CS 456/556, Fall 2024

Programming Assignment 1
Image Filtering and Hough Transform

Adarsh Salukhe

Q2.1 Hough Transform Line Parameterization

- **Image plot, indicating the intersection point.**
- **For this line, what is (m, c) in the equation of the line $y = mx + c$?**

Answer:

- The points given were (10,10), (20,20), and (30,30).
- All three points had a sinusoidal curve in Hough space.
- The plotted sinusoidal curves intersected at a common point.
- This intersection represented the parameters (ρ, ϑ) of the line passing through these points.
- The slope m is calculated as:

$$m = \frac{20-10}{20-10} = 1$$

- Using the point (10,10) in $y = mx + c$:

$$10 = 1(10) + c$$

- Solving for c, we get $c = 0$.
- Therefore, the equation of the line is $y = x$.

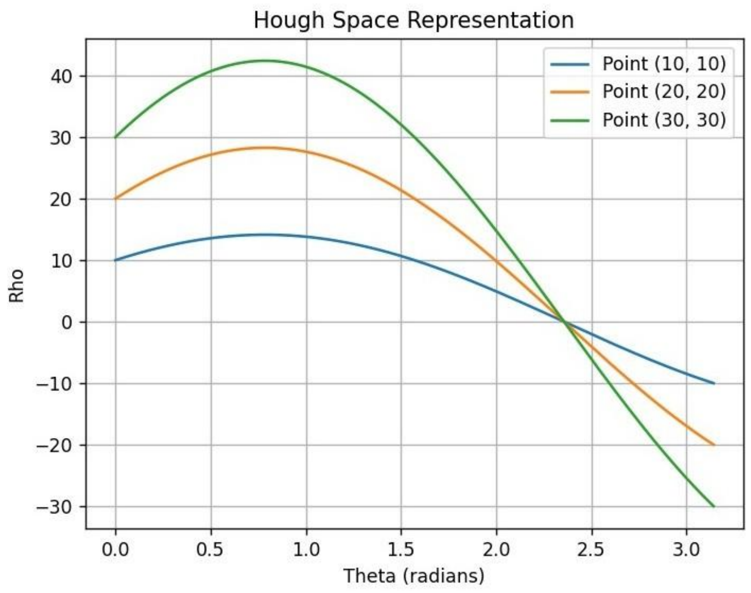


Figure 1: Intersection of sinusoidal curves in Hough space

Q3.1 Convolution/Correlation

- **State if your function implements convolution or correlation.**
- **Explain how you implemented convolution or correlation.**

Answer:

- The function in myImageFilter.py implements correlation.
- for every pixel we extract a patch from the padded image that is the same size as the filter h, then we perform element-wise multiplication with h and sum the result to produce the output pixel value.
- The boundary pixels which are generated are handled properly by reflecting the same from the nearest values instead of setting them to 0.
- For each of the pixel present at the position (i,j) a small patch of the same size as the filter is extracted from the padded image.
- The extracted image patch is multiplied with the filter h.
- The sum of these multiplied values is assigned to the corresponding output pixel.

Q3.2 Edge Detection

• Explain how you implemented edge detection.

• What is your analysis after using different filter sizes, and which size did you decide to stay with at the end?

• Display the output of your function for three of the given images in the hand-out.

Answer:

- First, I applied Gaussian smoothing using a kernel whose size is determined by sigma (computed as 2·ceil(3· sigma)+1). This step reduces noise and prevents spurious edge responses.
- I computed image gradients using x and y-oriented Sobel filters. The gradient magnitude and angle are calculated from these responses. Then, I performed non-maximum suppression by comparing each pixel's gradient magnitude to those of its neighbors along the quantized gradient direction (0°, 45°, 90°, or 135°).
- After tuning all the parameters, I observed that when i used small sigma value the details were seen however the ouptut contained noise that made the edge detection less reliable.
- After multiple testing on the program I found out that the sigma value of 1.2 provided me the best balance between the noise reduction and edge clarity that made the edges accurate

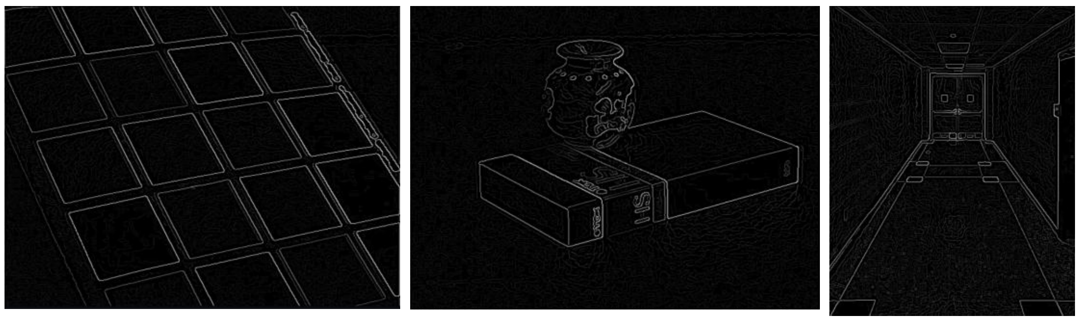


Figure 2: Edge detection results

Q3.3 Hough Transform

• **Explain how you implemented Hough transform.**

• **Analyze your results with different resolutions of the accumulator, and explain why you chose the final parameter values.**

• **Display the output of your function for three of the given images in the handout (same images as before).**

Answer:

- I implemented the Hough Transform in myHoughTransform.py to detect straight lines in an edge detected image. I first compute the diagonal length of the image so that the variable rho can range from negative M to positive M, ensuring that every possible line is covered. Then, I create a range of rho values and a range of theta values, with resolutions determined by the parameters rhoRes and thetaRes. For each edge pixel in the threshold image, I compute the corresponding rho values for each theta by using the rho equation
- After experimenting with different resolutions, I chose parameter values that balanced accuracy and computational efficiency. In our implementation, using rhoRes = 2 pixels and thetaRes = pi/180 worked well. These settings provided a detailed enough Hough space to detect lines accurately without over-fragmenting the votes, ensuring that strong lines produced clear peaks in the accumulator.

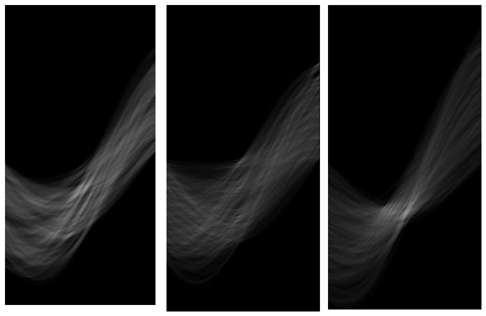


Figure 3: Hough Transform results

Q3.4 Finding Lines

- **Explain how you implemented Hough transform.**
- **Analyze your results with different values for nLines.**
- **Display the output of your function for three of the given images in the handout (same images as before).**

Answer:

- In myhoughlines.py, the Hough transform is implemented by first applying an edge detection step to obtain a thresholded image. The function takes in a Hough accumulator (H) and the desired number of lines (nLines). It normalizes H to the range 0–255 using OpenCV's normalization, then converts the result to an unsigned 8-bit format. A 7×7 kernel is used to dilate the normalized image. This dilation helps highlight local maxima potential line candidates by expanding the regions of high values.
- The code identifies local maxima by checking where the original normalized H equals its dilated version. These points are considered peaks in the Hough space. It then retrieves the corresponding accumulator values and sorts the peaks in descending order to prioritize the most prominent ones.
- To avoid selecting multiple nearby peaks that essentially represent the same line, the code applies a non-maximum suppression step. It skips peaks that are too close in terms of the rho and theta parameters (with thresholds defined by minmum rho. Finally, once the specified number of distinct lines is reached, the function returns the rho and theta values of these selected lines as NumPy arrays.
- When analyzing the results with different values for the number of lines to extract, I observed that choosing a smaller number focuses on the most dominant and clear lines.
- The rho values were 10 pixels apart and the theta values were pi/180 to ensure no redundant lines were included in the output.In contrast, selecting a larger number leads to the detection of additional lines, some of which might be less significant or even redundant.

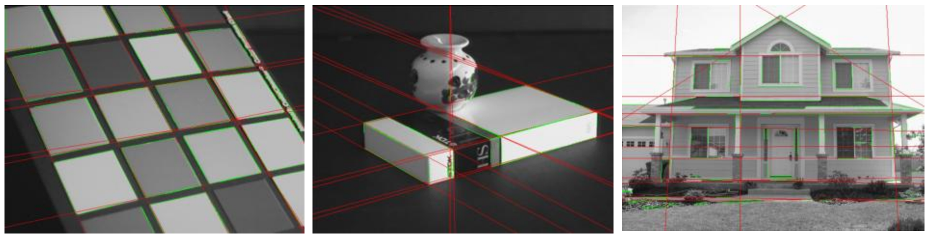


Figure 4: Finding nLines results

Q3.5 Implementing HoughLinesP yourself

• **Explain how you implemented the HoughLineSegments function.**

• **Display the output of your function for three of the given images in the handout (same images as before), as well as the corresponding results from OpenCV's HoughLinesP call.**

• **Briefly compare the results.**

Answer:

- My custom HoughLineSegments function works as follows. First, it examines the Hough accumulator and finds the strongest peaks by repeatedly selecting the maximum accumulator value and then zeroing out a neighborhood around that peak. Each peak corresponds to a candidate line, from which the corresponding distance and angle parameters are derived. For each candidate line, the function then ”traces” along the line in both directions in the image space.
- It collects points where the edge image is active above a threshold and groups these into continuous segments. A minimum segment length is enforced so that very short or noisy detections are discarded, and a gap threshold is used to merge segments that are close together.
- This results in a set of line segments with start and end points that approximate the lines present in the image.
- I observed how nLines affected the output, when they were low, only the strongest lines were visible and all the edges were missed. when the nLines value was increased above 20, many lines appeared which made the image noisy and weak edges were detected.
- Therefore, setting the nLines to 15 was the optimal choice for detecting all the important lines and not causing too much noise.

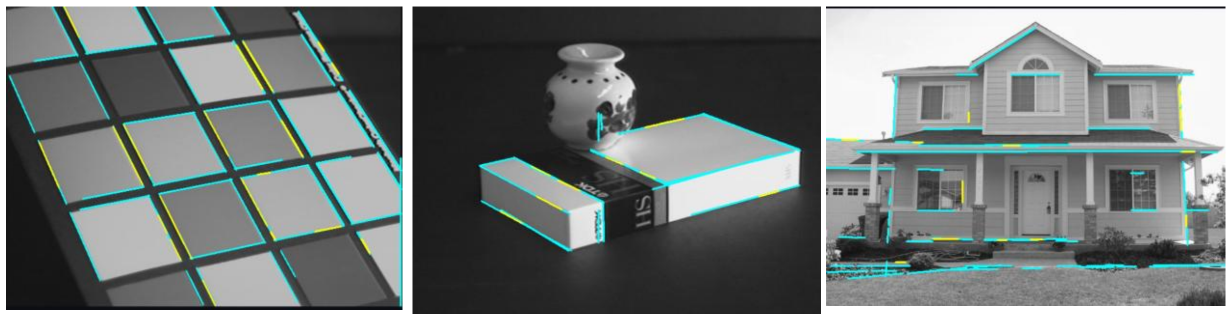


Figure 5: Hough Line Segments results

Q3.6 Try your own images!

- **Include input and resulting images..**

- **Briefly explain the results.**

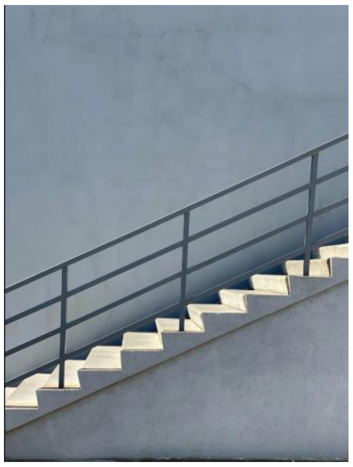
Answer:

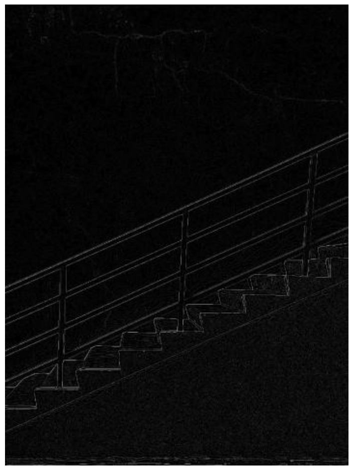
- Firstly my approach was to apply an edge detection filter myEdgeFilter. This step was crucial because a well-defined edge map determined how effective the subsequent line detection would be. Each image is loaded using OpenCV and converted to grayscale if it is in color. The image is normalized by scaling pixel values to the range [0, 1] for further processing. The custom function (myEdgeFilter) applies Gaussian smoothing (with sigma=1.2) to reduce noise before edge detection. However, I noticed that some background textures still appeared faintly. The choice of sigma for the Gaussian filter had a big impact here; too low, and the image remained noisy, too high, and I lost finer details. Sobel operators are used to compute image gradients, and non-maximum suppression is performed to thin the edges. Overall, the edge-detected image provided a decent output.

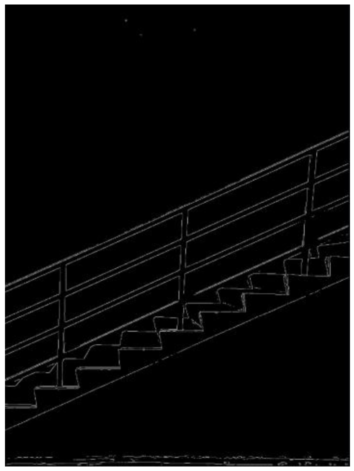
- To refine the edges, A binary threshold is applied to the edge magnitude to produce a clear edge map. The Hough transform is applied to the thresholded image using myHoughTransform. This step computes an accumulator matrix, along with corresponding rho and theta scales, to represent potential lines in the image. When I set threshold to 0.3, pixels with a strong gradient response were retained. The difference between the raw edge detection and this threshold version was different as this step removed most of the weaker edges and background noise while keeping the major elements intact.

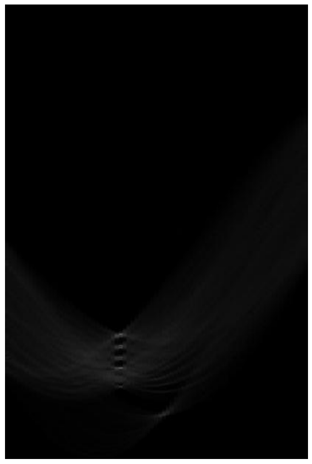
- Two sets of line segments are extracted:
  Red segments via myHoughLineSegments, which detects segments by analyzing the Hough accumulator.
  Green segments via detectGreenLines, offering an alternative detection approach.Both functions return lists of segments, each defined by start and end coordinates. The resulting Hough space visualization indicated strong, bright regions corresponding to strong lines in the image for the diagonal edges of the staircase. However, thinner structures, such as the railings, were not as prominent.

- Lastly, The custom approach worked well for capturing clear, strong edges like the staircase's main boundaries, but it sometimes resulted in broken line segments rather than continuous lines. This likely happened because gaps in the thresholded edge map interrupted some of the detected edges. However, it occasionally introduced extra lines that were likely caused by noise or less significant edges.


(a) Input Image


(b) Edge


(c) Threshold


(d) Hough

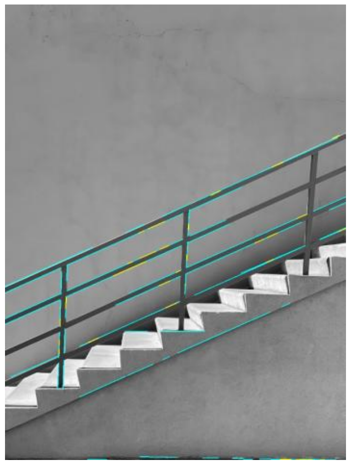

(e) Lines

Figure 6: Input Image and Corresponding Results

Citations

- For hough Transform I referred to https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html https://scikitimage.org/docs/0.23.x/auto_examples/edges/plot_line_hough_transform.html
- For HoughLines I referred to https://www.mathworks.com/help/images/ref/houghlines.html
- For Image Filtering I referred to https://www.geeksforgeeks.org/image-filtering-using-convolution-in-opencv/
- for Edge Detection I referred to https://learnopencv.com/edge-detection-using-opencv/
- For Hough Line Segments I referred to https://stackoverflow.com/questions/70614351/opencv-houghlines-produces-too-many-lines-python
- for 'ec' I used the HoughScriptWithSegment.py code itself as it gave me decent output.

Note:

- I have used 3x3 window size in houghScriptWithSegments.py because it was giving me accurate output and was more optimized as compared to 7x7 window size.
- Parameters:
  Sigma: 1.2
  threshold: 0.3
  rhoRes: 2
  thetaRes: np.pi/ 180
  nLines: 15