Laboratory Exercise 3

Lane Detection for Autonomous Driving

This exercise introduces you to the Hough transform and how it can be used for detecting lanes in an autonomous driving application. The exercise uses the Canny edge detector that you implemented in Exercise 2.

Lane Detection Using Line Detection

Lane detection is a technique that uses a vehicle's sensor data to determine the vehicle's position relative to the lanes and/or road boundaries. The position information is used to make steering adjustments and to change lanes, which makes lane detection a fundamental part of an autonomous vehicle.

There are many different lane detection schemes that make use a variety of different sensor combinations. In this exercise, we will implement a simple scheme that uses a camera attached to the front of the vehicle. Since lane and road boundaries tend to appear as lines in the camera feed, we will accomplish lane detection by detecting the lines in the image. Once we have the positions and angles of the lines, we can use our knowledge of the perspective of the camera to infer the vehicle's position relative to the lane boundaries. To detect the lines, we will employ edge detection and a technique called the Hough transform. Figures 1, 2, and 3 show the application of edge detection followed by the Hough transform to detect lines in an image.

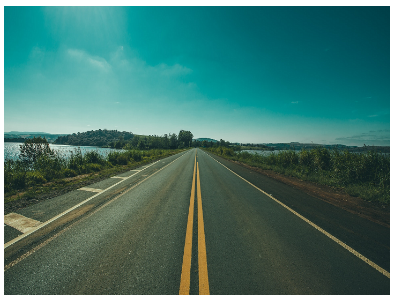


Figure 1: Original image.

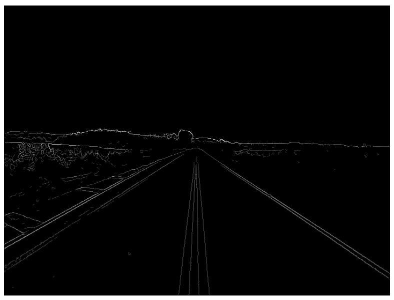


Figure 2: Edge-detected image.

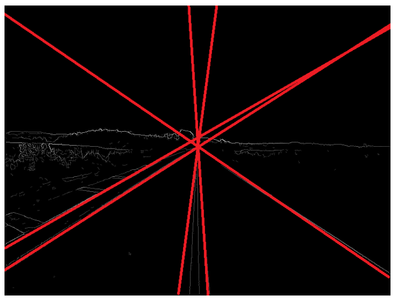


Figure 3: Line-detected image.

The Hesse Normal Representation for Lines

Before starting our discussion of line detection, we must first consider the system we will use to describe lines in an image. For efficiency, we want to describe a line using the fewest parameters possible. A commonly used line representation is $y = mx + b$ which describes lines using two parameters *m* and *b*. The problem with this system is that it cannot describe vertical lines as m would assume an undefined value. The line representation that we will use is known as the Hesse normal form. This system uses two parameters (ρ, θ) to describe a line **A** as follows:

1. ρ is the distance from the origin (the center of the image) to the closest point on A.
2. θ is the angle between the x axis and the normal line connecting the origin to the closest point on A.

To account for all possible lines in an image, the parameters range: $0 \leq \theta < 180°$ and
$-\sqrt{(image\_height^2 + image\_width^2)}/2 \leq \rho \leq \sqrt{(image\_height^2 + image\_width^2)}/2$

Examples of lines and their (ρ, θ) parameters are shown in Figure 4. The line being described is displayed in red, and the normal line is displayed in magenta. Take note that ρ can be a negative value which extends the normal line in the reverse direction from the angle denoted by θ, as shown in Examples 2, 3, and 4. The line shown in Example 3 is parameterized as $(-7, 0°)$ and not $(7, 180°)$, due to the valid range of theta: $0 \leq \theta < 180°$. For the same reason, the line in Example 4 is parameterized as $(-7, 90°)$ and not $(7, 270°)$.

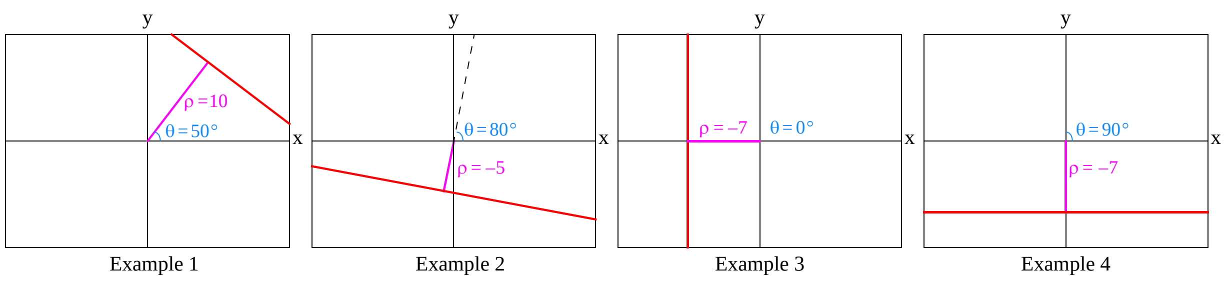


Figure 4: Lines (in red) and their (ρ, θ) parameters.

Lines described in (ρ, θ) form can be mapped to pixels in (x,y) coordinates and vice-versa using the equation $\rho = x * cos(\theta) + y * sin(\theta)$. Note that a pixel can lie on an infinite number of lines, as there are the infinite θ values from 0° to 180°. Figure 5 shows a pixel at (x,y) coordinate (8,5) and some of the lines that it is positioned upon.

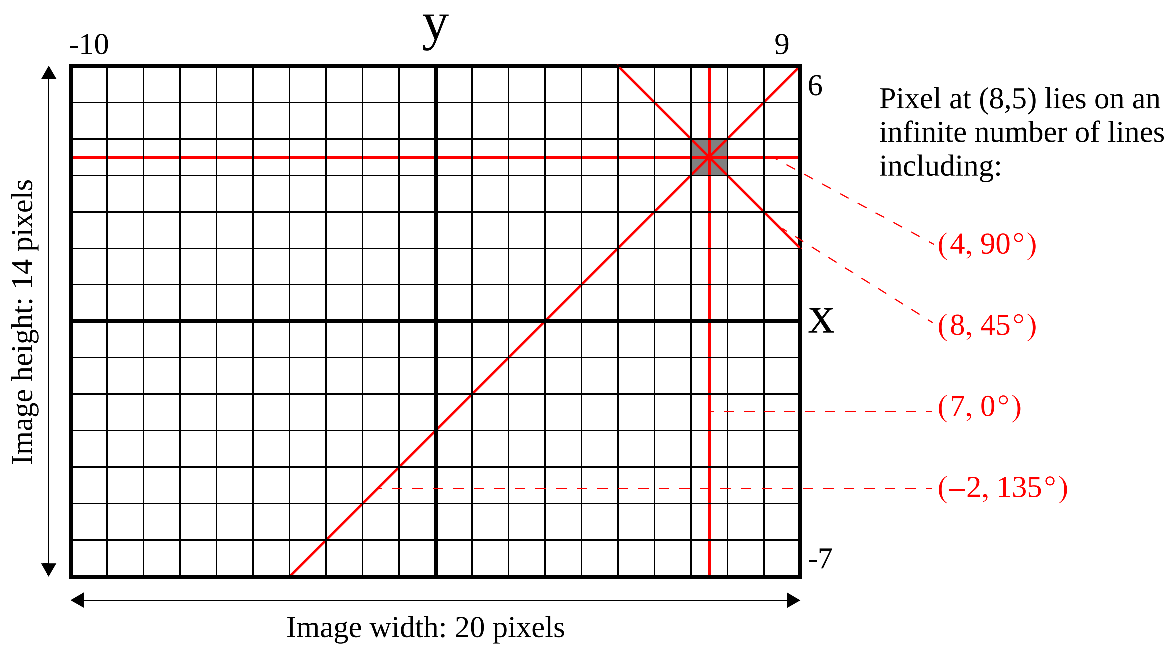
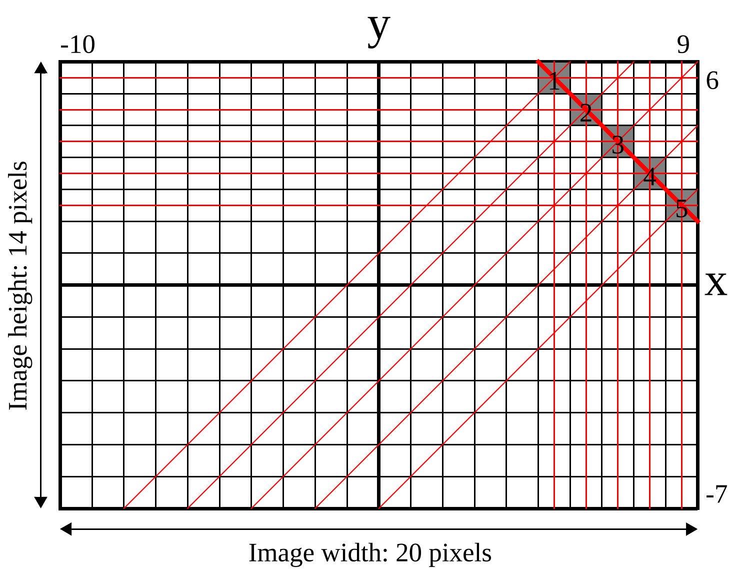


Figure 5: A pixel in an image can lie on an infinite number of lines, including the four lines shown.

The Hough Transform

The Hough transform takes an edge-detected input image and transforms it to a 2-dimensional integer array called the accumulator. The two dimensions of the accumulator are indexed by ρ and θ respectively, and each element stores how many edge pixels in the input image are positioned along the corresponding line (ρ, θ). The most prominent lines in the input image can then be inferred by finding the lines with the highest values in the accumulator. The process by which the Hough transform calculates the values in the accumulator is described below.

The accumulator elements are first initialized to zero, then *accumulate* in value as the Hough transform algorithm sweeps across the pixels of the input image. Each edge pixel encountered by the algorithm contributes +1 to the accumulator values of all lines on which it is positioned. Since there are theoretically an infinite number of such lines, we must select a resolution for ρ and θ to limit the number of lines being tracked by the algorithm.



The pixels' contributions to accum:

1: accum[5][0], **accum[8][45]**, accum[6][90], accum[1][135]

2: accum[6][0], **accum[8][45]**, accum[5][90], accum[-1][135]

3: accum[7][0], **accum[8][45]**, accum[4][90], accum[-2][135]

4: accum[8][0], **accum[8][45]**, accum[3][90], accum[-4][135]

5: accum[9][0], **accum[8][45]**, accum[2][90], accum[-5][135]

Figure 6: Applying the Hough transform (θ resolution = $45°$) on a 20x14 image with five edge pixels.

As a demonstration, let us choose a θ resolution of $45°$ and a ρ resolution of 1 pixel to transform a 20x14 image. This requires an accumulator array, `accum[RHOS][THETAS]`, with dimensions RHOS = 24, and THETAS = 4. Integer values of ρ range $-12 \leq \rho < 12$, and θ values range $0°, 45°, 90°$, and $135°$. Figure 6 shows the operations involved in transforming a 20x14 edge-detected image with five edge pixels in the top-right corner. The five pixels form the line $(8, 45°)$, which is shown as the bold red line. Each pixel contributes +1 to the four accumulator indices corresponding to the four possible θ values. The ρ index corresponding to each θ is calculated using the formula $\rho = x * cos(\theta) + y * sin(\theta)$. After the Hough transform completes, the accumulator is as shown in Figure 7. As expected, `accum[8][45°]` has the highest value of 5, since there were five pixels along it in the input image.

Theta

-12												0											11	
0	0	0	0	0	0	0	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	135°
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	90°
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	45°
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0°

Rho

Figure 7: The accumulator resulting from the Hough transform of the image in Figure 6.

Extracting Lines from Local Maxima in the Accumulator

Upon completion of the Hough transform we are left with the task of extracting prominent lines from the accumulator. The simplest method for extracting the N most prominent lines from the accumulator is to find the N highest values (global maxima). However, this approach is prone to issues which we will discuss below.

A source of error when using the Hough transform for line detection is the potential for the pixels of a single "real" line in an image contributing to multiple adjacent elements in the accumulator. This could happen if the line was not perfectly straight due to noise, or if our resolution for ρ or θ was inadequate. For example, consider transforming an image with the line (11,7.5°) using a θ resolution of 1°. During the transform some of the pixels in the line may contribute to `accum[11][7]` while others may contribute to `accum[11][8]` resulting in multiple nearby accumulator elements with similarly high values. After the transform if we simply searched for the global maxima to detect lines, we could falsely detect two or more lines resulting from the single real line.

To eliminate false duplicates, we will extract lines from local maxima in the accumulator rather than global maxima. Whenever there are multiple high accumulator values in a vicinity we will only consider the one whose value is highest, and eliminate those neighboring. This approach allows us to detect the single line which is likely the best fit to the real line in the input image. Figures 8 and 9 illustrate the differences between global maxima and local maxima extraction. Note that this approach runs the risk of elliminating lines that actually did exist near eachother in the input image, but for the purposes of lane detection we can assume lane markings are sufficiently spaced apart for this to not be an issue.

Searching for local minima also allows us to search for a fewer lines in total while cover all of the lanes in the image. Consider an image with two lane markings where one was significantly longer than the other. In this case, the longer lane may yield multiple accumulator values that are greater than any of the values for the shorter lane. If we then searched for two global maxima with the expectation of detecting the two lanes, we would detect the longer lane twice and the shorter lane not at all. Searching for local maxima means that we would only extract one line corresponding to the longer lane, allowing for the detection of the second lane as long as it was the second highest local maximum in the accumulator.

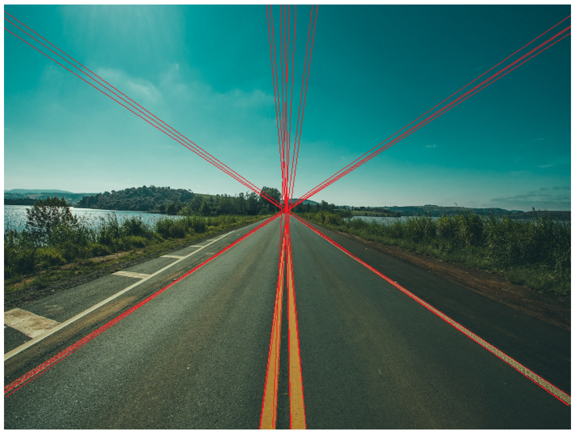


Figure 8: Lines extracted from global maxima.

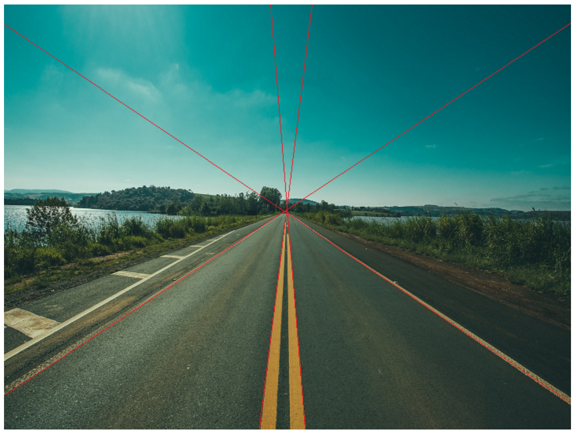


Figure 9: Lines extracted from local maxima.

Elliminating the Horizon and Horizontal Lines

To improve the accuracy of our lane detector, we will make two simple optimizations to the Hough transform and line extraction.

The first optimization aims to prevent detecting lines from objects far in the horizon (such as those detected from hills or clouds). Figure 9 shows an example where the upper portion of the image contains the horizon and the sky. When implementing the Hough transform we will elliminate the horizon by only going through the bottom half of the edge-detected image when incrementing the accumulator. Not only will this optimization potentially prevent false lines, it will reduce the runtime of the Hough transform by skipping over half of the input pixels.

The second optimization elliminates horizontal lines from being extracted from the accumulator. This optimization is based on the knowledge that lanes boundaries will tend to point outward from the camera's perspective (from the bottom of the image towards the top), meaning that vertical and slanted lines are the ones likely to correspond to lanes. Figure 10 shows a scenario where this optimization is particularly beneficial, as the merging bus contains features that would result in prominent horizontal lines being detected. We can implement this optimization in our line extraction stage by ignoring the lines in the accumulator whose θ indicates a horizontal line ($80^\circ < \theta < 100^\circ$).

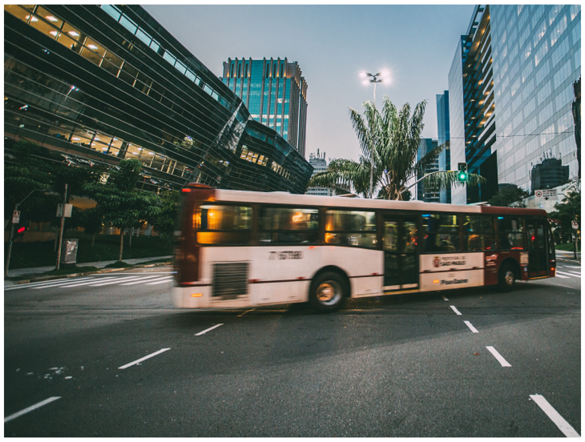


Figure 10: A merging bus contains prominent horizontal lines that may be detected instead of the lane markings.

Part 1

Write a C-language program to implement the Hough transform. Start with the skeleton code provided in */design_files/part1/hough.c*. Your program should load a 720x540 BMP image, apply edge detection, apply the Hough transform, extract the lines, overlay the lines onto the original image, then save the resulting image as a BMP file.

Your Hough transform should use a θ resolution of 1° and a ρ resolution of 2 pixels. Ellminate the horizon and horizontal lines using the optimizations described in the previous section. Extract five lines from the five highest local maxima in the accumulator. A value is considered the local maximum if it is the highest value in the 5x5 square around it (+/- 2 to the ρ and θ indices). Overlay the extracted lines onto the original image (as shown in Figure 9), then save it as a BMP file. Use the images provided in */design_files/test_images/* to test your program. How closely do the detected lines align to the lane boundaries? For images where lane detection was less successful, explain potential source(s) of error.

Record the runtime of the Hough transform across the test images. Why does the Hough transform runtime vary by image? Using the average total runtime across the images, calculate framerate at which you processed them. The total runtime includes edge detection, Hough transform, and line extraction, but not operations involved with loading and saving the BMP images. Determine the upper bound on the runtime of the Hough transform and calculate the minimum framerate.

Part 2

Create an OpenCL kernel for the Hough transform. Start with the the skeleton code provided in */design_files/part2/*. Use the same parameters as in Part 1. To improve performance, use local memory to hold the accumulator array and unroll critical loop(s). Instantiate your edge detection kernel from Exercise 2 to compile a single *.aocx* image that contains both the edge detector and the hough transform.

Skeleton host code is provided in */design_files/part2/host/*. You must add to the code so that the host program first calls the edge detection kernel, waits for it to complete, then feeds the edge detected data buffer to the hough transform kernel. As in Part 1, your host program should extract five lines from the five highest local maxima in the accumulator, overlay them onto the original image, and save it as a BMP file.

Record the runtime of the Hough transform across the test images. How does the runtime compare to your implementation in Part 1? Using the average total runtime across the images, calculate framerate at which you processed them. Determine the upper bound on the runtime of the Hough transform and calculate the minimum framerate.

When implementing the Hough transform kernel, you can use the pragma `#pragma ivdep` to tell the OpenCL compiler that accesses to `accum` in subsequent loop iterations are free of data dependencies for better pipelining of the circuit. The OpenCL compiler would otherwise have difficulty in determining the lack of dependency, as subsequent indices calculated by the formula $\rho = x * cos(\theta) + y * sin(\theta)$ are seemingly unpredictable. We as designers however, know that the formula will ensure subsequent accesses to `accum` will be at different indices, as we sweep θ from $0°$ to $179°$ for each edge pixel.

Part 3

Improve the performance of your design from Part 2 by using OpenCL pipes to connect the output of your edge-detection kernel to the input of your Hough transform kernel. Doing this allows your kernels to share data completely inside the FPGA, rather than using slow external memory as intermediary storage. Use a blocking pipe with a depth of 1. Compile your piped kernels using the `aoc` flag `-profile=all` to enable profiling. Make the necessary modifications to your host program to support your newly piped kernels.

After running your application, use the profile monitor to check for kernel stalls. Do your kernels stall as a result of the blocking pipe? What do the stalls (or lack of stalls) indicate about the throughput of your edge detection kernel compared to your Hough transform kernel?

Record the runtime of your piped implemention across the test images. How does the runtime compare to your implementation in Part 2? Using the average total runtime across the images, calculate framerate at which you processed them. Determine the upper bound on the runtime of the Hough transform and calculate the minimum framerate.