**INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR**

DIGITAL IMAGE PROCESSING LABORATORY A REPORT ON

## TERM PROJECT

**1. Implementation of Canny Edge Detector from scratch.**

**2. Implement (from scratch) Hough Transform to identify rectangular patches in an image**

**3. Count the number of rectangles in the given attached image**

**4. The Hough transform should use the edge detection algorithm developed by us**

## 19.11.2020

**SOURAV PATEL(20EC65R25)**

**DISHANT SATULEY(20EC65R03)**

**DEPT OF ELECTRONICS AND ELECTRICAL COMMUNICATION ENGINEERING**

### VISUAL INFORMATION AND EMBEDDED SYSTEMS

**Contents**

1. [Abstract 1](#_bookmark0)
2. [Introduction 1](#_bookmark1)
3. [Working Model 2](#_bookmark2)
4. Gaussian Blur 3
5. Determining intensity gradient 4
6. Non Maximum Suppression 5
7. Double Thresholding 6
8. Edge Tracking by Hysteresis 7
9. Cleaning Up 7

10. The Hough Transform 8

11. Line Detection 9

12. Algorithm 10

13. Result & Analysis 11

14. Conclusion 13

15. References 13

# Abstract

In today’s modern life, there is increased demand of edge detection of the image for medical and defence applications. So, it is required to study the edge detection algorithm. The various edge detection algorithms such as Prewitt, Robert, Sobel etc. are failed to meet the low area and reduced delay. The purpose of edge detection in general is to significantly reduce the amount of data in an image, while preserving the structural properties to be used for further image processing. Several algorithms exists, and this worksheet focuses on a particular one developed by John F. Canny (JFC) in 1986 . Even though it is quite old, it has become one of the standard edge detection methods and it is still used in research .

The aim of JFC was to develop an algorithm that is optimal with regards to the following

criteria:

**1.** Detection: The probability of detecting real edge points should be maximized while the

probability of falsely detecting non-edge points should be minimized. This corresponds to

maximizing the signal-to-noise ratio.

**2.** Localization: The detected edges should be as close as possible to the real edges.

**3.** Number of responses: One real edge should not result in more than one detected edge

(one can argue that this is implicitly included in the first requirement).

With JFC’s mathematical formulation of these criteria, Canny’s Edge Detector is optimal for

a certain class of edges (known as step edges). A Python implementation of the algorithm has

been written, and this will be further described. The images used throughout this

worksheet are generated using this implementation.

**Introduction:**

A fundamental task in application of computer vision is the edge detection[1] since from the early 1970's. Edge detection is an important pre-processing step for tasks including segmentation, active contours and object recognition. Edge detection refers to the process of identifying and locating sharp discontinuities in an image. The abrupt changes in pixel intensities characterizes the boundaries[2] of objects in an image. Classical edge detection methods[3] uses the convolution on the spatial domain of the image with an operator. An operator (a 2-D filter) is constructed to be sensitive to large gradients in the image while returning values of zero in uniform regions. Large number of edge detection operators[4] available each operator is designed to be detection of certain type of edges[3]. Edge orientation, noise and edge structure determine the choice of edge detection operator. The geometry of the operator determines a characteristic direction in which it is most sensitive to edges. Operators can be used to look for horizontal, vertical or diagonal edges. In noisy images detection of edge[5] is a difficult task, since both the edges and noise contain high frequency content. Reduction of noise results distorted and blurred images. Edge detectors for noisy images can average enough data to minimize localized noisy pixels. This results in a reduced amount of accurate localization of the detected edges. Due to noise, there are problems of edge localization, high computational time, missing true edges and false edge detection will occur. Many methods are available for detection of edges. Edge detection operators are grouped into two categories Gradient based and Laplacian based. Differentiation of image is the most common edge detection method. Gradient operators use first-order derivatives of the image by looking for the maximum and minimum. Laplacian operators searches for zero crossings in the second-order derivatives of the image to find edges.

# Working Model

The Canny edge detector needs to be implemented in a specific manner and that’s why it is the only method. Below I have listed down the sequence to do it.

Step 1 - Grayscale Conversion

Step 2 - Gaussian Blur

Step 3 - Determine the Intensity Gradients

Step 4 - Non Maximum Suppression

Step 5 - Double Thresholding

Step 6 - Edge Tracking by Hysteresis

Step 7 - Cleaning Up

All these steps are described in this paper and there implementation by us is also noted down.

# (i) Grayscale Conversion

Converted the image to grayscale. It is done using the inbuilt function

# (ii) Gaussian Blur

# In image processing, a Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function (named after mathematician and scientist Carl Friedrich Gauss)

# It is a widely used effect in graphics software, typically to reduce image noise and reduce detail. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen, distinctly different from the bokeh effect produced by an out-of-focus lens or the shadow of an object under usual illumination.

# Gaussian smoothing is also used as a pre-processing stage in computer vision algorithms in order to enhance image structures at different scales.

A sigma of 1.4 is used in this example and was determined through trial and error.

The 3 Gaussian kernel used are as followed

Gaussian kernel 1=

[0.093124, 0.0118914, 0.093124]

[0.118914, 0.151845, 0.118914

[0.093124, 0.118914, 0.093124]

Gaussian kernel 2 =

[0.102059, 0.115349, 0.102059]

[0.115349, 0.130371, 0.115349]

[0.102059, 0.115349, 0.102059]

Gaussian kernel 3 =

[0.012841, 0.026743, 0.03415, 0.026743, 0.012841]

[0.026743, 0.055697, 0.071122, 0.055697, 0.026743]

[0.03415, 0.071122, 0.090818, 0.071122, 0.03415]

[0.026743, 0.055697, 0.071122, 0.055697, 0.026743]

[0.012841, 0.026743, 0.03415, 0.026743, 0.012841]

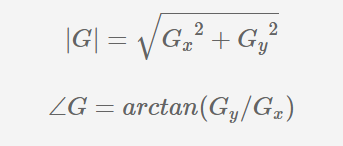
**(iii) Determining The Intensity Gradient:**

Edge is all about sudden intensity change. The intensity change of the pixel is edge.

Now the Sobel operator has to applied over the input image and the steps and sequences remain the same as the process explained in the Sobel edge detection process. The resultant Sobel operated images presented below. This is referred as gradient magnitude of the image.

We preferred Sobel operator and it is the general approach but it is not a mandated rule to always go with the Sobel operator it can be any gradient operator and the result should be the gradient magnitude of the image.

The resultant gradient approximation can be calculated with



The G will be compared against the threshold and with which one can understand the taken point is an edge or not.

The formula for finding the edge direction is just

Theta = inv tan(Gy/Gx)

The kernel value of the Sobel operator used here is as followed:

Sobel kernelX =

[1, 0, -1]

[2, 0, -2]

[1, 0, -1]

Sobel kernelY =

[1, 2, 1]

[0, 0, 0]

[-1, -2, -1]

Sobel kernelX2 =

[-1, -2, 0, 2, 1]

[-4, -10, 0, 10, 4]

[-7, -17, 0, 17, 7]

[-4, -10, 0, 10, 4]

[-1, -2, 0, -2, 1]

Sobel kernelY2 =

[1, 4, 7, 4, 1]

[2, 10, 17, 10, 2]

[0, 0, 0, 0, 0]

[-2, -10, -17, -10, -2]

[-1, -4, -7, -4, -1]

**(iv) Non Maximum Suppression:**

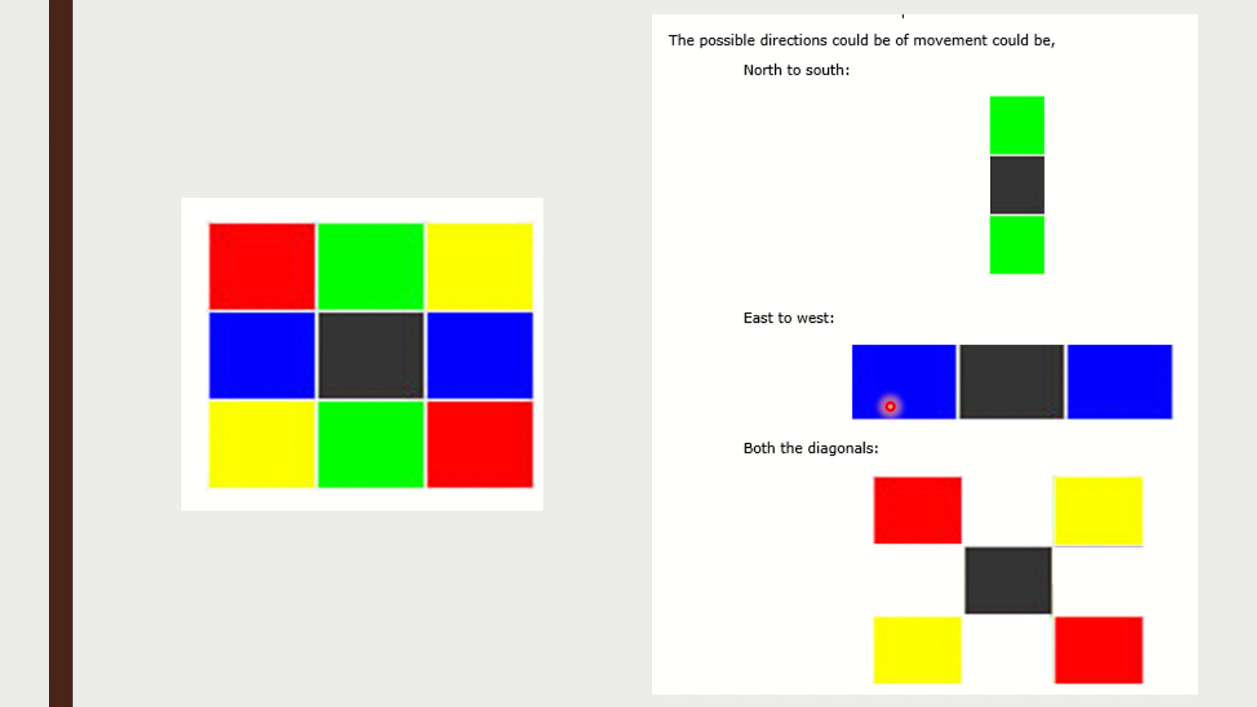
This is the next step in the sequence. The gradient magnitude operators discussed in the previous stage normally of tense Thick edges but the final image is expected to have thin edges.

Hence the process non maximum suppression enable us to derive thin edges from thicker one through the following steps.

We have the edge direction already available with us. The subsequent step is to relate the identified as direction to a DIRECTION that can be sketched in the image. i.e. Ideally, it is a prediction of how the movement of edges could happen.

An example is always handy and we have taken 3\*3 matrix as a reference. It is all about the colors and the below is to be visualized as three cross three Matrix for the scenario being discussed.





The center cell is the region of interest for us. It is important to understand this point explained below.

There can be only four possible directions for any pixel they are

1- 0 degrees

2- 45 degrees

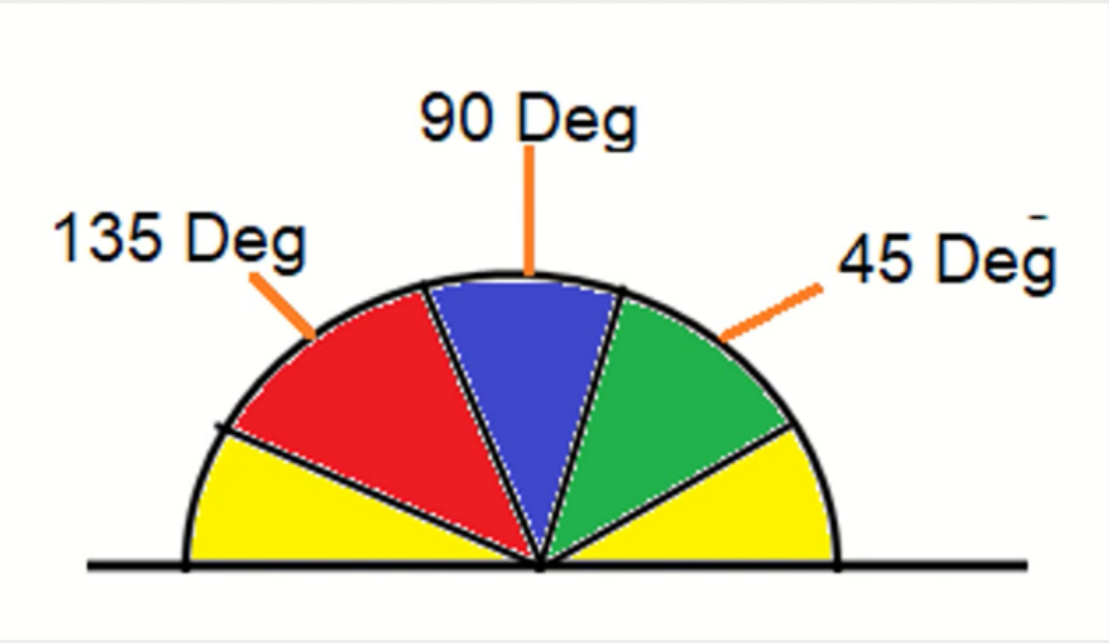
3- 90 degrees

4- 135 degrees

Hence it forces us to a situation where the edge has to be definitely oriented to one of these four directions.

This is kind of approximation where if the orientation angle is observed to be 5 degrees then it is taken as 0 degrees. Similarly if it is 43 degrees it cell be made as 45 degrees.

For ease of understanding we have drawn a semicircle with color shading. It is representing 180 degrees but actual scenario is for 360 degrees



With the help of picture as reference the following rules are to be framed:

1. Any edge which comes under the yellow range is set to 0 degrees(which means from 0 to 22.5 degree 57.5 degrees are set to 0 degrees)
2. Any as which come under the green range is all set to 45 degrees.(which means 22.5 degrees to 67.5 is 45 degrees)
3. Any age coming under the blue range is all set to 90 degrees.(which means 67.5 to 112.5 degrees is set as 90 degrees)
4. The last remains easy-to-understand any age coming under the red range is all set to 135 degrees(which means 112.5 to 157.5 degrees is set as 135 degrees)

After this process, direction of the ages is mapped to any of the four directions mentioned above. The input image now shall be like the one presented below were the directions of the edges are appropriately mapped.

* The last step in the process comes now.
* The edge directions are all determined and the non maximum suppression is to be applied.
* Non maximum suppression as the name suggests, is a process for separation of the pixels 2019 not be considered as an edge is carried out. This will enable the system to generate a thin line in the output image as shown below.
* These results are obtained below the thresholding and as expected the next is to perform thresholding and smoothing.

**(v)****Double Thresholding:**

As one could see from the previous stage results, non maximum thresholding has not provided as excellent results.

* there is still some noise. The image when raises a thread in mind that some of the ages Sone Mein not really be so some major School be mixed in the process. Hence, there has to be a process to address this challenge. The process to be followed is double thresholding.

We need to go with double thresholding and in this process we have to set two thresholds on a high and another a low.

Simple assume high threshold value as 0.7. Any pixel with Value above 0.7 is to be seen as strong edge.

* Another threshold the lower one can be 0.3. In that case any pixel below this value is not an edge at all and hence set them all to 0.

And all the values in between they may or may not be an edge. They are referred as a weak edge. There has to be a process to determine which of the weak edges are actual edges so as to not to miss them.

**(vi) Edge Tracking by Hysteresis:**

* As discussed in the previous section, it is important for us to now understand which of the weaker edge are actual edges.
* Simple approaches to be followed. We can call the weak edges connected to strong edges as strong/ actual edges and retain them. Weak edges which are not connected to stronger ones are to be removed.

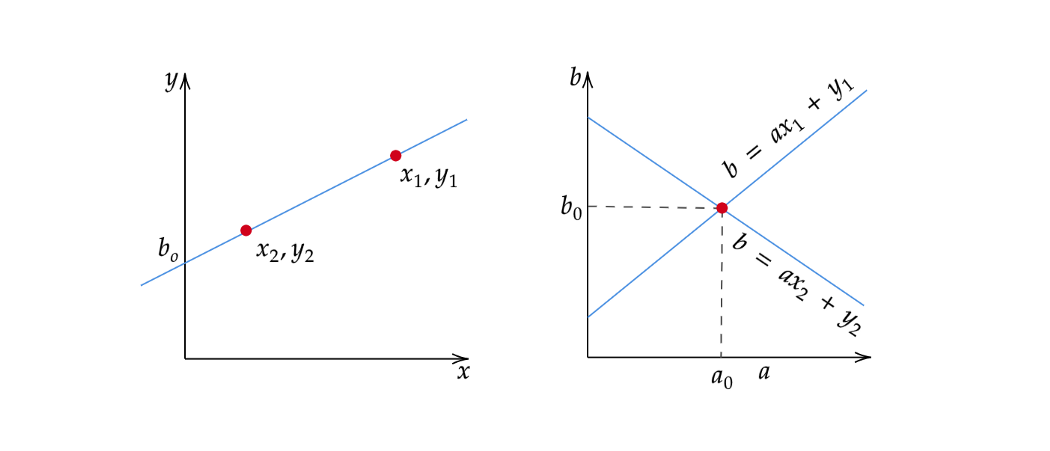
**(vii) Cleaning Up:**

All the remaining with ages can be removed and that is it. The process is complete. Once this process is done an, we could get the following output images as the result.

**The Hough Transform:**

The Hough Transform is an algorithm patented by Paul V. C. Hough and was originally invented to recognize complex lines in photographs (Hough, 1962). Since its inception, the algorithm has been modified and enhanced to be able to recognize other shapes such as circles and quadrilaterals of specific types. In order to understand how the Hough Transform algorithm works, it is important to understand four concepts: edge image, the Hough Space and the mapping of edge points onto the Hough Space, an alternate way to represent a line, and how lines are detected.

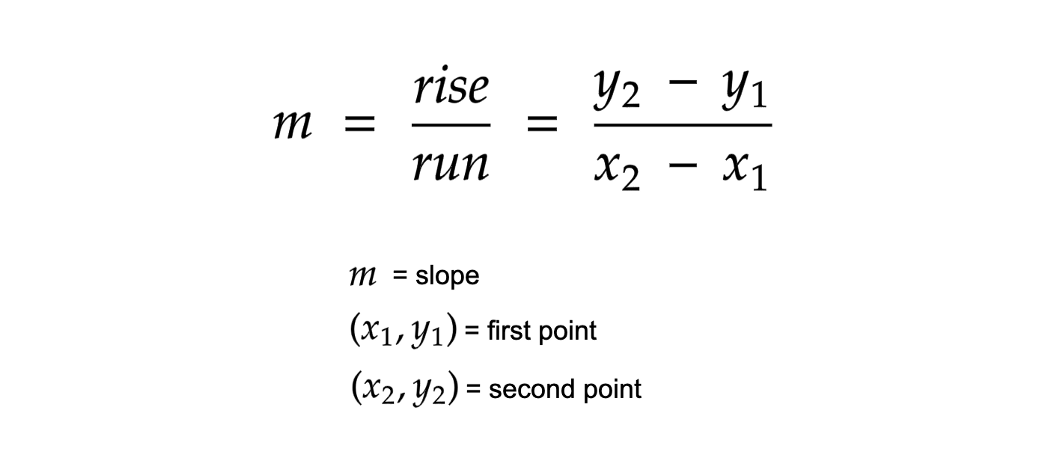
**The Hough Space and the Mapping of Edge Points onto the Hough Space:**

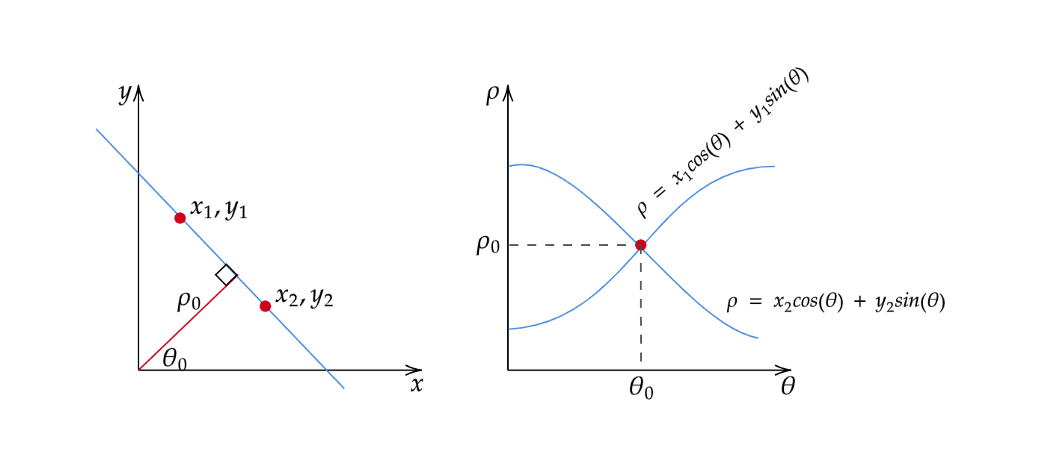


The Hough Space is a 2D plane that has a horizontal axis representing the slope and the vertical axis representing the intercept of a line on the edge image. A line on an edge image is represented in the form of y = ax + b (Hough, 1962). One line on the edge image produces a point on the Hough Space since a line is characterized by its slope a and intercept b. On the other hand, an edge point (xᵢ, yᵢ) on the edge image can have an infinite number of lines pass through it. Therefore, an edge point produces a line in the Hough Space in the form of b = axᵢ + yᵢ (Leavers, 1992). In the Hough Transform algorithm, the Hough Space is used to determine whether a line exists in the edge image.

**An Alternate Way to Represent a Line:**

There is one flaw with representing lines in the form of y = ax + b and the Hough Space with the slope and intercept. In this form, the algorithm won’t be able to detect vertical lines because the slope a is undefined/infinity for vertical lines (Leavers, 1992). Programmatically, this means that a computer would need an infinite amount of memory to represent all possible values of a. To avoid this issue, a straight line is instead represented by a line called the normal line that passes through the origin and perpendicular to that straight line. The form of the normal line is ρ = x cos(θ) + y sin(θ) where ρ is the length of the normal line and θ is the angle between the normal line and the x axis.

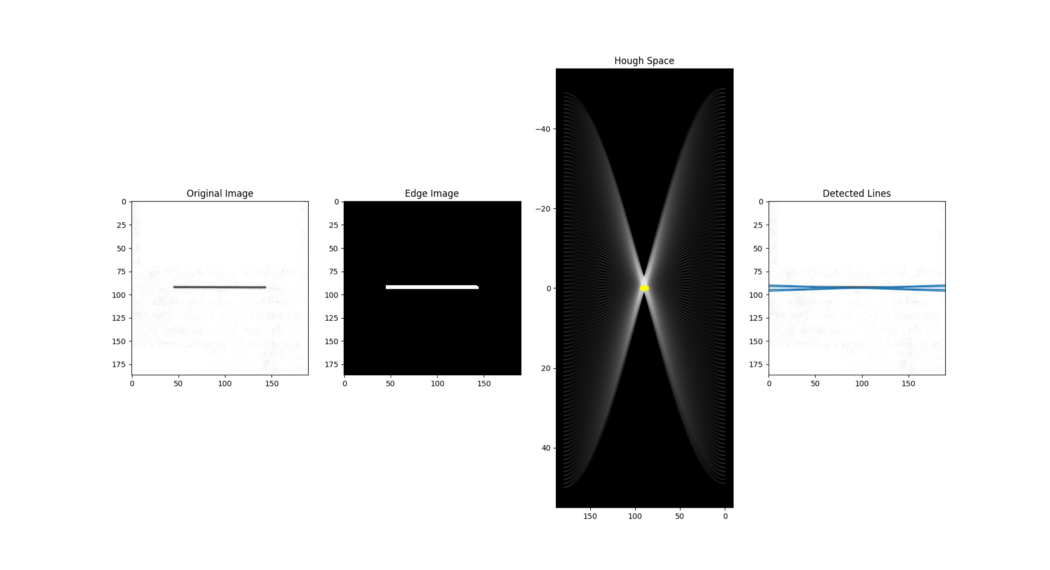




Using this, instead of representing the Hough Space with the slope a and intercept b, it is now represented with ρ and θ where the horizontal axis are for the θ values and the vertical axis are for the ρ values. The mapping of edge points onto the Hough Space works in a similar manner except that an edge point (xᵢ, yᵢ) now generates a cosine curve in the Hough Space instead of a straight line (Leavers, 1992). This normal representation of a line eliminates the issue of unbounded value of a that arises when dealing with vertical lines.

**Line Detection:**

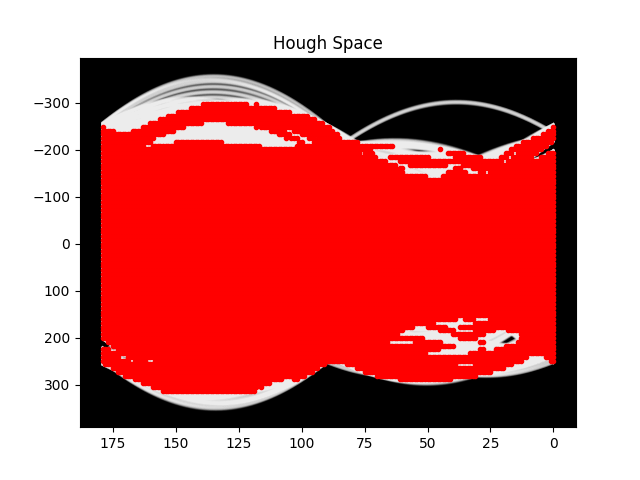
As mentioned, an edge point produces a cosine curve in the Hough Space. From this, if we were to map all the edge points from an edge image onto the Hough Space, it will generate a lot of cosine curves. If two edge points lay on the same line, their corresponding cosine curves will intersect each other on a specific (ρ, θ) pair. Thus, the Hough Transform algorithm detects lines by finding the (ρ, θ) pairs that has a number of intersections larger than a certain threshold. It is worth noting that this method of thresholding might not always yield the best result without doing some preprocessing like neighborhood suppression on the Hough Space to remove similar lines in the edge image.



**Algorithm:**

1. Decide on the range of ρ and θ. Often, the range of θ is [ 0, 180 ] degrees and ρ is [ -d, d ] where d is the length of the edge image’s diagonal. It is important to quantize the range of ρ and θ meaning there should be a finite number of possible values.
2. Create a 2D array called the accumulator representing the Hough Space with dimension (num\_rhos, num\_thetas) and initialize all its values to zero.
3. Perform edge detection on the original image. This can be done with any edge detection algorithm of your choice.
4. For every pixel on the edge image, check whether the pixel is an edge pixel. If it is an edge pixel, loop through all possible values of θ, calculate the corresponding ρ, find the θ and ρ index in the accumulator, and increment the accumulator base on those index pairs.
5. Loop through all the values in the accumulator. If the value is larger than a certain threshold, get the ρ and θ index, get the value of ρ and θ from the index pair which can then be converted back to the form of y = ax + b.

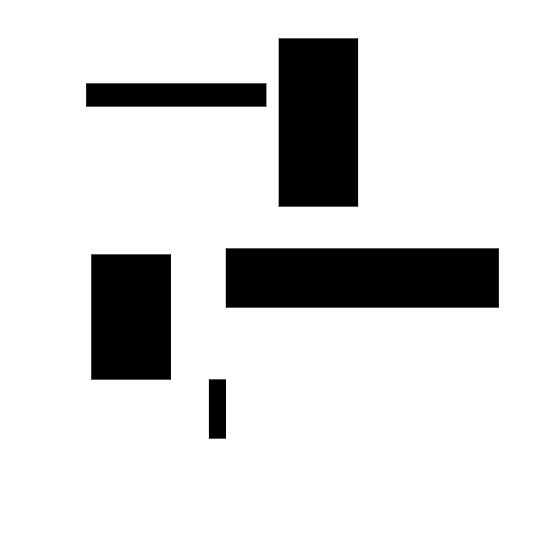
We wrote a python function to convert the edge image into hough space. When we applied the HoughSpace() function on the output image from the Canny Edge Detector, we got the following results,



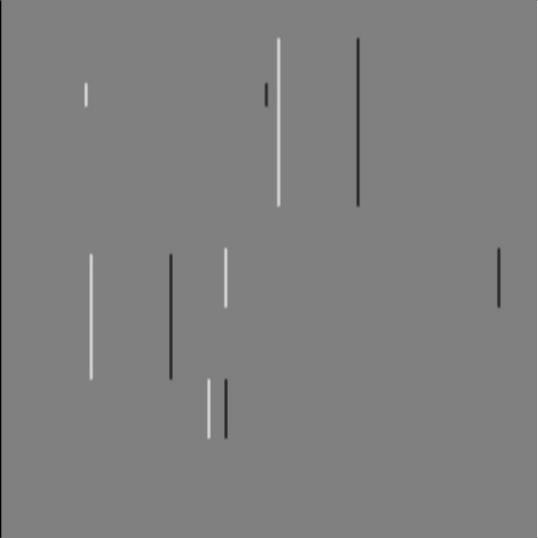
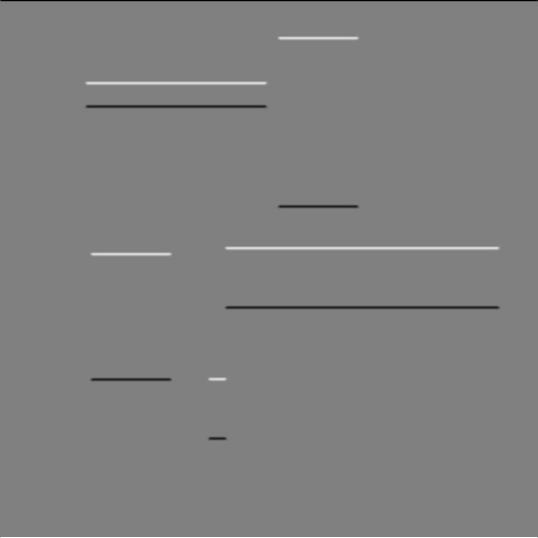
Corresponding values of rho and theta are marked with the red dot and plotted on the same graph.

**Results and Analysis:**

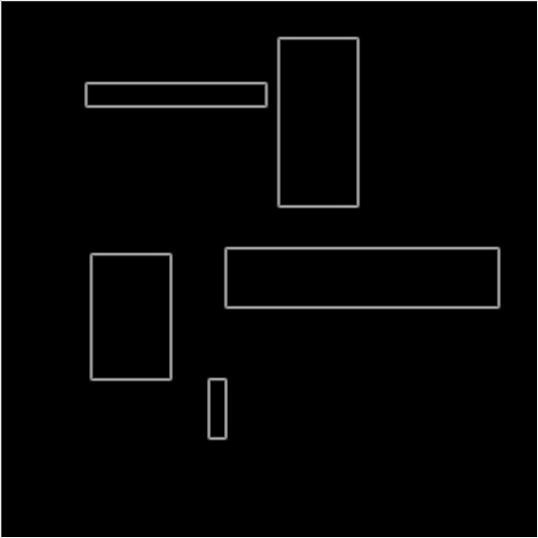
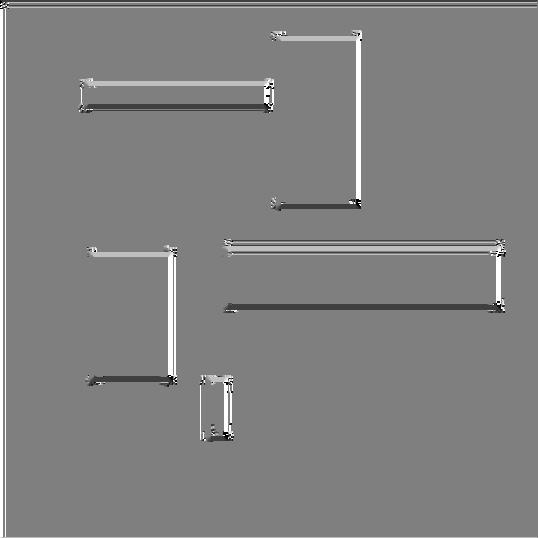
When we applied the edge detection on the provided rectangles’ image, we get the following results:

Original Image Gaussian Blurred Image

Gradient in X direction Gradient in Y Direction

Magnitude Image Angle Image

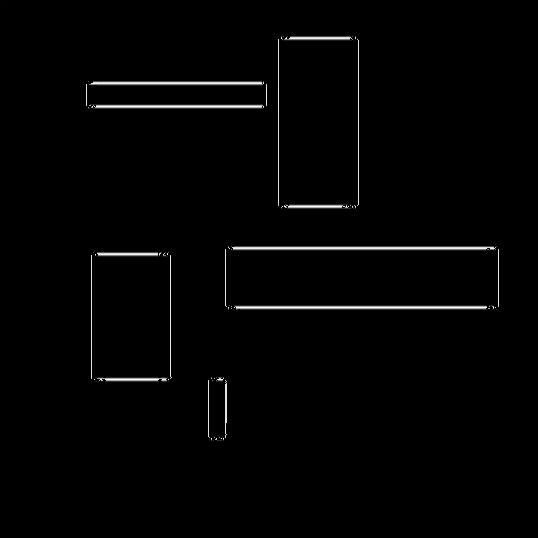
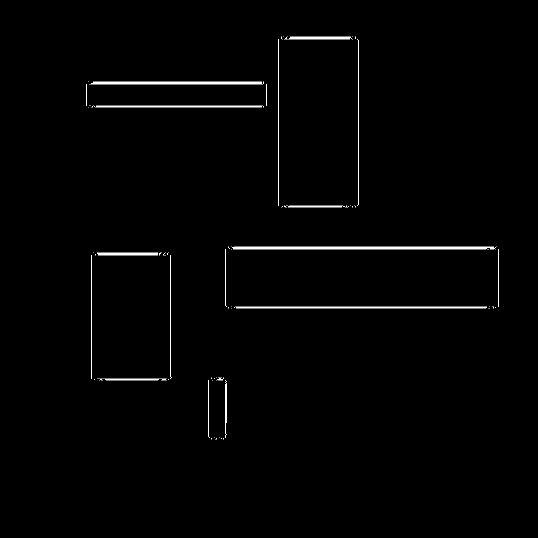
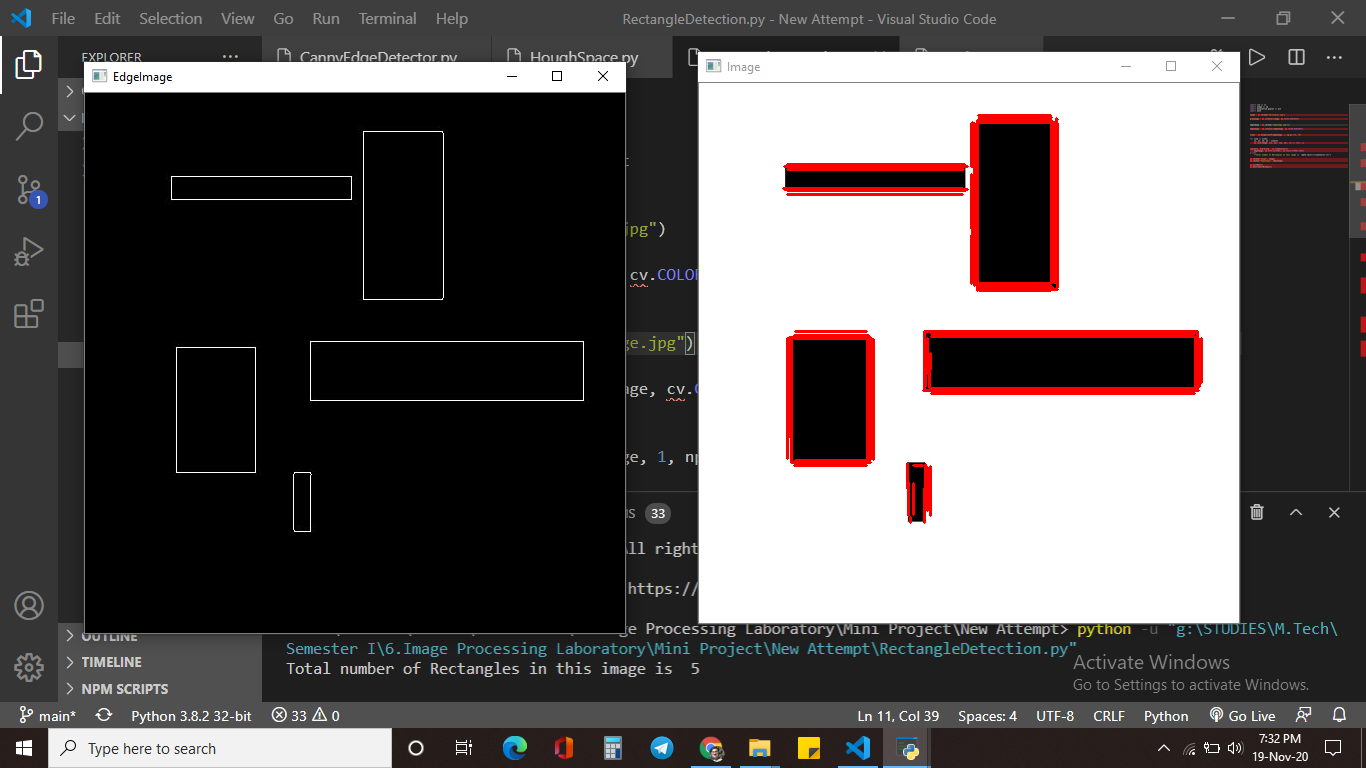
 

Image After Non Maximum Suppression     Final Edge Image

We wrote a python program that detects the number of rectangles in the image using the HoughLinesP() function of opencv. We get the following results from this function,



Detected Rectangles by HoughLinesP()

We used the findContours() function of opencv to detect the number of rectangles.

But maybe because of some discrepancy in the contour, all the edges of the rectangle were detected as the different contours. So we divided the total number of contours by number of edges to get the number of rectangles.

When we applied the inbuilt Canny() function from opencv, the results were perfect and there was no such need of extra calculation.

**Conclusion:**

We have implemented Canny edge detection technique from the scratch and later implemented that on our next task, which was finding the number of rectangles on an image by detecting the lines by Hough transform. It was done with the help of the algorithm used in the Canny’s edge detection technique.

**Reference:**

* <http://dev.theomader.com/gaussian-kernel-calculator/>
* <https://www.researchgate.net/post/What_is_the_algorithm_and_concept_behind_finding_contour_in_openCV>
* <https://nabinsharma.wordpress.com/2012/12/26/linear-hough-transform-using-python/>
* <https://www.geeksforgeeks.org/matplotlib-axes-axes-invert_yaxis-in-python/#:~:text=The%20Axes.,to%20invert%20the%20y%2Daxis>
* <https://stackoverflow.com/questions/56414282/detecting-lines-of-a-rectangle-image-using-hough-transform>