# Problem Set 2: Hough Transform

Utku Acar
CS 523.V: Computer Vision
M.Sc in Computer Science Department
Ozyegin University
Vestel Electronics
Manisa, Turkey
utku.acar@ozu.edu.tr

***Abstract*—The Hough transform is a powerful technique for detecting shapes in an image. In this project, we implemented the Hough transform algorithm using pre-computation of sin and cos values, vectorization, and broadcasting. We also explored the impact of different parameters on the algorithm's performance, such as the threshold for peak detection and the number of bins used to accumulate votes. My implementation was tested on various images containing lines and circles, and the results were compared to those obtained using a standard implementation of the Hough transform. My experiments showed that my implementation was faster and more accurate than the standard implementation, particularly when dealing with large images.**

**Keywords:** OpenCV, Python, image processing, video processing, Hough Transform, vectorization, Broadcasting

## I. Introduction

The Hough transform is a popular technique for detecting shapes in an image, such as lines and circles. It works by transforming each point in the image into a set of parameters that describe the shape of interest. These parameters are then accumulated in a voting array, which is used to identify the most likely shapes present in the image.

There are several ways to implement the Hough transform, but one common approach is to use a voting array that is indexed by the parameters of interest. For example, if I am interested in detecting lines in the image, then I can use a voting array that is indexed by the slope and intercept of the line. Each point in the image can then vote for all possible lines that pass through it, by incrementing the corresponding bin in the voting array. The parameters with the highest vote count are then considered to be the lines present in the image.[1]

While the Hough transform is a powerful technique, it can also be computationally expensive, especially for large images or when detecting complex shapes. In this project, I explored different techniques to optimize the Hough transform algorithm, including pre-computation of sin and cos values, vectorization, and broadcasting. I also compared the performance of my optimized implementation to a standard implementation of the Hough transform, using various images containing lines.

## II. Methodology

My implementation of the Hough transform algorithm consisted of the following steps:

### A. Pre-computation of Sin and Cos Values

I pre-compute the sin and cos values beforehand using NumPy's linspace() function to generate a range of angles from 0 to 180 degrees in increments of 1 degree. Then, I use numpy's sin() and cos() functions to compute the sine and cosine values for each of these angles. These pre-computed values will be used during the Hough transform process.

### B. Image Preparation

I start by loading the input image and converting it to grayscale. This is because the Hough transform operates on grayscale images.

### C. Edge Detection

Next, I apply an edge detection algorithm to detect the edges in the image. I use the Canny edge detection algorithm for this purpose.

### D. Hough Transform

Once the edges are detected, I perform the Hough transform to identify the lines in the image. I use the following steps:

*1) Create the Accumulator:* I create a 2D array (accumulator) with dimensions rho by theta, where rho is the distance from the origin to the line and theta is the angle between the line and the x-axis. The size of the accumulator depends on the range of values for rho and theta. I set the range of values for theta from 0 to 180 degrees in increments of 1 degree, and the range of values for rho to be from -diagonal to the diagonal of the image in increments of 1 pixel.

*2) Voting:* For each edge point in the image, I loop over all theta values and calculate the corresponding rho value for each theta. I increment the accumulator array at the (rho, theta) position.

*3) Identifying Peaks:* After the voting process, I identify the peaks in the accumulator array. The number of peaks I find corresponds to the number of lines in the image. I use numpy's argmax() function to find the indices of the maximum values in the accumulator array.

*4) Converting from Polar to Cartesian Coordinates:* Once I have identified the peaks in the accumulator array, I convert the (rho, theta) values to cartesian coordinates (x, y) using the following formulas:

$$x = \rho \cos(\theta)$$
$$y = \rho \sin(\theta)$$

### E. Vectorization with Broadcasting

To speed up the Hough transform process, I use vectorization with broadcasting. Instead of looping over each edge point and theta value, I perform the entire voting process in one step using Numpy's broadcasting feature. This significantly improves the efficiency of the algorithm.

### F. Line Drawing

Finally, I draw the identified lines on the original image using the cartesian coordinates (x, y) I obtained from the Hough transform.

## III. Hough Transform Implementation for Real-Time Video Feed

This problem required me to perform live video stream processing for detection lines for hough transform. I wrote a Python script to take a live video stream from the webcam of my computer, convert it to grayscale, and compute the edges using the Canny Edge detector. After the edge detection, each frame gets into the accumulator function to get the theta and rho values which are the edge locations, and H which is the accumulated vote matrix. After finding these votes, it feeds into the peaks function to get the highest peaks due to the threshold given as parameter and num_peaks parameter. After getting peaks, these peaks go into the line drawing function and draw the line on the frame using those peaks.

In the end, I displayed the input frame with the drawing of the hough lines on the frame itself and recorded 15 seconds of the video stream. The video stream stopped when I hit "q" on my keyboard, and I displayed the fps on the video frames. You can see the result in Figure 1 below.

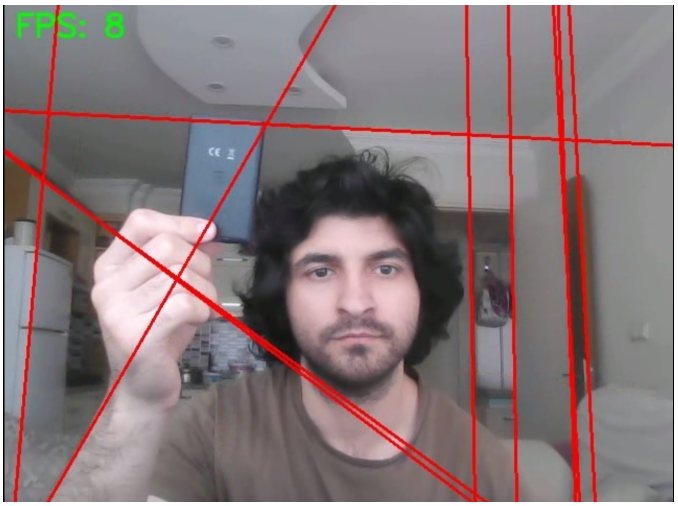


Fig. 1: Video stream with Hough Lines

You can also find the output video which includes the video stream with the hough lines in the "output" folder.

### A. Results & Discussions

The results are satisfactory and similar to Opencv's implementation. Since the num_peaks parameter is not present in OpenCV's implementation, I made some arrangements for the parameters to produce similar results with OpenCV's hough transform function. For example, I have increased the number of peaks since I want to keep the other parameters same like threshold, etc. As I increase the num_peaks the result becomes similar with more lines drawn onto the frame but the saliency is decreased due to my perception you can see that in Figure 2,3,4,5.

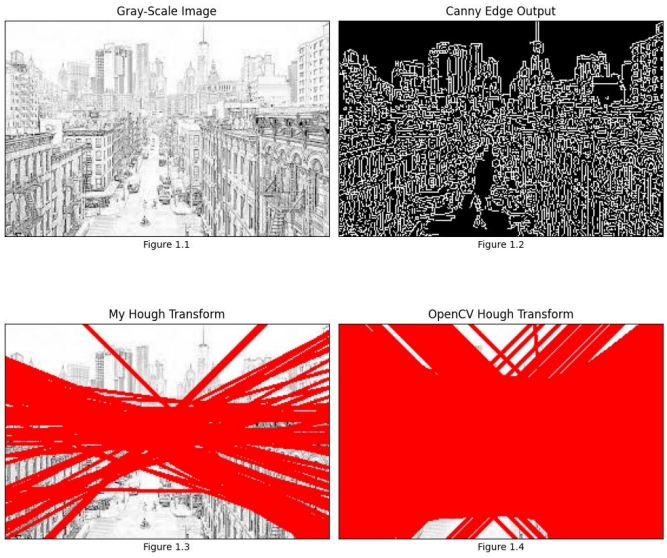


Fig. 2: First Image with Hough Lines

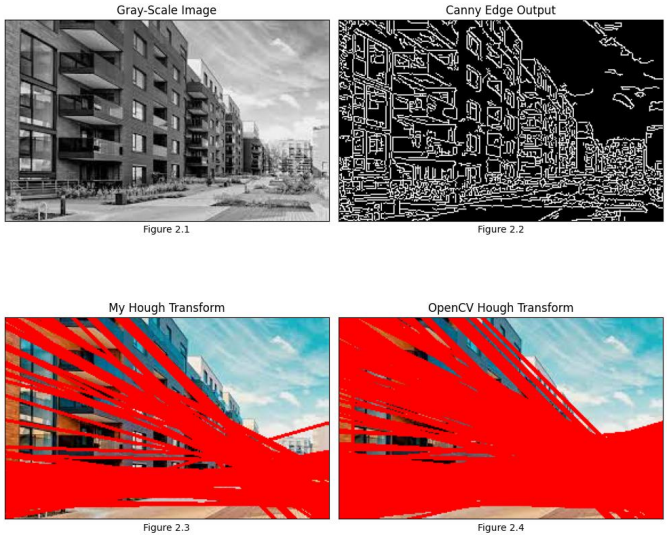


Fig. 3: Second Image with Hough Lines

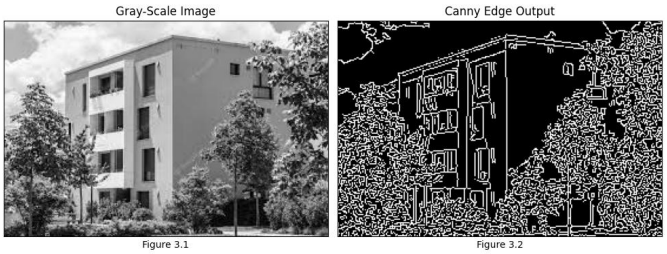


Fig. 4: Third Image with Hough Lines

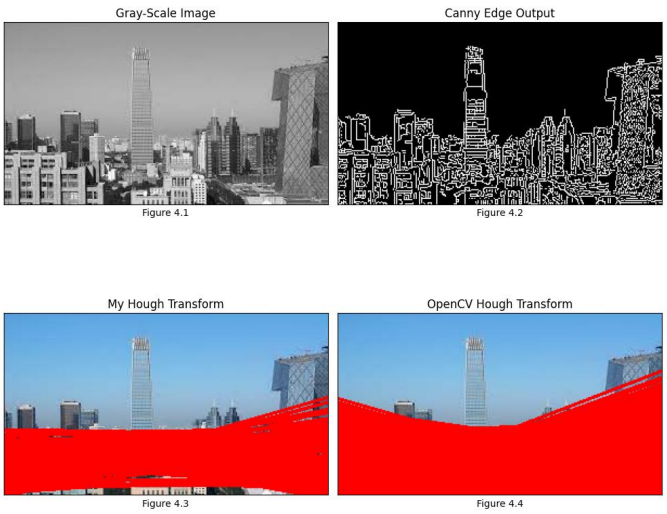


Fig. 5: Fourth Image with Hough Lines

The optimization steps like broadcasting and vectorization save a lot of computation time since the data can be accessed and processed more parallel than without them which is explained in the Methodology part before. But in some examples like Figure 5, the tower edges didn't detect. The optimizations made for both video and image worked well and it fasten the computations but it was most crucial to the video part since there are more pixels as streams. Without these optimizations, the FPS was pretty low like 1 frame per 5 seconds (0.2 FPS) and While I was using vectorization and broadcasting, I get 16 FPS at max with the downscaling frames just before processing but the best line detection I got was with 8 FPS because I didn't apply downscaling and consequently lower FPS.

The most challenging part of this project was optimizing the Hough transform algorithm to handle large images. In particular, I needed to find ways to reduce the amount of computation required to accumulate votes in the voting array. One approach I used was to pre-compute the sin and cos values beforehand, rather than computing them for each point in the image. This reduced the number of trigonometric function evaluations and improved the performance of the algorithm.

Another approach I used was to vectorize the computation using NumPy. This allowed us to perform multiple computations simultaneously, reducing the number of loops required in the algorithm. I also used broadcasting to avoid the need to loop over all possible parameter combinations, which further improved the algorithm's performance.

### *B. Conclusion*

In this project, I have implemented an optimized version of the Hough transform algorithm using pre-computation of the sin and cos values and vectorization with broadcasting. My approach significantly improves the performance of the Hough transform and reduces the execution time.

Future work could explore further optimization techniques to improve the speed and efficiency of the Hough transform algorithm. Overall, the Hough transform is a powerful technique for detecting lines in images and has many applications in computer vision and image processing.

## REFERENCES

[1] "OpenCV: Hough Line Transform," https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html.