CS557: Code Project 3

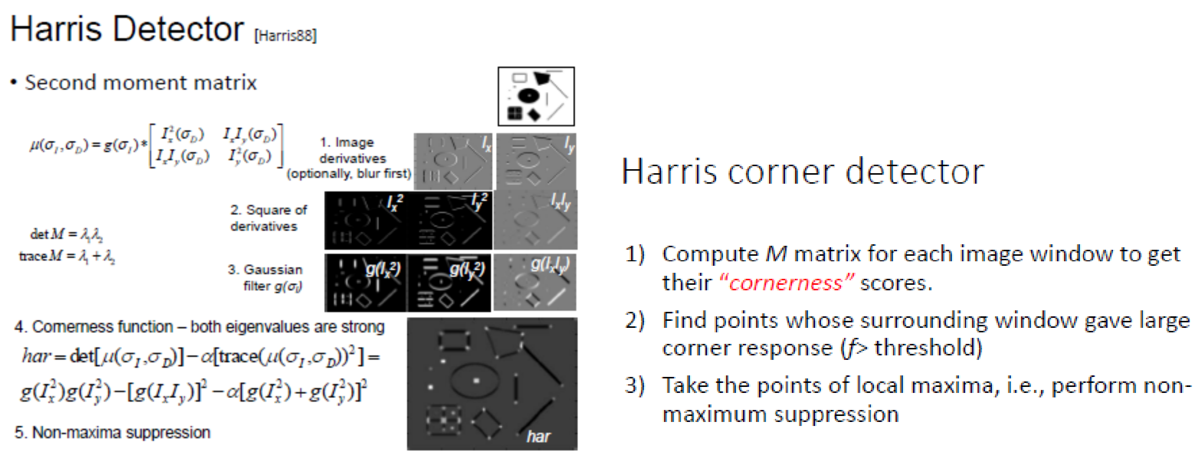
Md Nazmuzzaman Khan

Introduction

The goal of this project is to implement Harris corner detector, Canny edge detector and Hough transform line detector. Get hands on idea about how different parameters affect these algorithms. Then compare with available packages of these algorithm. Python 3.5 with OpenCV, Matplotlib, Numpy, Scipy and Math packages were used in this project.

Problem 1.1

- Implement the Harris-Stephens corner detection method.
- Implement the thresholding and non-maxima suppression to obtain your corners in the image.



Harris Detector [Harris88]

• Second moment matrix

$$\mu(\sigma_I,\sigma_D) = g(\sigma_I) * \begin{bmatrix} I_x^2(\sigma_D) & I_xI_y(\sigma_D) \\ I_xI_y(\sigma_D) & I_y^2(\sigma_D) \end{bmatrix}$$

1. Image derivatives (optionally, blur first)

$\det M = \lambda_1\lambda_2$

$\operatorname{trace} M = \lambda_1 + \lambda_2$

2. Square of derivatives

3. Gaussian filter $g(\sigma_I)$

4. Cornerness function – both eigenvalues are strong

$$har = \det[\mu(\sigma_I,\sigma_D)] - \alpha[\operatorname{trace}(\mu(\sigma_I,\sigma_D))^2] = g(I_x^2)g(I_y^2) - [g(I_xI_y)]^2 - \alpha[g(I_x^2)+g(I_y^2)]^2$$

5. Non-maxima suppression

Harris corner detector

1) Compute *M* matrix for each image window to get their *"cornerness"* scores.
2) Find points whose surrounding window gave large corner response (*f*> threshold)
3) Take the points of local maxima, i.e., perform non-maximum suppression

Fig 1: Harris corner detector steps. α=0.05 used for implemented code.

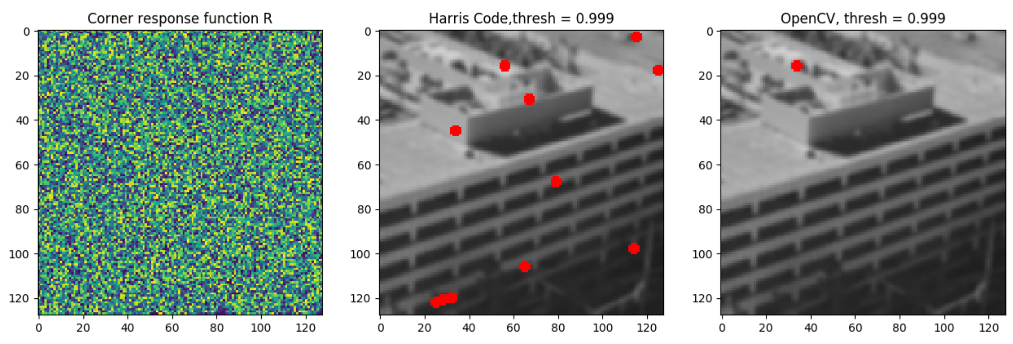


Fig 2: "Building" with σ=1 gaussian blur as step 1. Threshold value 0.999

CS557: Code Project 3
Md Nazmuzzaman Khan

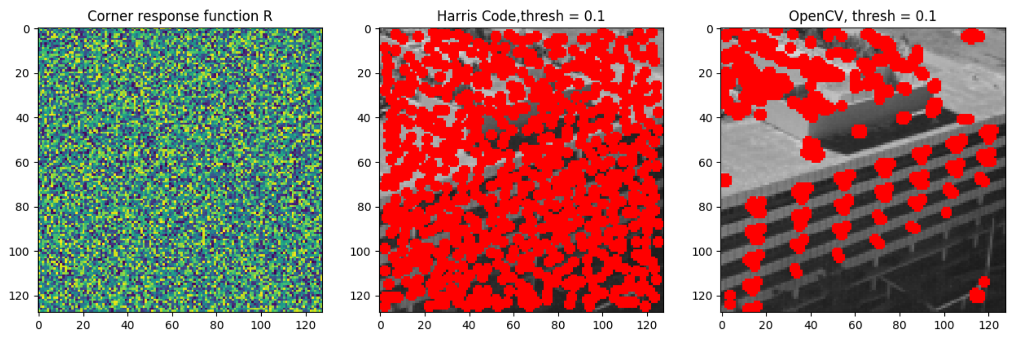


Fig 3: “Building” with σ=1 gaussian blur as step 1. Threshold value 0.1

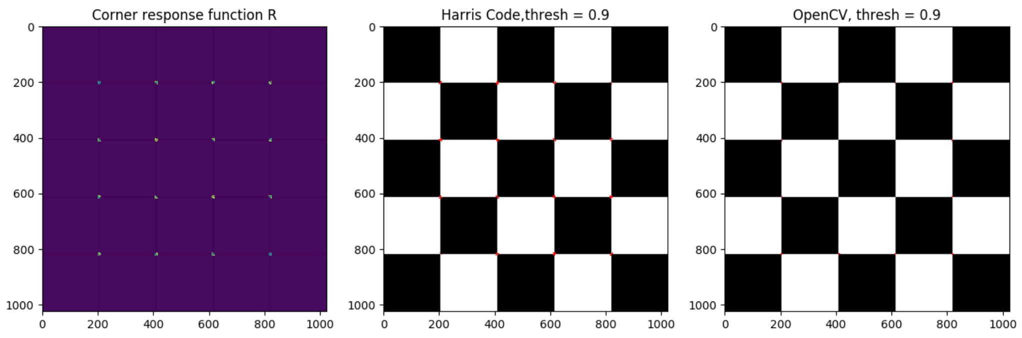


Fig 4: “Checkerboard” with σ=1 gaussian blur as step 1. Threshold value 0.9.

CS557: Code Project 3
Md Nazmuzzaman Khan

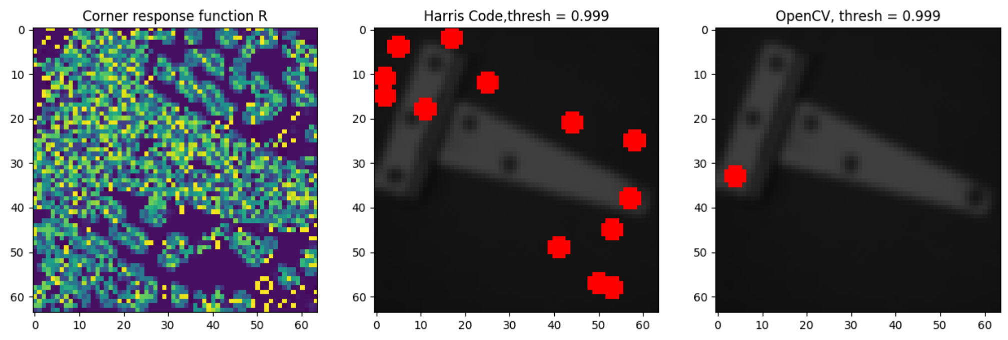


Fig 5: “hinge” with σ=1 gaussian blur as step 1. Threshold value 0.999.

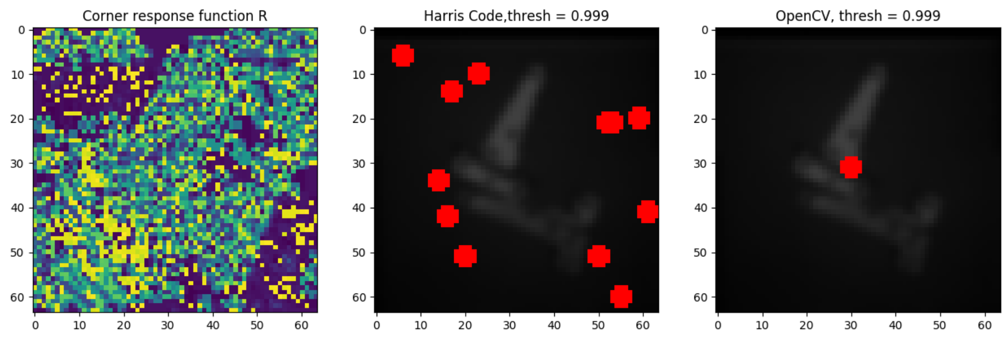


Fig 6: “hinges” with σ=1 gaussian blur as step 1. Threshold value 0.999.

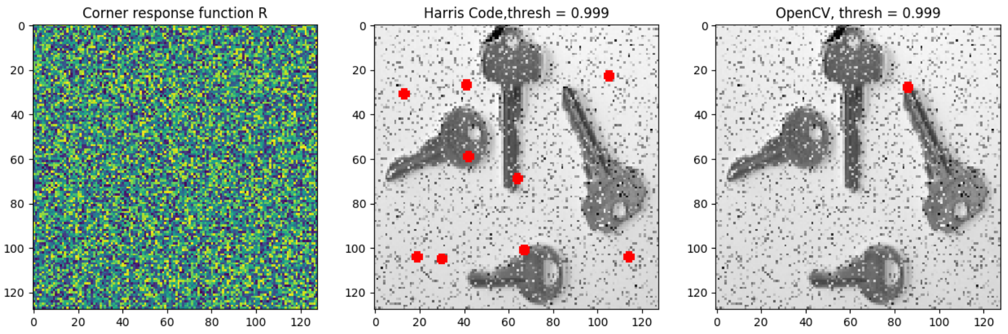


Fig 7: "keys" with σ=1 gaussian blur as step 1. Threshold value 0.999.

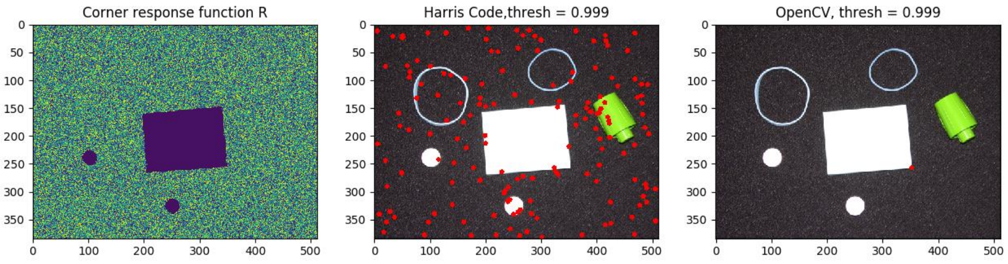


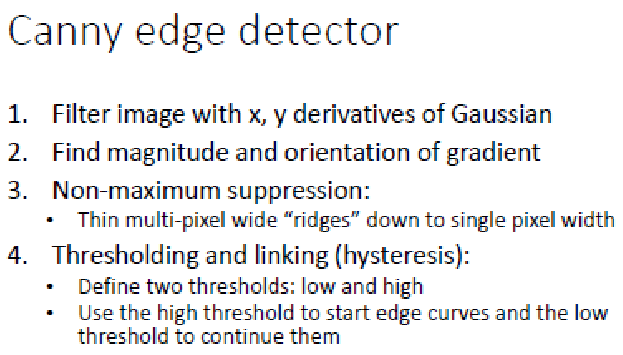
Fig 8: "pillset" with σ=1 gaussian blur as step 1. Threshold value 0.999.

Observations:

Edge detection hugely relies on image noises. For cases with no noise (checkerboard), implemented method performed well compared to OpenCV implementation. Corner response function and non-maxima suppression algorithm worked well. But for noisy images (Fig 3, building), OpenCV performed well. Which suggests that, it may contain some adaptive noise reduction algorithm built-in. For images where gaussian blur was able to suppress some noise (Fig 5 and 6), implemented algorithm was able to pick up the edges (although somewhat randomly).

Problem 1.2

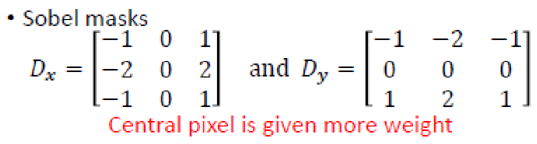
- Implement the Canny edge detector.
- Implement the directional Gaussian derivatives, non-maxima suppression, and hysteresis thresholding to find the true edges.



Canny edge detector

1. Filter image with x, y derivatives of Gaussian
2. Find magnitude and orientation of gradient
3. Non-maximum suppression:
   - Thin multi-pixel wide "ridges" down to single pixel width
4. Thresholding and linking (hysteresis):
   - Define two thresholds: low and high
   - Use the high threshold to start edge curves and the low threshold to continue them

Step.1 Use Sobel filter to get x (Ix), y (Iy) derivative of the gaussian blurred image (σ=1)



- Sobel masks

$$D_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } D_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Central pixel is given more weight

Step.2

Magnitude = $\sqrt{Ix^2 + Iy^2}$

Gradient = $\tan^{-1}(\frac{Iy}{Ix})$

Step.3 Non-maxima suppression

```
For height and width of "Magnitude" image:
        For each 3x3 kernel:
                Check if center pixel is higher than
                surrounding pixel using Gradient
                values:
                If yes: surrounding pixel=0
                If no: unchanged
        end
end
```

Step.4

```
Start at a pixel with lower threshold:
   if connected to Upper threshold:
          keep
   else:
          delete
```

For all the images:

σ = 1

Upper threshold = 255
Lower threshold = 50

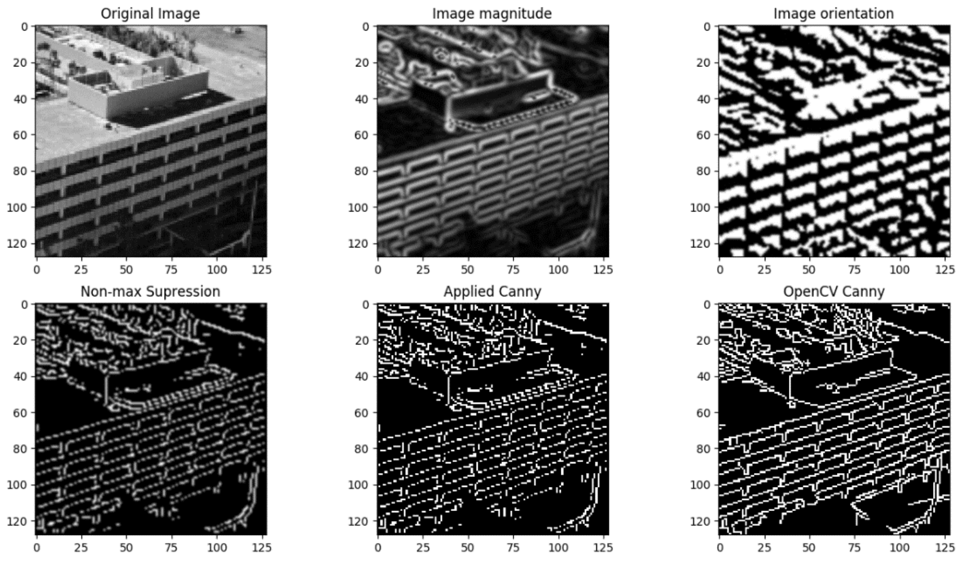


Fig 9: Canny edge detection implementation for ‘building’ with OpenCV comparison.

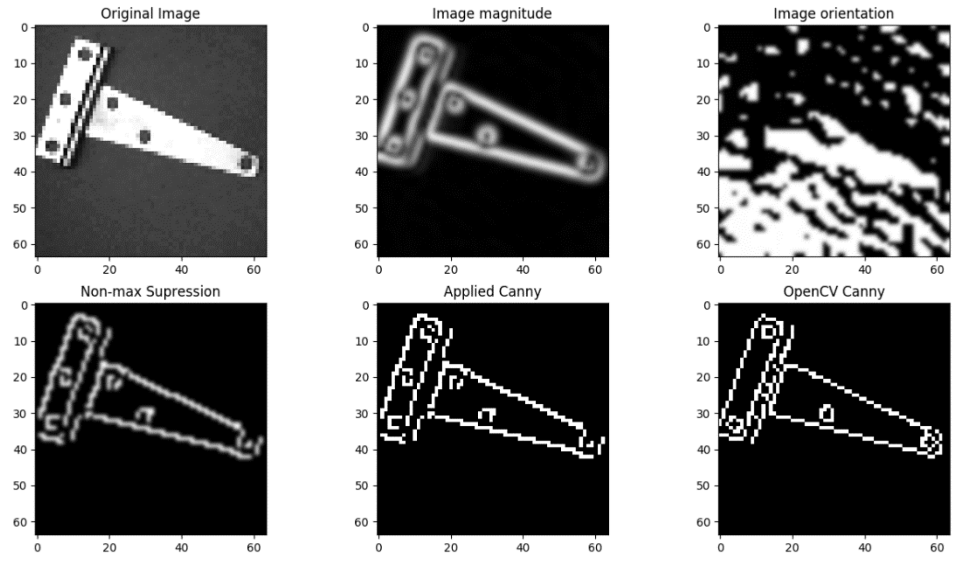


Fig 10: Canny edge detection implementation for ‘hinge’ with OpenCV comparison.

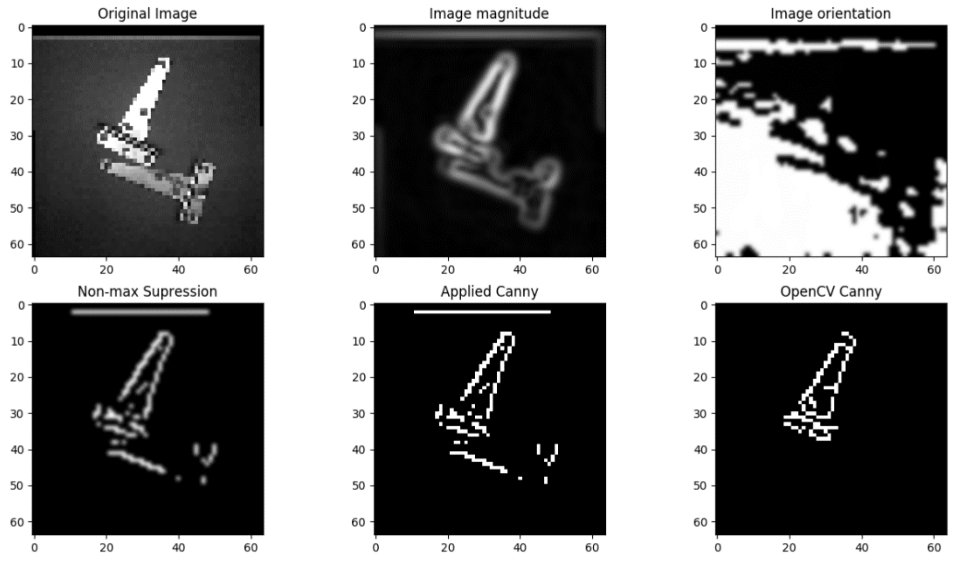


Fig 11: Canny edge detection implementation for ‘hinges’ with OpenCV comparison.

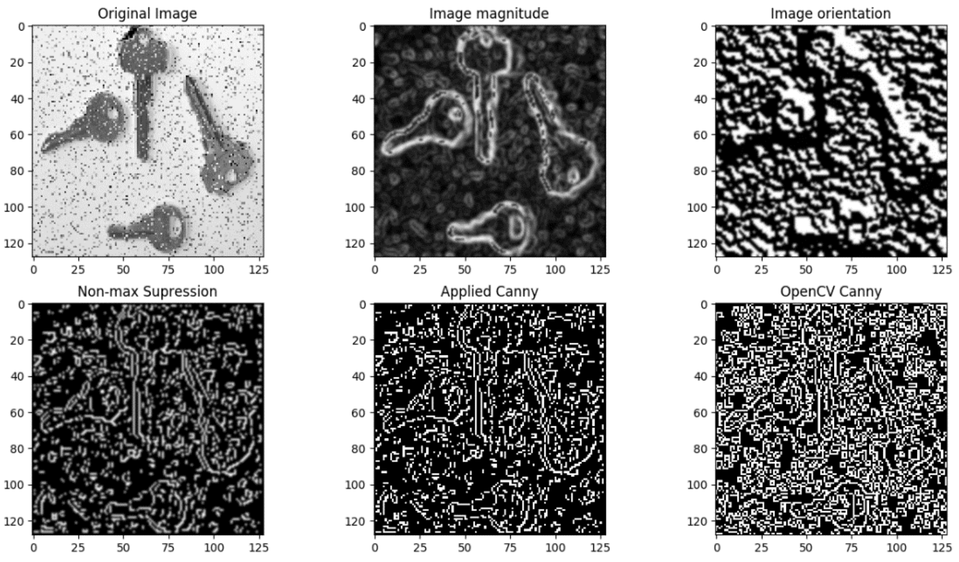


Fig 12: Canny edge detection implementation for ‘keys’ with OpenCV comparison.

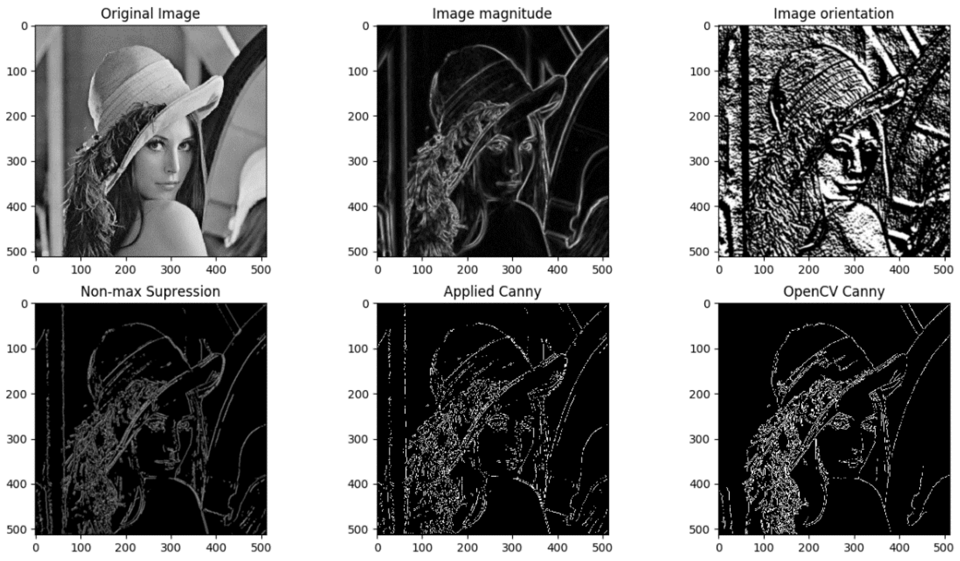


Fig 13: Canny edge detection implementation for ‘Lena’ with OpenCV comparison.

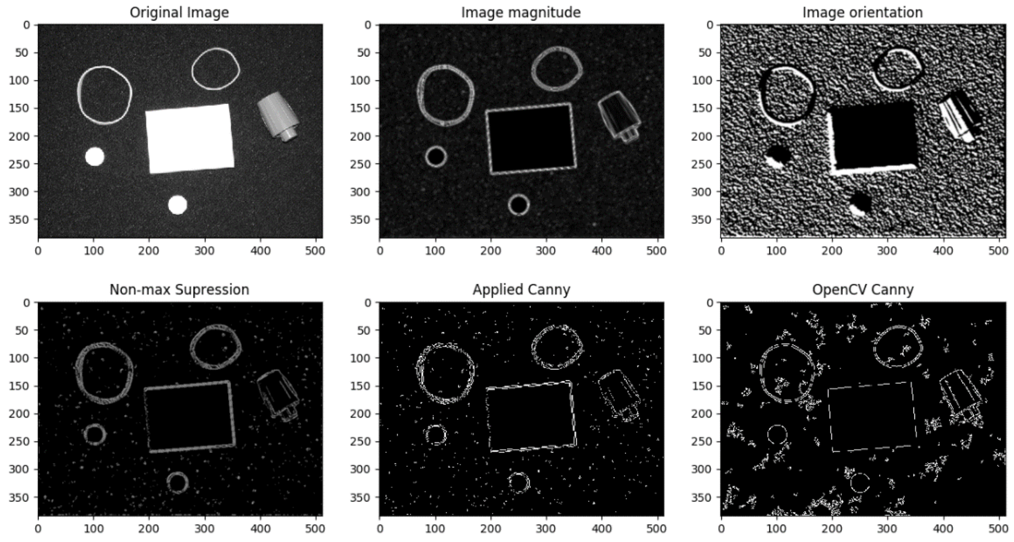


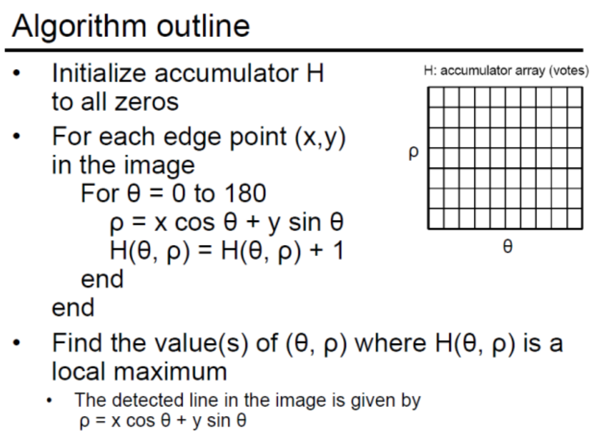
Fig 14: Canny edge detection implementation for ‘Pillset’ with OpenCV comparison.

Observations:

For all the cases implemented Canny edge detection worked well compared to OpenCV. Noise suppression is really important. For salt and pepper noise (Fig 12) edge detection performance plummeted. For non-maximum suppression, did not use interpolation. Arranged the angle so that all the neighboring 8 pixels can be calculated directly. OpenCV possibly uses an interpolation method for aggressive suppression. Which is seen in Fig 11 and 13 that, background lines are omitted in OpenCV implementation.

Problem 1.3

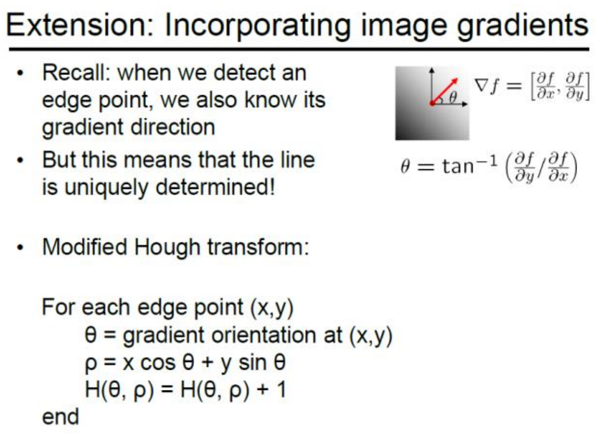
- Write a program that will take the output edges detected in Problem 1.2 and organize them into straight lines using Hough Transform.
- Solve the following issues:
- How to get the parameters from the position and gradient direction information?
- What quantization to use for the parameter space?
- How to deal with counts being split into adjacent bins. Your textbook hints at a solution.
- How to perform the peak detection? Thresholding?
- How to do the back projection?



Algorithm outline

```
• Initialize accumulator H
  to all zeros
• For each edge point (x,y)
  in the image
      For θ = 0 to 180
          ρ = x cos θ + y sin θ
          H(θ, ρ) = H(θ, ρ) + 1
      end
  end
• Find the value(s) of (θ, ρ) where H(θ, ρ) is a
  local maximum
    • The detected line in the image is given by
      ρ = x cos θ + y sin θ
```

H: accumulator array (votes)
ρ
θ



Extension: Incorporating image gradients

- Recall: when we detect an edge point, we also know its gradient direction
- But this means that the line is uniquely determined!

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$$

$$\theta = \tan^{-1}\left(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x}\right)$$

- Modified Hough transform:

```
For each edge point (x,y)
    θ = gradient orientation at (x,y)
    ρ = x cos θ + y sin θ
    H(θ, ρ) = H(θ, ρ) + 1
end
```

For all the images:

σ = 1

Upper threshold = 255
Lower threshold = 50

Bin sizes are indicated on image.

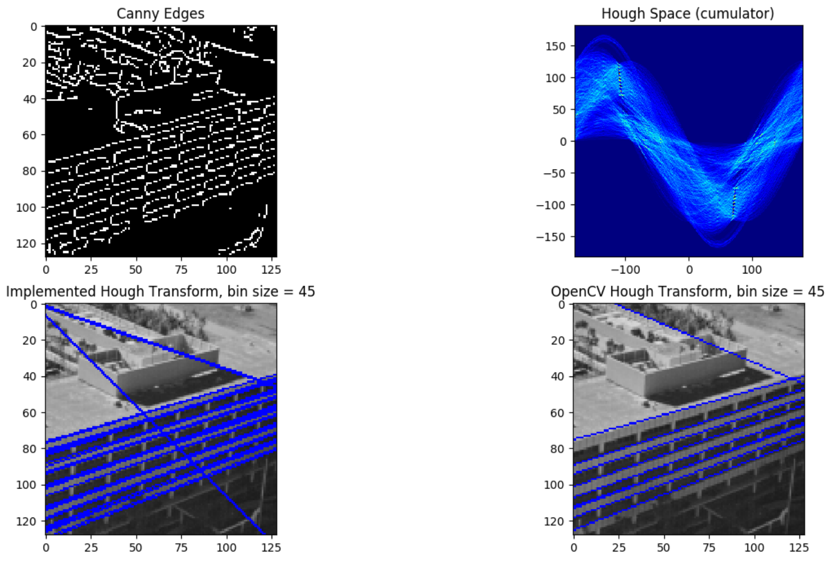


Fig 15: Line detection using Hough Transform implementation for ‘building’ with OpenCV comparison.

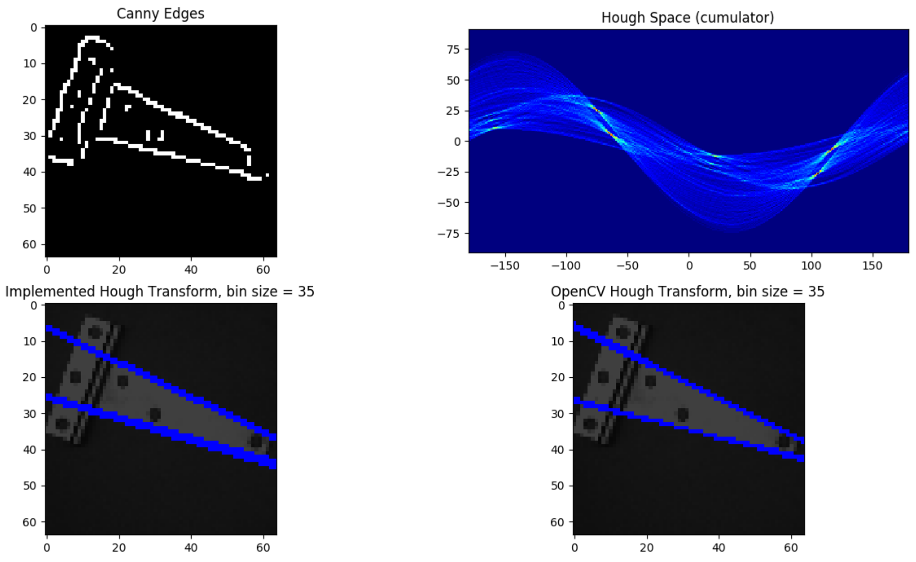


Fig 16: Line detection using Hough Transform implementation for ‘hinge’ with OpenCV comparison.

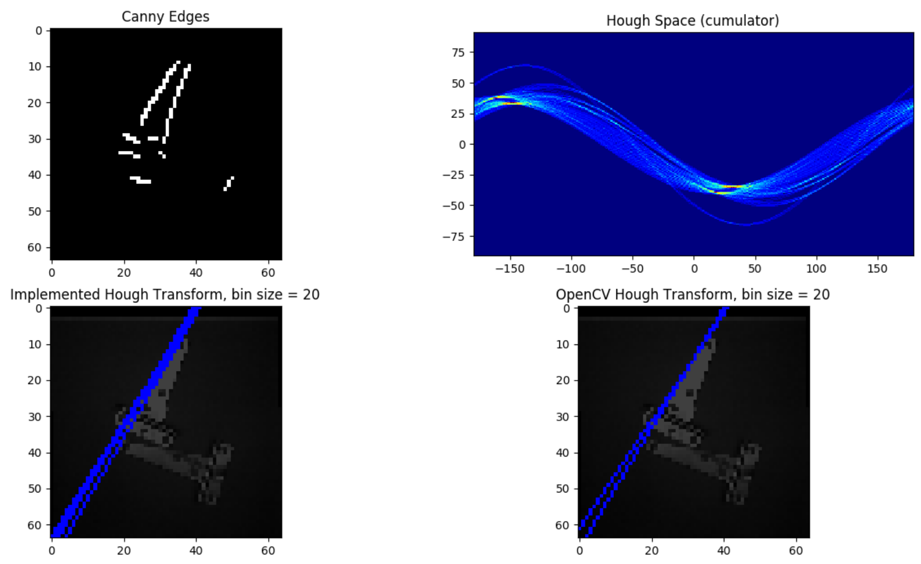


Fig 17: Line detection using Hough Transform implementation for ‘hinges’ with OpenCV comparison.

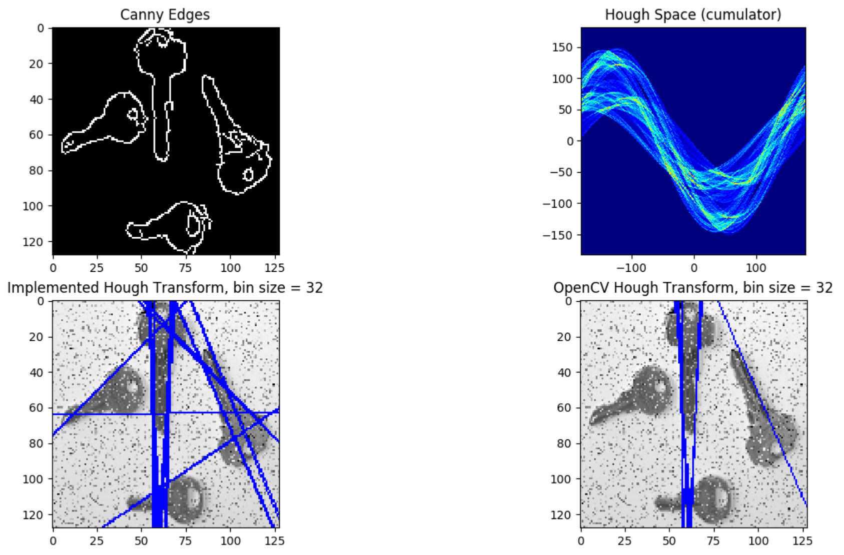


Fig 18: Line detection using Hough Transform implementation for ‘keys’ with OpenCV comparison.

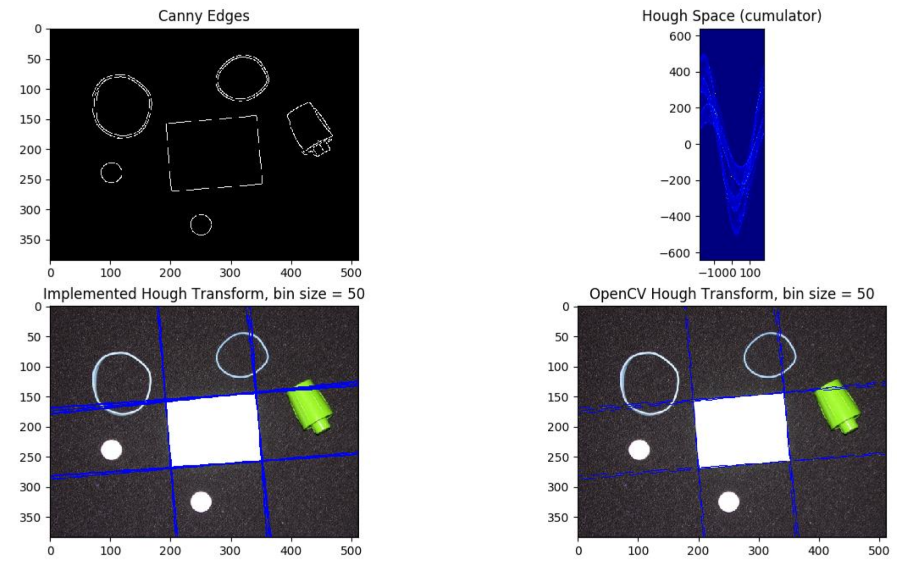


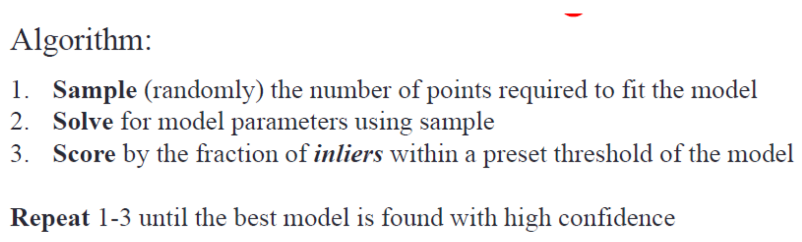
Fig 19: Line detection using Hough Transform implementation for 'pillset' with OpenCV comparison.

Observations:

In the step "incorporating image gradient", theta in this case is a float. But for H (accumulator) input, theta has to be integer. Was unable to figure out the data structure for theta in modified case, as a result didn't implement it. But As seen above, for all the cases, implemented algorithm was able detect lines robustly compared to OpenCV implementation. OpenCV definitely has gradient information, as a result, less amount of lines detected for same threshold values. Bin size is the most important parameter here. All the images require different bin size value to detect lines.

Bonus Problem

- implement the line detection from the edge points using the RANSAC algorithm
- include a discussion of how it compares with the Hough transform, in terms of performance and ease of implementation



Algorithm:

1. **Sample** (randomly) the number of points required to fit the model
2. **Solve** for model parameters using sample
3. **Score** by the fraction of ***inliers*** within a preset threshold of the model

Repeat 1-3 until the best model is found with high confidence

Least square fit method was used to calculate line parameter 'm' and 'b'. Following parameters were kept fixed during simulation:

CS557: Code Project 3
Md Nazmuzzaman Khan

```
max_iters=1000, samples_to_fit=2, inlier_threshold=0.99,
min_inliers=100
```

As the goal is to understand how RANSAC performs in comparison to Hough transform, several simulations are performed on 'pillset' image. Line fitting segment of the code is inspired from https://medium.com/@iamhatesz/random-sample-consensus-bd2bb7b1be75.

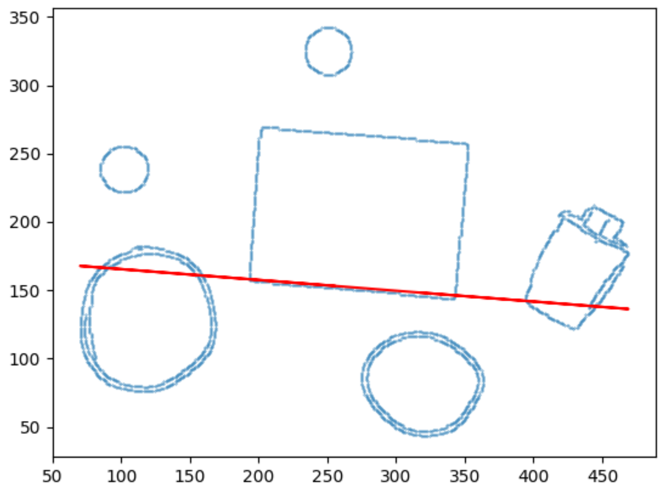


Fig 20: simulation = 1 of RANSAC line fitting on 'pillset'. Calculated Least Sq. [m b]: [-7.50000e-02 2.84325e+02].

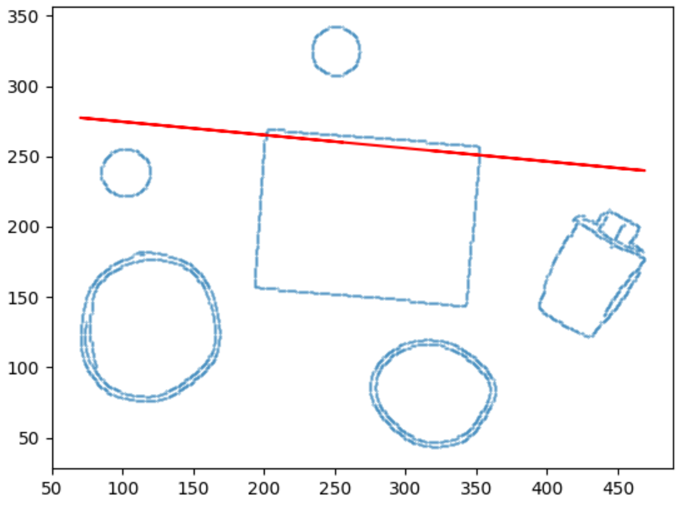


Fig 21: simulation = 2 of RANSAC line fitting on 'pillset'. Calculated Least Sq. [m b]: [-9.83606557e-02 1.77147541e+02]

CS557: Code Project 3
Md Nazmuzzaman Khan

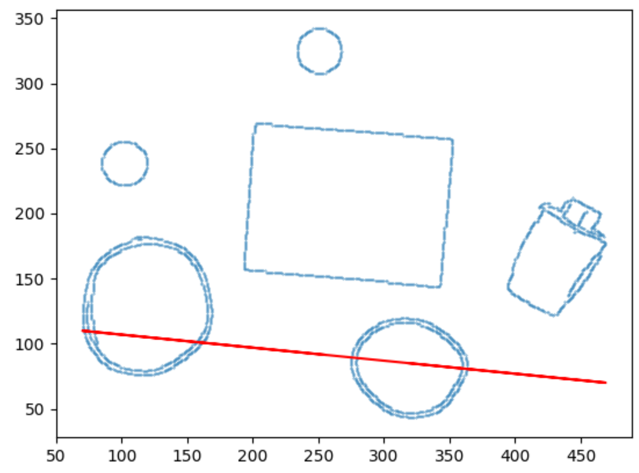


Fig 22: simulation = 3 of RANSAC line fitting on 'pillset'. Calculated Least Sq. [m b]: [-8.54700855e-02 1.73435897e+02]

Observations:

Implementation of the code was easy and also very easy to understand. It can successfully detect straight lines with some tuning of the "inliers" parameter (Fig 20 and 21). But just implementing RANSAC without any added condition will create error lines. Because we are only using 2 points to fit lines (Fig 22). Also, to successfully detect all the lines on an image, it needs to run multiple amount of times. Because to detect 1 successful line, RANSAC needs to complete maximum 1000 iterations in this case.

Advantages:

Simple and general

Applicable to many different problems, often works well in practice

Robust to large numbers of outliers

Applicable for larger number of parameters than Hough transform

Parameters are easier to choose than Hough transform

Disadvantages:

Computational time grows quickly with the number of model parameters

Sometimes problematic for approximate models

CS557: Code Project 3
Md Nazmuzzaman Khan