Assignment 1
Canny Edge and Harris Corner Detector

Ujjwal Yadav
2018CSB1127

Abstract

This assignment deals with implementing a Canny Edge Detector and Harris Corner Detector from scratch without using the library functions .We are given some images as Data . We find the edges in the images using the Canny Edge Detector Algorithm and find the corners in the image using the Harris Corner Detector Algorithm . These Algorithms include a number of steps which we describe in this report along with the results obtained.

1 Introduction

We see multiple images in our day to day life and can relate them with the objects in our surroundings . We can perceive the features like color , intensity , features of the image by our eyes and mind . We can also detect Edges and Corners in the image easily . But how do Computers detect images and the features in the image. They are given an image as input in form of a matrix of pixels with varying intensities . They perform mathematical manipulations on the image to extract a number of features from the image . Edges in the images can be thought of as a boundary which separates regions with distinguishing features . The direction of edges is perpendicular to the direction of maximum gradient in the image. Corners are points in the image where generally two edges meet . They exist at the intersection of two or more directions of abrupt intensity change in the image . In the part one of the assignment ,we implement a Canny Edge Detector algorithm which uses the similar principles to detect edges of various strengths in the image .In the second part we implement Harris Edge Detector which detects the corners of the image.

2 Procedure Canny Edge Detector

The Canny Edge Detection Algorithm consists of the following steps:
1) Loading an Image and Extracting the Luminance of the Image.
2) Smoothing with a Gaussian Filter and convolution with suitable filters to calculate the gradient of the image along the x-direction and y-direction.
3) Computing the Gradient Magnitude and Gradient Direction with the help of Gradients along the X-direction and Y-direction . The Gradient Magnitude at each pixel is equal to the square root of the sum of the squares of the X-direction and Y-direction Gradients.The Gradient Direction at every pixel is found using the formula direction= tan(Y-direction gradient/X-direction gradient).

4) Now we thin the edge of the image .This is done by approximating the direction gradient of the image pixel with one of the four possible directions .
5) After approximating the gradient direction of the image pixel ,we see if the gradient magnitude is larger than the pixels in its neighbourhood in that direction.If the gradient magnitude is larger than the gradient magnitude of the neighbours which are oriented in that particular direction , then the gradient magnitude of the thinned image at that pixel is kept as it is else it is set to zero .
6) Now we perform Hysteresis Threshold on the image pixels .This is done by first setting the values of upper threshold and lower threshold.
7) If the edge strength is greater than upper threshold , then that Pixel is marked as Strong Edge. If the edge strength is between the Upper threshold and the lower threshold , then it is marked as Weak Edge ,else it is marked as No Edge.
8) Now from every pixel marked as Strong Edge , we find the the longest connected chain of pixels which are either marked as Strong edge or Weak Edge pixels and mark all the Weak Edge pixels in that connected chain as Strong Edge Pixels , ie we include them in the final edges of the image .This is done using the Depth First Search Algorithm.
9) Finally all the pixels which are marked as Strong Edge Pixels are included in the final edges of the image.

-> This task is completed using the following functions whose algorithms are described below:

2.1 Convert to Luminence Image and Applying Gaussian Filter to reduce Noise

Here we extract the Luminence Image from the RGB Image which reduces the image matrix from 3 dimensions to 1-dimension and does not affect the edge detection that much . This makes our work easier as luminence is a linear combination of the intensities along the red channel,green channel and blue channel. This is followed by Smoothing the Luminenece Image with the help of Gaussian Filter to reduce the subtle noise and make image smooth.

Algorithm 1 Convert to Luminence Image and Applying Gaussian Filter to reduce Noise

1: $red\_channel \leftarrow image[:,:,0]$
2: $green\_channel \leftarrow image[:,:,1]$
3: $blue\_channel \leftarrow image[:,:,2]$
4: $lum\_image \leftarrow (0.2989 * red\_Channel) + (0.5870 * green\_Channel) + (0.1140 * blue\_Channel)$
5: $lum\_image \leftarrow GaussianBlur(lum\_image, shape = (5,5), sigma = 1))$
6: **return** lum_image

2.2 Calculate the Gradient Magnitude and Direction

Here we use the filter similar to the derivative of the Gaussian Filter in order to find the gradient along the x-direction and y-direction.
The direction of maximum change in pixel intensity can be calculated by using the x-gradient and y-gradient .

We get the gradient magnitude and direction which helps us in further determining the covariance matrix around each pixel.
the algorithm is discussed:

Algorithm 2 $\text{get}_g radient(image)$

$x\_dir \leftarrow np.asarray[[1,0,-1],[1,0,-1],[1,0,-1]]$
$y\_dir \leftarrow np.asarray[[1,1,1],[0,0,0],[-1,-1,-1]]$
$gradient\_x \leftarrow ndimage.convolve(image, x\_dir)$
$gradient\_y \leftarrow ndimage.convolve(image, y\_dir)$
$magnitude\_gradient \leftarrow \sqrt{gradient\_x^2 + gradient\_y^2}$
$direction\_gradient \leftarrow tan^{-1}(gradient\_y / gradient\_x)$
return direction_gradient, magnitude_gradient

2.3 Non Maximal Suppression

Here we find the approximate direction of each of the gradient direction and check if the neighbourhood pixels in that direction have higher edge strength than the current pixel or not.
This helps us to get thin edges with width of one pixel.

Algorithm 3 suppression(gradient_image)

```
direction_gradient ← gradient_image[0]
magnitude_gradient ← gradient_image[1]
h ← image_height
w ← image_width
for i ← 0 : h do
    for j ← 0 : w do
        if (direction_gradient[i][j] ≤ 0) then
            direction_gradient[i][j] += π
            for i ← 1 : h − 1 do
                for j ← 1 : w − 1 do
                    if (0 ≤ direction_gradient[i][j] < π/8) or (7π/8 ≤ direction_gradient[i][j] ≤ π) then
                        nb1 ← direction_gradient[i][j + 1]
                        nb2 ← direction_gradient[i][j − 1]
                    else if (π/8 ≤ direction_gradient[i][j] < 3π/8) then
                        nb1 ← direction_gradient[i + 1][j − 1]
                        nb2 ← direction_gradient[i − 1][j + 1]
                    else if (3π/8 ≤ direction_gradient[i][j] < 5π/8) then
                        nb1 ← direction_gradient[i + 1][j]
                        nb2 ← direction_gradient[i − 1][j]
                    else if (5π/8 ≤ direction_gradient[i][j] < 7π/8) then
                        nb1 ← direction_gradient[i − 1][j − 1]
                        nb2 ← direction_gradient[i + 1][j + 1]
                    end if
                    if direction_gradient[i][j] ≥ nb1 and direction_gradient[i][j] ≥ nb2 then
                        suppressed_image ← magnitude_gradient[i][j]
                    else
                        suppressed_image ← 0.00
                    end if
                end for
            end for
            returnsuppressed_image
```

2.4 Threshold

-> Here we classify the image pixels as Strong edge or Weak edge or No edge depending on the threshold values .
-> This gives us strong edge matrix,weak edge matrix and no edge matrix.

Algorithm 4 get_threshold_matrices(sup_image,low_v,high_v)

```
threshold_matrix ← zeros.shape_like(sup_image) array
no_edge_matrix ← zeros.shape_like(sup_image) array
strong_edge_matrix ← zeros.shape_like(sup_image) array
weak_edge_matrix ← zeros.shape_like(sup_image) array
h ← image_height
w ← image_width
for i ← 0 : h do
    for j ← 0 : w do
        if (sup_image[i][j] ≥ high_v) then
            threshold_matrix[i][j] ← 1.0
            strong_edge_matrix[i][j] ← 1.0
        else if (sup_image[i][j] ≥ low_v) and (sup_image[i][j] ≤ high_v) then
            threshold_matrix[i][j] ← 0.5
            weak_edge_matrix[i][j] ← 1.0
        else if (sup_image[i][j] ≤ low_v) then
            no_edge_matrix[i][j] ← 1.0
        end if
    end for
end for
return threshold_matrix,no_edge_matrix,strong_edge_matrix,weak_edge_matrix =0
```

2.5 Longest Chain of Edge Pixels

->Here we apply the Depth First Search starting from the Strong edge pixel and classify all the pixels with weak edges as strong if they are in neighbourhood of a strong edge pixel.
->This is done recursively as a weak pixel in connection with a strong pixel becomes strong itself and the chain flows like this.

Algorithm 5 depth_for_search_sec(image)

```
visited ← zeros.shape_like(image) array
h ← image.shape[0]
w ← image.shape[1]
for i ← 0 : h do
    for j ← 0 : w do
        if (image[i][j] == 1) and (visited[i][j] == 0) then
            image, visited = depth_for_search_prim(image, visited, i, j)
        end if
    end for
end for
for i ← 0 : h do
    for j ← 0 : w do
        if (image[i][j]! = 1) then
            image[i][j] = 0
        end if
    end for
end for
return image
```

Algorithm 6 depth_for_search_prim(image,visited,i,j)

```
h ← image.shape[0]
w ← image.shape[1]
if i < 0 or j < 0 or i ≥ h − 1 or j ≥ w − 1 or visited[i][j] == 1 then
    return image,visited
end if
visited[i][j] = 1
if image[i][j] == 0 then
    return image,visited
end if
image[i][j] = 1
count = 8
image, visited ← depth_for_search_prim(image, visited, i + 1, j + 1)
image, visited ← depth_for_search_prim(image, visited, i + 1, j)
image, visited ← depth_for_search_prim(image, visited, i + 1, j − 1)
image, visited ← depth_for_search_prim(image, visited, i, j + 1)
image, visited ← depth_for_search_prim(image, visited, i, j)
image, visited ← depth_for_search_prim(image, visited, i, j − 1)
image, visited ← depth_for_search_prim(image, visited, i − 1, j + 1)
image, visited ← depth_for_search_prim(image, visited, i − 1, j)
image, visited ← depth_for_search_prim(image, visited, i − 1, j − 1)
return visited, image,visited
```

3 Harris Corner Detector

3.1 Get Luminence Image

-> Here we load the image and get the luminence image from the RGB Image . By getting the Luminence Image we are working with an image matrix which is not 3 dimensional . So it reduces our efforts and also the luminence image does not modify the results of the Harris Corner Detection. ->The algorithm is described below:

Algorithm 7 Get Luminence Image

1: $red\_channel \leftarrow image[:,:,0]$
2: $green\_channel \leftarrow image[:,:,1]$
3: $blue\_channel \leftarrow image[:,:,2]$
4: $lum\_image \leftarrow (0.2989 * red\_Channel) + (0.5870 * green\_Channel) + (0.1140 * blue\_Channel)$
5: **return** lum_image

3.2 Calculate Gradient in X and Y directions

-> This is similar to the canny edge gradient calculator , where we use two filters to calculate the gradients along the x and y directions.
->The gradients helps us in determining the change in pixel intensity along the x and y direction.

Algorithm 8 get_gradient(image)

1: $x\_dir \leftarrow np.asarray[[1,0,-1],[1,0,-1],[1,0,-1]]$
2: $y\_dir \leftarrow np.asarray[[1,1,1],[0,0,0],[-1,-1,-1]]$
3: $gradient\_x \leftarrow ndimage.convolve(image, x\_dir)$
4: $gradient\_y \leftarrow ndimage.convolve(image, y\_dir)$
5: **return** gradient_x, gradient_y

3.3 Get Corner Response

-> Here we find the corner response of Harris Corner Detector for each pixel by evaluating a Covariance Matrix around each pixel by taking a window of size 9*9.
->First we calculate the square of gradients along the x direction and along the y-direction.Also we calculate gradient_xy which is equal to the product of gradient along x with gradient along y.
->For every pixel ,we find the 9*9 matrix by placing the pixel in the centre for each of gradient_x , gradient_y and gradient_y. We sum the each of these matrix and calculate the Covariance Matrix in this way.

Algorithm 9 get_corner_response(image, gradient_x, gradient_y)

$k \leftarrow 0.04$
$m \leftarrow 4$
$h \leftarrow image.shape[0]$
$w \leftarrow image.shape[1]$
Initialize $gradient\_x\_2 = gradient\_x^2, gradient\_y\_2 = gradient\_y^2, gradient\_xy = gradient\_x * gradient\_y$
$height, width = img.shape$ and Initialize zero matrix cov_matrix(h, w)
for $i \leftarrow m : h-m$ **do**
    for $j \leftarrow m, w-m$ **do**
        $sum\_x\_2 \leftarrow gradient\_x\_2((i-m : i+m+1, j-m : j+m+1)).sum()$
        $sum\_xy \leftarrow (gradient\_xy(i-m : i+m+1, j-m : j+m+1)).sum()$
        $sum\_y\_2 \leftarrow (gradient\_y\_2(i-m : i+m+1, j-m : j+m+1)).sum()$
        $e\_value \leftarrow (((sum\_x\_2 * sum\_y\_2) - (sum\_xy * *2)) - (k * ((sum\_x\_2 + sum\_y\_2) * *2)))$
        $cov\_matrix(i, j) = e\_value$
    end for
end for
return cov_matrix

3.4 Get Threshold Corners

-> We select all those corners whose corner responses are greater than a given threshold.
->The threshold values prove to be important in determining the accuracy and the number of corners detected.

Algorithm 10 get_corner_threshold(image,cov_matrix,t_ratio)

$h \leftarrow image.shape[0]$
$w \leftarrow image.shape[1]$
$Initialize list\_corners as empty list$
$t\_value \leftarrow t\_ratio * cov\_matrix.max$
for each pixel (i,j) in image **do**
    if $cov\_matrix(i, j) > t\_value$ **then**
        $list\_corners.append(i, j, cov\_matrix(i, j))$
    end if
end for
returnlist_corners

3.5 Non Maximal Suppression

->Here we first sort all the corners obtained after thresholding according to the corner response values .
->Starting from the corner with the highest corner response , we find all the corners in the corners list which are in the neighbourhood of the current corner.
->Any such neighbouring corners are eliminated as they have lower corner responses as com-

pared to the present corner and in this way the Maximal Suppression Takes place.

Algorithm 11 sup_corners(list_corners)

```
indces_sorted ← argsort((list_corners)[:,2])
Initailize an empty list as list_final
length ← len(indces_sorted)
for i ← 0 : length do
    if list_corners(indces_sorted(i))(0)! = −1 then
        x_cord ← list_corner(indces_sorted(i))(0)
        y_cord ← list_corner(indces_sorted(i))(1)
        list_final.append(cornerList(i, j, list_corner(indces_sorted(i))(2))
        list_corner(indces_sorted(i))(1) ← −1
        list_corner(indces_sorted(i))(0) ← −1
        for every k which is neighbour pixel of (x_cord,y_cord) and is present in
    list_corners do
            list_corners(k)(1) ← −1
            list_corners(k)(0) ← −1
        end for
    end if
end for
return list_final
```

4 Observations and Takeaways

->We make the following observations with respect to Canny Edge Detector and Harris Corner Detection:

1) As the number of value of lower threshold increases , we get less and less visible edges as most of the pixels will have the edge strength less than the given lower threshold s, so most of them will be classified as No_edge_pixels.

2) As the value of lower threshold decreases the more and more edge pixels will be lying in the bound between the lower threshold and upper threshold. So more and more edges will be detected.

3) As the value of upper threshold increases more and more pixels have edge strength less than it , so very less pixels will be classified as Strong edge detected ,thus reducing the edges .

4) As the value of upper threshold decreases , more and more pixels will be classified as Strong edge pixels resulting in more edges detection.

5) Thinned edge image makes the width of the edges small due to the fact that we compare the pixels with their neighbourhood pixels and assign them as edge pixels only and only if they have gradient magnitude in their approximated gradient direction greater than all neighborhood pixels .

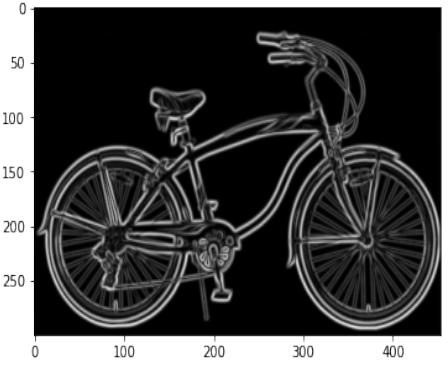
6) We have considered all the eight pixels surrounding a pixel in its neighbourhood , but if we consider only 4 or 6 pixel neighbourhood then the results will not be as good as we see in case of 8 neighbourhood .

7) The gradient direction shows us the direction of maximum change in pixel intensity.

8) The edges are in direction perpendicular to the gradient direction as edges are always perpendicular to the direction of change in maximum pixel intensity.
9) The Hysteresis Thresholding helps us to extract the features and almost all major edges of the images . This is because in Hysteresis Thresholding we start from a pixel marked with strong edge intensity onto the nearby pixels until we find a no edge pixel. In this way we continue to draw and extend the edge as long as there is significant change in the pixel intensities.
8) Now we mention some of the observations of the Harris Edge Detector.
8) As we have a high value of threshold for a pixel to be marked as corner , a lesser number of corners are detected .
9) As we lower the threshold value for corner detection , more and more number of corners are detected.
10) As we have lower value of threshold for corner detection , then some of the points which have very less chances of being corners are also marked as corner , thus we have to trade off with accuracy in corner prediction as Threshold value decreases.
11) As we have higher values of Threshold the accuracy in marking of corner increases and we have less number of but more accurate corners.
12) As we rotate the figure the Harris Detector has no change in the corners detected as rotation does not change the relative pixel intensities of the image.
13) Also adding or subtracting the intensity from image does not change the corner detection as all the pixels of the image are affected if the intensity of the images is increased or decreased .
14) Harris Corner detector is still used for facial features extraction.
15) Canny Edge detector and Harris Corner detector can be widely used in Human Recognition and Feature Extraction .
16) Many modern technologies use these algorithms for verification, security , testing etc.

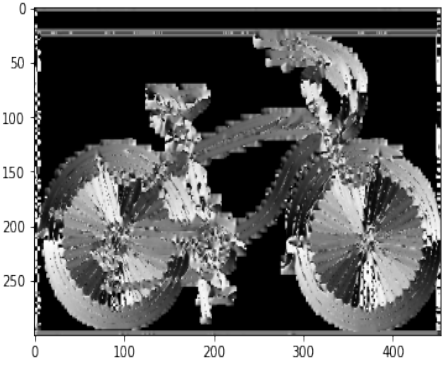
5 Result

->We now give all the images output for the Canny Edge Detector and Harris Edge detector.
-> For canny edge detector , every input image has six output images which are Gradient Magnitude, Gradient Direction, Thinned OR NMS Image ,Hysteresis Thresholded Image,Strong Edge,Weak Edge Image .The lower and upper thresholds are around 0.07 and 0.26 .
->For Harris Edge Detector , every input image has two output images - one the luminence image and the second one is the final corner detected output. The threshold ratio is 0.1 initially.



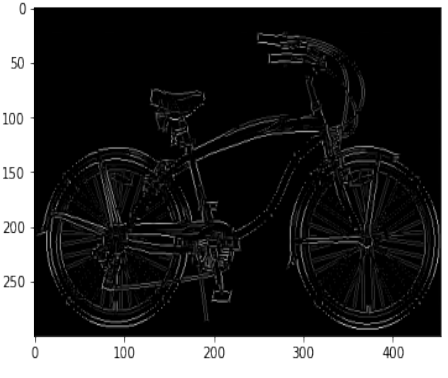
(a)

Figure 1: Gradient Magnitude for Bicycle



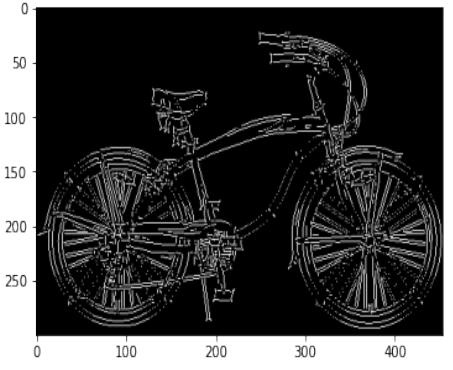
(a)

Figure 2: Gradient Direction for Bicycle



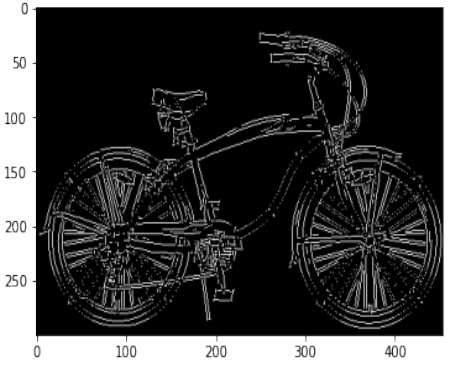
(a)

Figure 3: Thinned Edge Bicycle



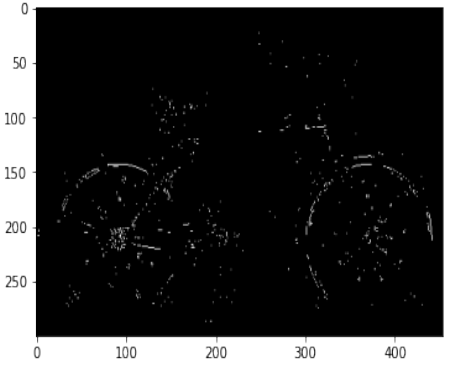
(a)

Figure 4: Hysteresis Thresholded



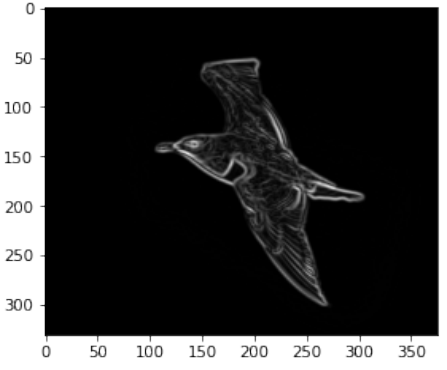
(a)

Figure 5: Strong Edge



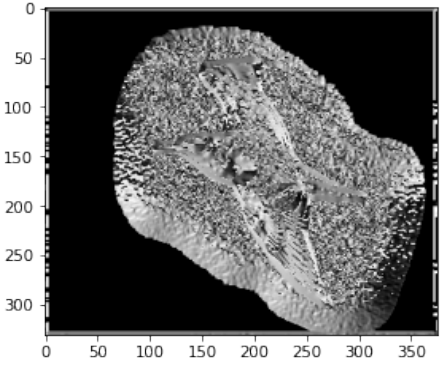
(a)

Figure 6: Weak Edge



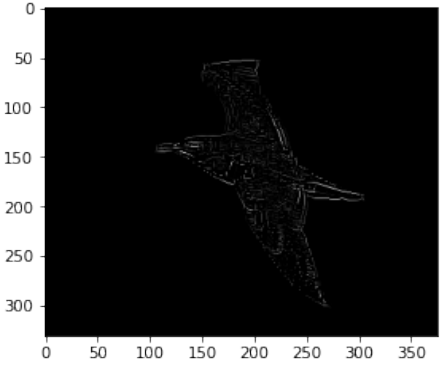
(a)

Figure 7: Gradient Magnitude for Bird



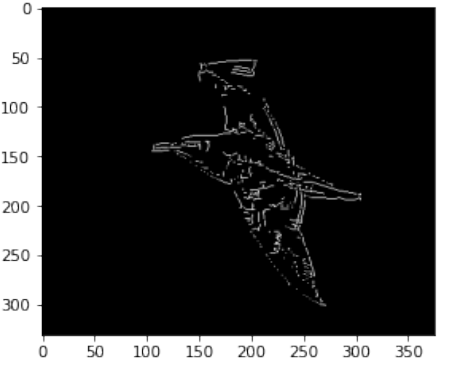
(a)

Figure 8: Gradient Direction for Bird



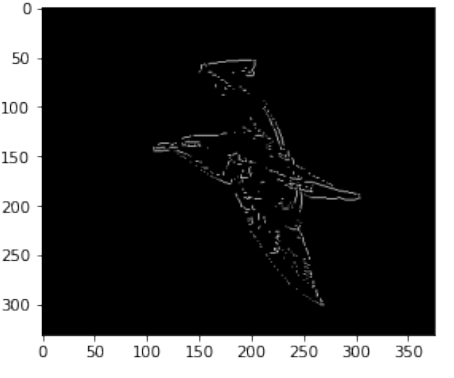
(a)

Figure 9: Thinned Edge Bird



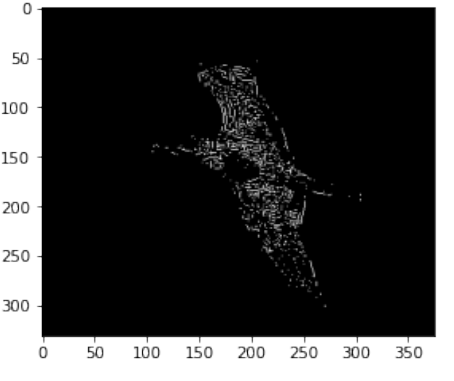
(a)

Figure 10: Hysteresis Thresholded



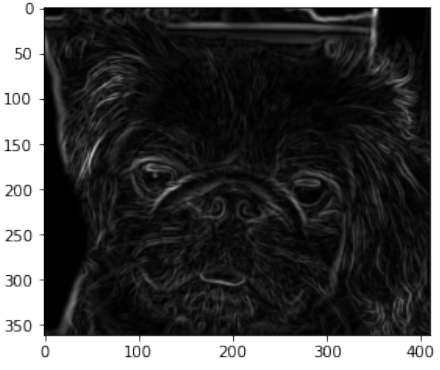
(a)

Figure 11: Strong Edge



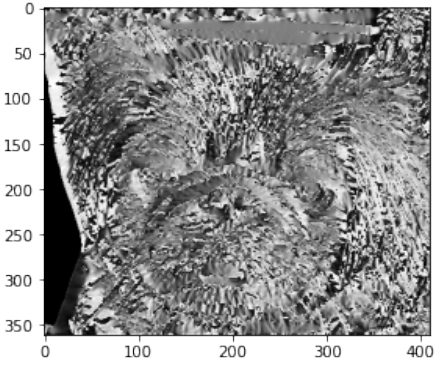
(a)

Figure 12: Weak Edge



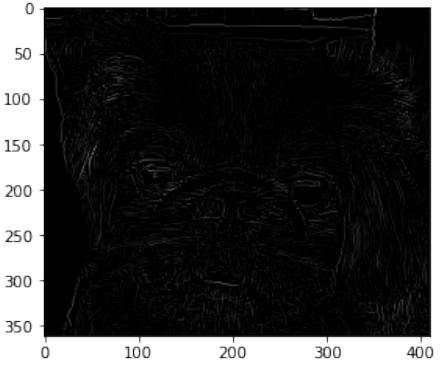
(a)

Figure 13: Gradient Magnitude for Dog



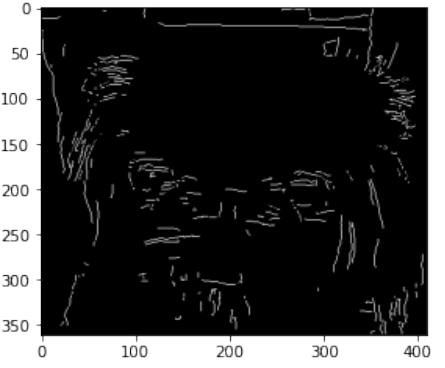
(a)

Figure 14: Gradient Direction for Dog



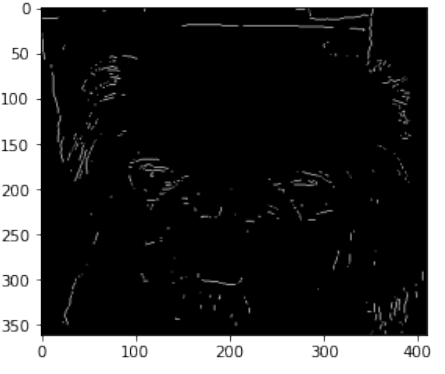
(a)

Figure 15: Thinned Edge Dog



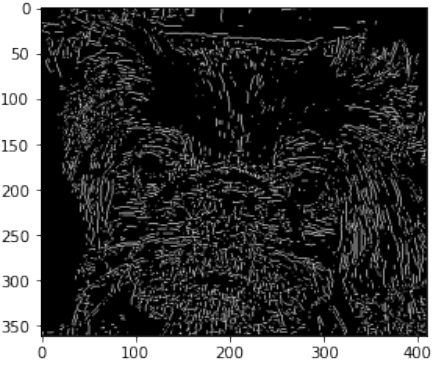
(a)

Figure 16: Hysteresis Thresholded



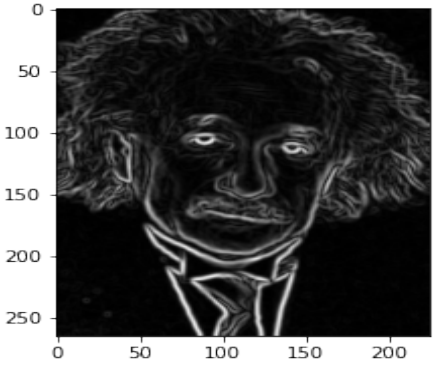
(a)

Figure 17: Strong Edge



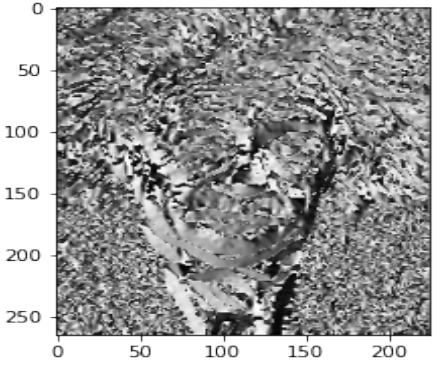
(a)

Figure 18: Weak Edge



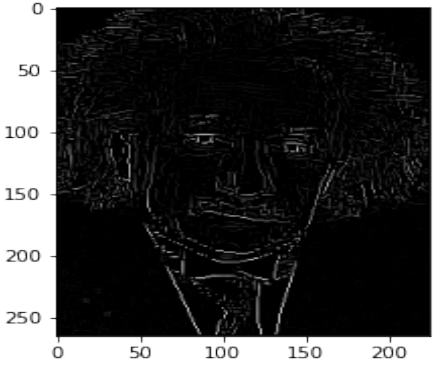
(a)

Figure 19: Gradient Magnitude for Einstein



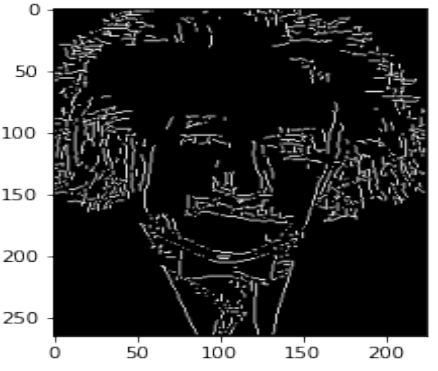
(a)

Figure 20: Gradient Direction for Einstein



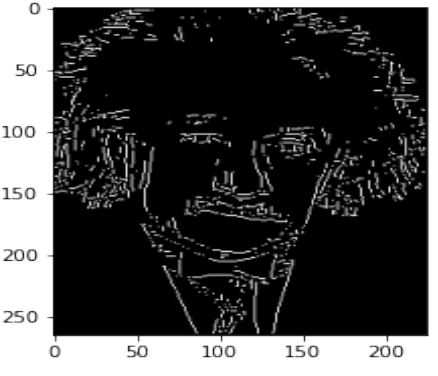
(a)

Figure 21: Thinned Edge Einstein



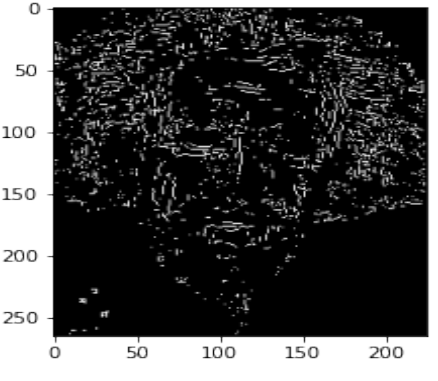
(a)

Figure 22: Hysteresis Thresholded



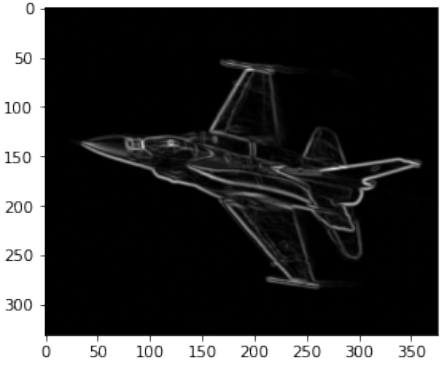
(a)

Figure 23: Strong Edge



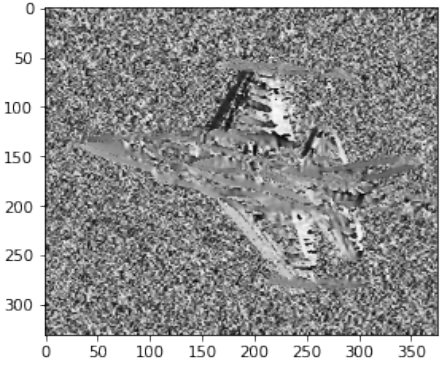
(a)

Figure 24: Weak Edge



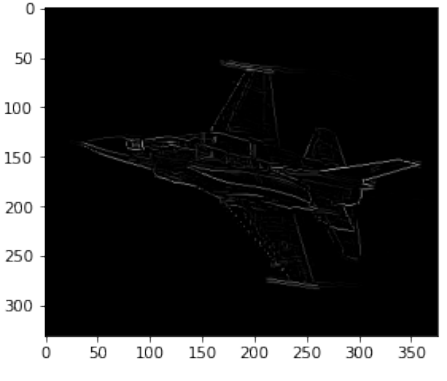
(a)

Figure 25: Gradient Magnitude for Plane



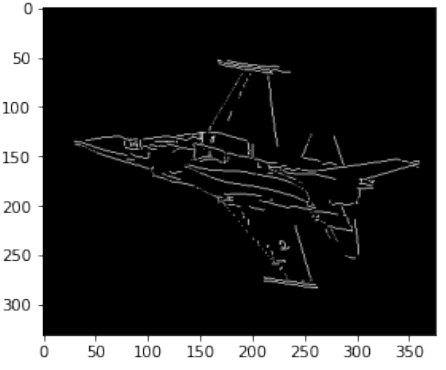
(a)

Figure 26: Gradient Direction for Plane



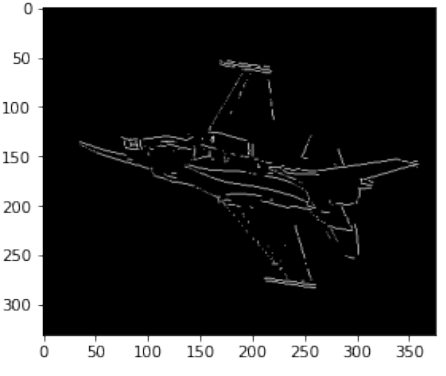
(a)

Figure 27: Thinned Edge Plane



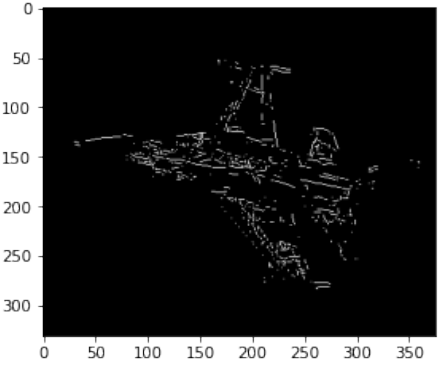
(a)

Figure 28: Hysteresis Thresholded



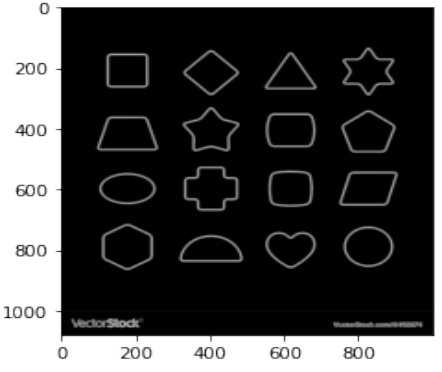
(a)

Figure 29: Strong Edge



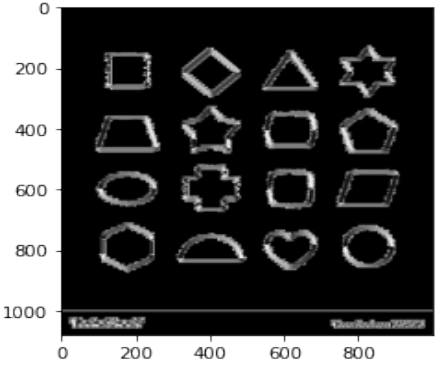
(a)

Figure 30: Weak Edge



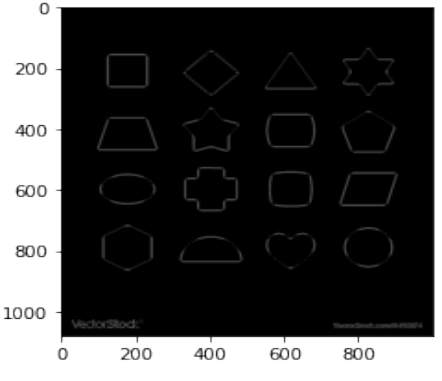
(a)

Figure 31: Gradient Magnitude for Toy Image



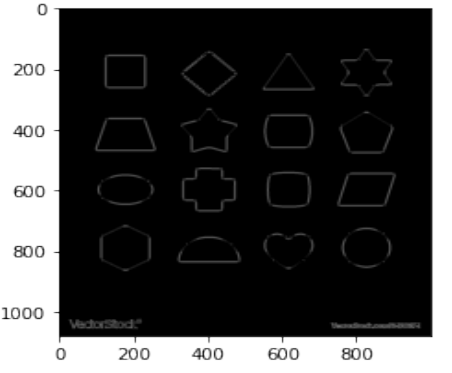
(a)

Figure 32: Gradient Direction for Toy Image



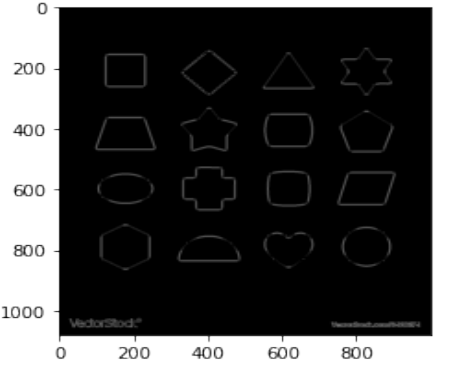
(a)

Figure 33: Thinned Edge Toy Image



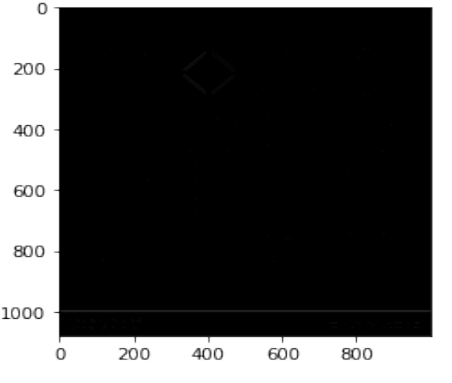
(a)

Figure 34: Hysteresis Thresholded



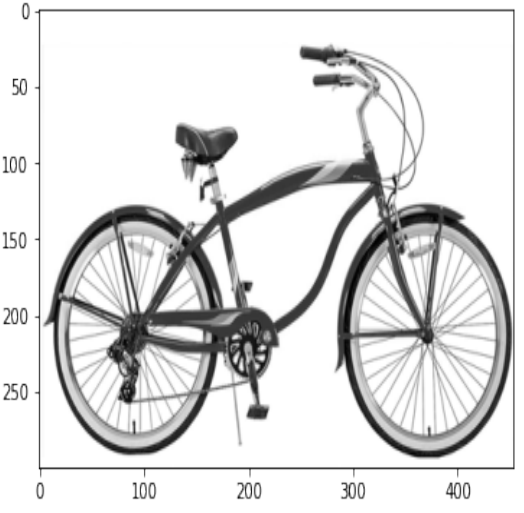
(a)

Figure 35: Strong Edge



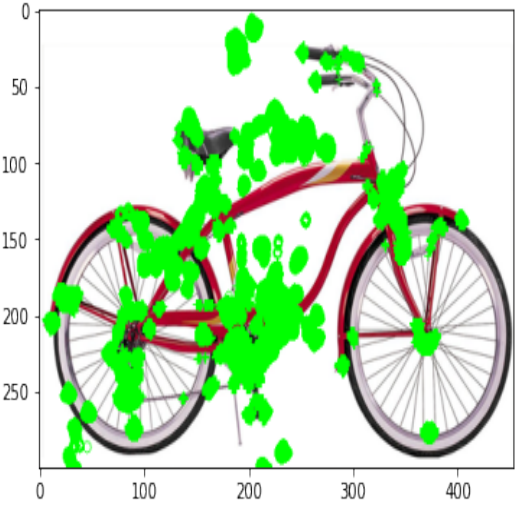
(a)

Figure 36: Weak Edge



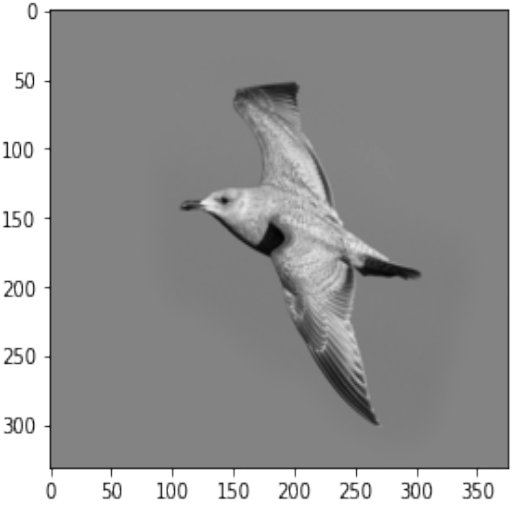
(a)

Figure 37: Luminence Image



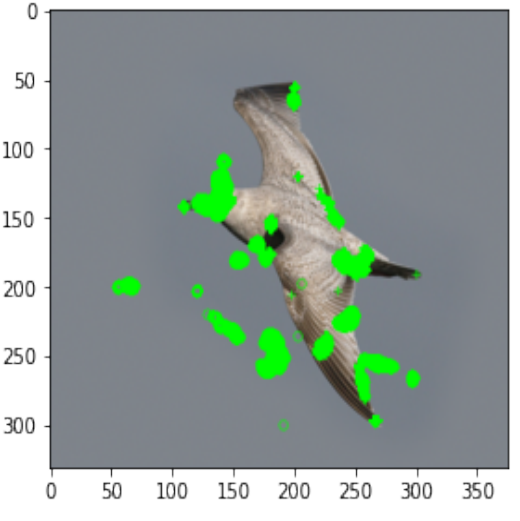
(a)

Figure 38: Image with Corner Detected



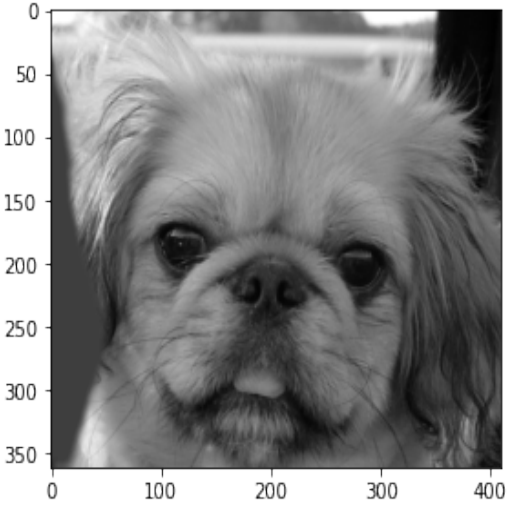
(a)

Figure 39: Luminence Image



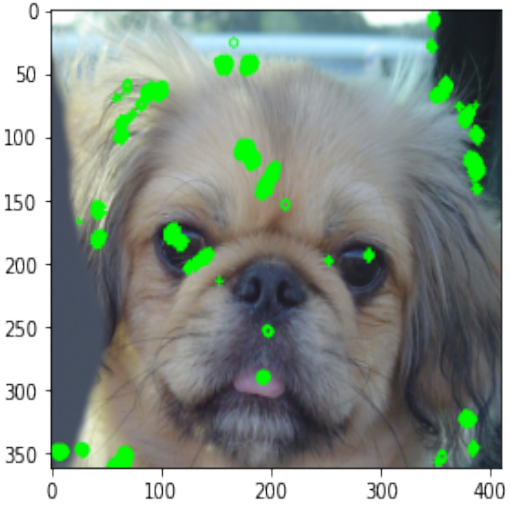
(a)

Figure 40: Image with Corner Detected



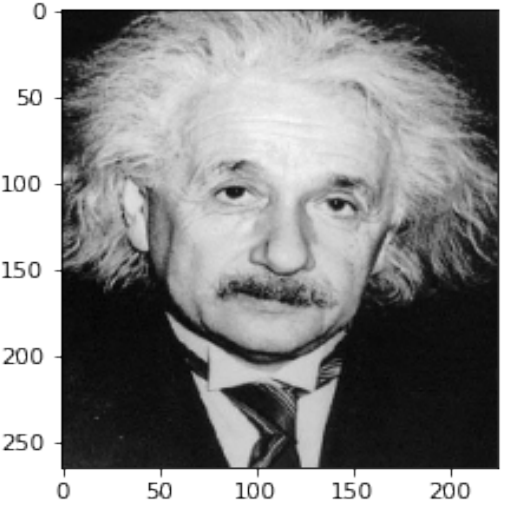
(a)

Figure 41: Luminence Image



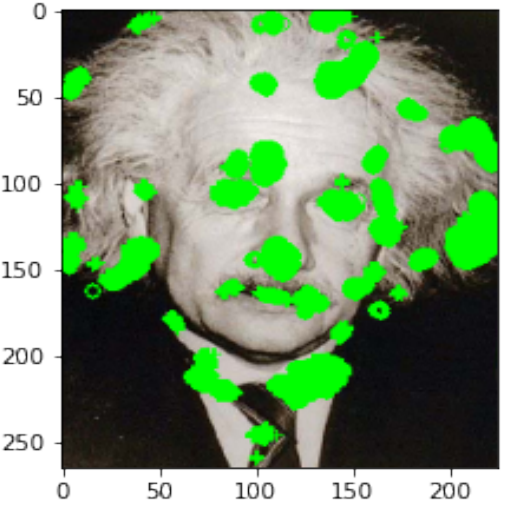
(a)

Figure 42: Image with Corner Detected



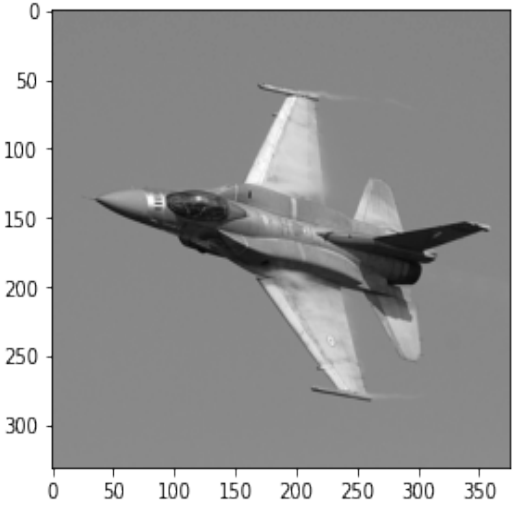
(a)

Figure 43: Luminence Image



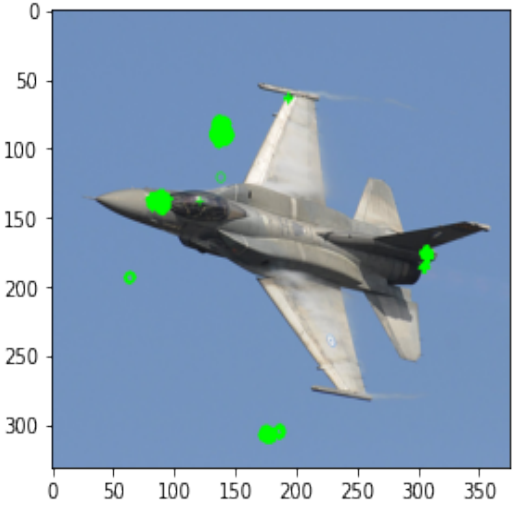
(a)

Figure 44: Image with Corner Detected



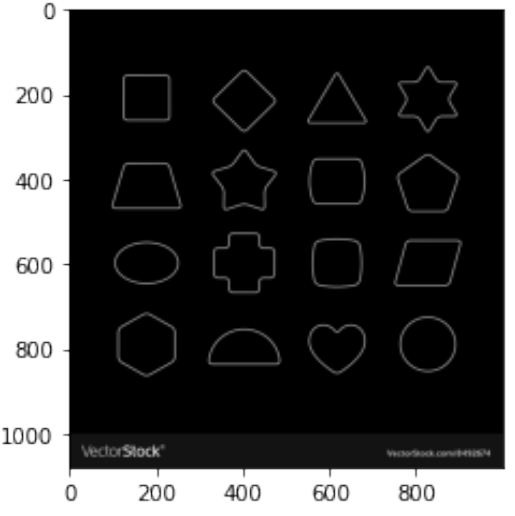
(a)

Figure 45: Luminence Image



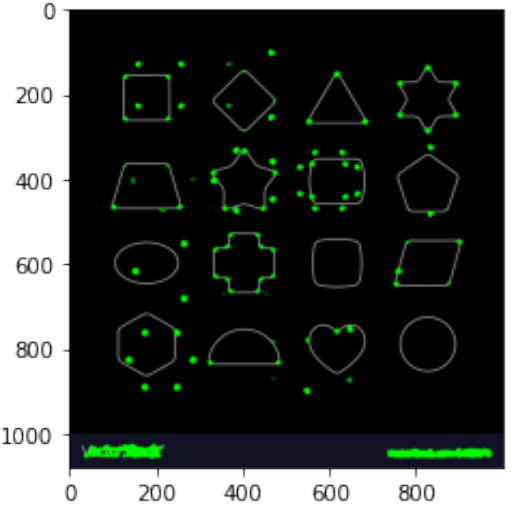
(a)

Figure 46: Image with Corner Detected



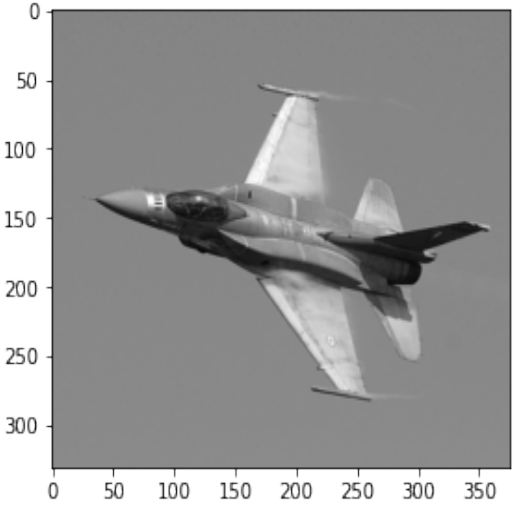
(a)

Figure 47: Luminence Image



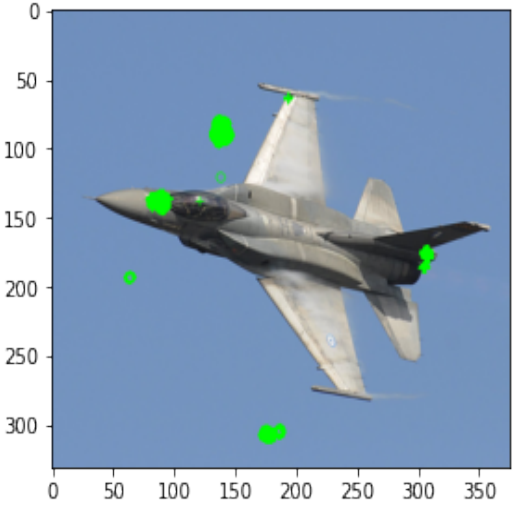
(a)

Figure 48: Image with Corner Detected



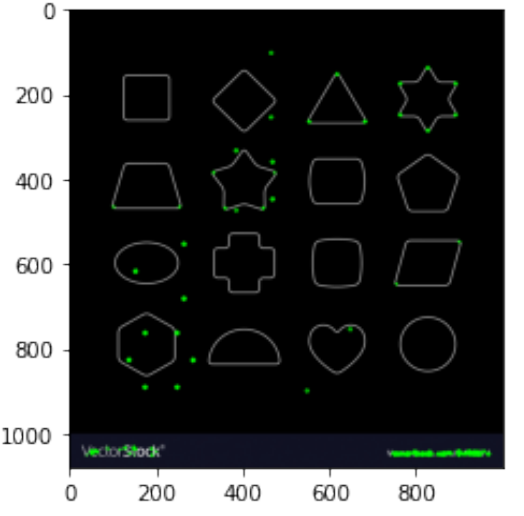
(a)

Figure 49: Luminence Image



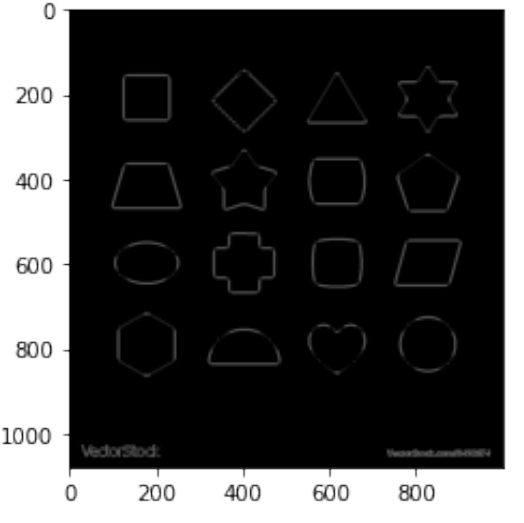
(a)

Figure 50: Image with Corner Detected



(a)

Figure 51: Image with higher value of Threshold for Corner Response thus lesser corners



(a)

Figure 52: Image with increased lower and upper thresholds for Canny Edge Detector thus lighter edges