program, of a car driving through an empty road. Since a video is nothing more than pictures compacted together, the algorithm should be able to find and then mark the lanes within the road. The program would then be run inside of a loop to output the markings of the road in real-time throughout the duration of the input video.

To analyze the results of this experiment, individual tests can be performed on the computational Canny edge detector and the lane detection algorithm to test its accuracy. Since the Canny function returns an array, the computational Canny edge detector's output array can be compared with the output array of the built in Canny edge detector output array to test its validity. Furthermore, the algorithm can be tested against images with known edges (truth edges) to analyze its metrics. Since the input video is also just an array (at any given instance), the array of markings that were calculated by the program can also be compared against

that of the original input image to determine its accuracy. The video will also contain various challenging driving scenarios such as curved roads and noisy backgrounds to test the flexibility of the program.

# Implementation

## Canny Edge Detector Algorithm

The first step is to create a program that will receive an input image and using the Canny edge detector algorithm described in the background section, output an image containing the edges of the input image. Note that the necessary libraries required to complete this program are OpenCV and NumPy.

### Gaussian Noise Reduction Algorithm

The first step in Gaussian noise reduction is to greyscale the image to one color channel. From there, NumPy's meshgrid function can be called twice to create the Gaussian kernel in both the horizontal and vertical directions. Finally, you can substitute the Gaussian kernels into the Gaussian filter equation (Equation 7) and use OpenCV's "filter2D" function to convolve the filter with the input image.

### Gradient Calculation Algorithm

Once Gaussian blurring has been applied, the gradient magnitude and direction of each pixel in the image must be calculated. This computation can be performed by calling on OpenCV's "Sobel" function which will return the gradient in both the horizontal and vertical direction. Using Equation 9 and Equation 10, the gradient magnitude and direction can be calculated from the gradients.

### Non-Maximum Suppression Algorithm

Before we can perform non-maximum suppression, the gradient direction array (found in the previous step) must be filtered. All values within the array must be converted from radians to degrees using Equation 15.

$$n[deg°] = m[rads]\left(\frac{180}{\pi}\right), 0° < n < 180°$$

*Equation 15: Conversion from Radians to Degrees*

From there, using NumPy's "zeros_like" function, an empty array can be created with the same dimensions as the gradient magnitude array. Two nested loops can then be created that will iterate through each element in the gradient magnitudes 2D array. As the loops iterate through each element in the array, five control statements can be put inside the nested loops to implement the non-maximum suppression logic.

As seen in Figure 1, there are four pairs of two pixels that neighbour any given pixel in the image. One pair horizontally, one pair vertically, and two pairs in the diagonal directions. These pairs will define the four possible gradient directions (one for each control statement) that a pixel could have. If the gradient magnitude of both pixels in every direction is less than the magnitude of the currently processing pixel, then rewrite the value of the current pixel's magnitude into the empty array from earlier, while maintaining the pixel's array position.

## Double Threshold Algorithm

Similar to the non-maximum suppression step, an empty array can be created with the same dimensions as the gradient magnitude array. NumPy's "where" function can then be used to find the indices of where the strong, weak, and non-relevant pixels are in the array based on the user-inputted low and high thresholds. Using the classification of the pixel and its returned indices, the newly created array can be set. Each pixel in the array will either be set to 255 (white) if strong, zero (black) if non-relevant, or the low threshold value if classified as a weak pixel.

## Hysteresis Algorithm

The final step in Canny's algorithm involves converting the weak classified pixels into either strong pixels or non-relevant pixels, black or white. Once again, two nested loops can be used to iterate through each element in the latest array. If any neighboring pixel (any pixel that directly borders the pixel being processed) was defined as a strong pixel, then the current pixel also becomes white. Otherwise, if no neighboring pixels are classified as strong, then the current pixel will be set to black. Once this step has been completed, the final array should only have two values, absolute black or absolute white.

## Canny Execution

Once the Canny edge detection algorithm has been written, the last step is to call the implementation. To do this, you can use OpenCV's "imread" function to read the input image by passing in the absolute path of the image into the function. Once the image has been read, the Canny function can be called by passing in the input image along with the necessary parameters into the Canny function that was just created.

The first parameter will be the input image, the second parameter will be the low threshold which is typically set at 100 for this application. The third parameter is the high threshold which Is usually set at 200 and the kernel size is usually set at either 3 or 5 (must be an odd number). Finally, the last parameter is sigma (used in Gaussian blurring) which is typically set between values of 0.8-1.2.

## Lane Detection Algorithm

The last half of the program is to create the lane detection algorithm using the newly created Canny edge detection algorithm. The algorithm will be split into four steps, line detection, line marking, lane detection and finally outputting the results.

### Line Detection Algorithm

The first step in the algorithm is to detect the straight lines that appear in the video. Since real-life videos contain a lot of noise (especially in the background), it's vital that this process filters as out as much unwanted noise as possible while maintaining the necessary straight lines.

The first step in this process is to call on the Canny edge detection algorithm that was just made. This computation will significantly reduce the amount of information in the video while keeping its necessary information. After this, the Hough lines transform function (mentioned earlier) can be called to filter out even more unwanted noise while keeping all the straight lines that were detected from the Canny algorithm. The Hough Lines function will return an array of detected lines where each row is a line that is expressed in polar form.

The first parameter of the Hough lines transform is the image you want to pass it. The second parameter ρ (pixel resolution) is typically set at 1 for this application. The third parameter is $\Theta$ (angle resolution) which is set to $\pi/180$ and the final parameter is the threshold (number of intersections needed) which is typically set between 40-60.

### Line Marking Algorithm

The second step in the algorithm is to mark the lines that were just detected. To highlight the accuracy of the detected lines, they were marked on top of the original video. To draw the lines on top of the original video, OpenCV's "line" function can be called using the coordinates of the line that were found in the earlier step. It is important to note that before drawing any lines, there must be a control statement filtering out the smaller straight lines that were caused by unwanted noise. If Equation 14 deems the length of the line segment to be less than the user-inputted threshold, then it should be withheld from outputting onto the video. Due to this filter, there should only be between 2-6 straight lines that are drawn on the original video. These straight lines that remain, are the detected lane boundaries.

### Lane Detection Algorithm

At this point in the algorithm, you should have a video that contains between 2-6 straight lines drawn on top of the lane boundaries. The final step in the algorithm is to use these boundaries to determine the position of the center of the road.

To perform this computation the average positions of either end of the lane can be taken to find the center of the lane. To find the position of either lane, a fixed column must be set in the center of the image. Using this fixed column, you can iterate through the image from left to right and flag (store in an array) the indices of wherever there is a white pixel in the row. Since the image was greyscaled at the very start of the algorithm, the only values that are in the array are black and white pixels. After all the filtering was complete with Canny edge detection and the Hough transform, the only white pixels that are left are the white pixels that belong to the road boundaries. As mentioned before, the size of this array should only be between 2-6 in size. Finally, by using Equation 16, the center of the lane can be found by taking the midpoint of the first and the last element in the array. Using OpenCV's "circle" function, a circle can be drawn on the midpoint of the lane to indicate its position.

$$midpoint = \frac{array[0] + array[-1]}{2}$$

*Equation 16: Midpoint of the Lane*

### Lane Detection Execution

The final step in the process is to call everything that was just implemented. To do this, OpenCV's "VideoCapture" function can be used to read the input video. From there, a loop can be created to iterate through every frame within the video. Inside this loop, the newly created lane detection algorithm can be called to calculate and output the lane markings of the road in real time. Onto the original video.

# Results

## Canny Edge Detector

The first thing that was done to determine the accuracy of the computational Canny edge detection algorithm was to directly compare it to the built in Canny detector from OpenCV. As seen in Figure 3, the two edge detectors can be compared to the control image. It is important to note that both the computational Canny function and the Canny function built into OpenCV were tested with identical parameters.


*Figure 3: Computational Canny (left), Control (center), Built-in Canny (right)*

The computational Canny algorithm classified 10.98% of total pixels to be edges, while OpenCV's Canny function classified 13.60% of total pixels to be edges. Furthermore, when comparing the two arrays, it was calculated that the two arrays shared 83.38% of the same elements in their matrices.

To obtain even more metrics needed to measure the accuracy of the computational Canny edge detection algorithm a control was used. This control consisted of an image and its corresponding array of "true edges" published by Berkeley's computer vision group [15]. To test the accuracy of the Canny algorithm, the image could be passed into the program, and the array that was returned could be compared to the array that was provided by Berkeley to test its accuracy and other metrics. As seen in Table 1, the performance of the computational Canny function and built in Canny function can be seen.

*Table 1: Canny Performance Metrics*

| Metric | Computational Canny Function | OpenCV's Canny Function |
|---|---|---|
| Precision | 0.0414 | 0.0291 |
| Recall | 0.214 | 0.272 |
| F1 | 0.083 | 0.058 |
| MSE | 3359.24 | 8838.9 |

Precision is defined as the ratio of correctly detected edges to the total number of edges that the detector classified. Recall is the ratio of correctly detected edges to the total number of true edges defined by the control. The F1 score takes precision and recall into consideration and provides a single value ranging from zero to one. One indicates perfect precision and recall and zero indicates poor performance. Finally, the mean square error (MSE) is a metric used to measure the average squared difference between the detected edges and the true edges. A lower MSE value indicates better performance, while a larger difference indicates poor performance [7].

## Lane Detection

To determine the accuracy of the lane detection algorithm, visual inspection must be used since there are no ground truth datasets that can be used as a control for this specific application. As seen in Figure 4, when taking a specific frame of the video, the blue markings represent the center of the lane and its boundaries.

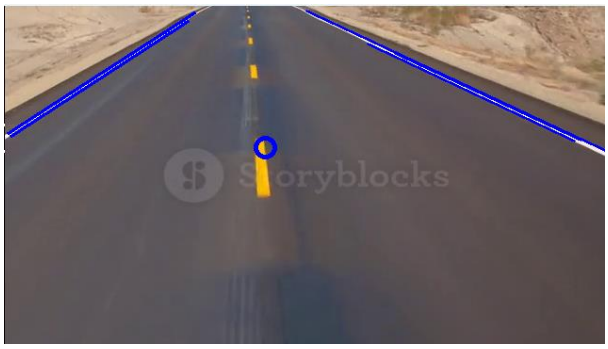

*Figure 4: Lane Detection Example [14]*

As seen in Figure 5, another test video was also used to determine the program's flexibility. This video was purposefully chosen as it has a completely different background and conditions to that of the first test video.

*Figure 5: Another Lane Detection Example [14]*

## Discussion

The first thing that was noticed in the results of the Canny edge detector, was that the performance between the computational and built-in functions was relatively simple. Both functions captured a similar percentage of edges and shared over 80% of their elements. The concerning metrics were in Table 1 where it appeared as if both functions had very poor performances in all categories. However, this could be explained if there was some sort of offset in detected edges.

As seen in Figure 3, both Canny functions can detect the edges of the image with relatively high accuracy. In other words, it's clear to see what object the edge boundaries form. However, even with this accuracy, the metrics seem to be extremely poor. This could be due to an offset in pixels. For example, if the detected edges from either function were offset from the true edges defined by the dataset, then only a small number of detected edges and true edges would overlap thus explaining the poor metrics. This would also explain why the output of both Canny functions seems to still resemble the output of the original image. A small offset in edges would not be a visible change to the human eye since in this image, there were over 154000 pixels.

Going further into detail on the metrics of the Canny function, it appears that the precision of the computational function is higher than the built-in function, however, its recall is lower. This could be explained by the fact that in this image, the computational function classified 10.98% of pixels as edges whereas the built-in function classified 13.60% of pixels as edges. Based on these metrics, it can be concluded that the built-in function detects more edges than the computational function and thus gets a higher ratio of the overall true edges correct (based on the dataset). However, in classifying more edges, the built it function has a lower precision to the number of correct edges that it classified. In essence, it can be concluded that the built in function favors quantity over quality, whereas the computational function favors quality over quantity.

This can be further proven using the F1 metric. This metric is used to equally balance both precision and recall, so even though each function beats the other in one category, the computational function has a higher F1 than the built in function. Furthermore, the MSE of the computational function is significantly lower than that of the built in function. These metrics can be used to conclude that the computational function has a higher performance (on this image) than the built in function provided by OpenCV.

Since there was no dataset to measure the accuracy of the lane detection algorithm, visual inspection was used. As seen in Figure 4, the blue lines are clearly over the original white lines used to describe the boundaries of the road. Because of this, the blue circle in the middle (used to mark the center of the lane) is almost directly over the dotted yellow line (reference). Furthermore, as seen in Figure 5, the blue lines almost perfectly overlap the original white lines of the frame. Because of this, the blue circle appears to be very close to the center of the lane, even if there was no reference line. It can thus be concluded that for straight lines, the algorithm is very accurate in detecting lane markers. A possible extension to the algorithm would be to take into consideration if the road was curved or not. This algorithm heavily depends on the Hough transform which is only used to detect straight lines in an image. However, if the road were to be curved, which occurs frequently in real life, the algorithm would not be able to detect this. A potential extension to the algorithm could be to use the Hough circle transform to detect road curvatures for autonomous driving.

## Conclusion

In conclusion, this study proved how you could computationally use the Canny edge detection algorithm and the Hough line transform to create a real-time lane detection algorithm. This study showed how the Canny edge detection algorithm can successfully detect edges in an image by using a multi-step process. It also showed how accurate the edge detector can be and how heavily dependent it is on the function's parameters. The lane detection algorithm incorporates the Canny edge detector along with the Hough line transform to highlight lane marking on the road. The study showed how accurate the algorithm can be on straight roads under various conditions. However, it also expresses the need to extend the study to incorporate curved roads using the Hough circle transform. Finally, this study showed how valuable the Canny edge detection algorithm can be in autonomous driving and other practical machine vision applications.

# References


[1] J. Canny, "A Computational Approach to Edge Detection," 1986 .

[2] S.Lakshimi, "A Study of Edge Detection Techniques for Segmentation Computing Approaches," 2010.

[3] P. Bao, "Canny Edge Detection Enhancement by Scale Multiplication," 2005.

[4] W. Rong, "An Improved Canny Edge Detection Algorithm," 2014.

[5] L. Xuan, "An Improved Canny Edge Detection Algorithm," 2015.

[6] R. Chellappa, "A Computational Approach to Boundary Detection," 1991.

[7] R. Srisha, "Operators Used in Edge Detection Computation: A Case Study," 2012.

[8] S. Sahir, "Canny Edge Detection Step by Step in Python — Computer Vision," Medium, 29 May 2020. [Online]. Available: https://medium.com/@ramitag18/performing-convolution-on-a-matrix-4682fd364591. [Accessed 15 July 2023].

[9] R. Agarwal, "https://medium.com/@ramitag18/performing-convolution-on-a-matrix-4682fd364591," Medium, 29 May 2020. [Online]. Available: https://medium.com/@ramitag18/performing-convolution-on-a-matrix-4682fd364591. [Accessed 15 July 2023].

[10] IceCream Labs, "3x3 convolution filters — A popular choice," IceCream Labs, 20 August 2018. [Online]. Available: https://icecreamlabs.medium.com/3x3-convolution-filters-a-popular-choice-75ab1c8b4da8. [Accessed 15 July 2023].

[11] Stock Media, "Download Empty Road At Night Royalty-Free Stock Video Clips," Stock Media, [Online]. Available: https://www.storyblocks.com/all-video/search/empty-road-at-night. [Accessed 25 July 2023].

[12] Indian Tech Warrior, "Canny Edge Detection for Image Processing," Indian Tech Warrior, [Online]. Available: https://indiantechwarrior.com/canny-edge-detection-for-image-processing/. [Accessed 18 July 2023].

[13] OpenCV, "Hough Line Transform," OpenCV, [Online]. Available: https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html. [Accessed 22 July 2023].

[14] Medium, "Evaluation Metrics 101," Medium, [Online]. Available: https://medium.datadriveninvestor.com/evaluation-metrics-101-7c8b4c3421c2. [Accessed 22 July 2023].

[15] University of California, Berkeley, "Contour Detection and Image Segmentation," University of California, Berkeley, [Online]. Available: https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html#bsds500. [Accessed 26 July 2023].