Task 1

Fig 1.1: Input Image

The steps of Canny Edge detection is shown below.

1. **Grayscale Image:**
   Firstly we have to convert the image into a grayscale image to apply edge detection.

Fig 1.2: Grayscale Image

2. **Gaussian Smoothing (Noise Reduction):**

Fig 1.3: Image after applying Gaussian Blurring.

3. **Gradient Calculation:**

Gradient are calculated in this step.

g_x =

```
gx =  [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

g_y =

```
gy =  [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

Then we calculated the magnitude and angle using g_x and g_y .

```
Magnitude =  [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
Angle =  [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

4. Non-maximum Suppression:

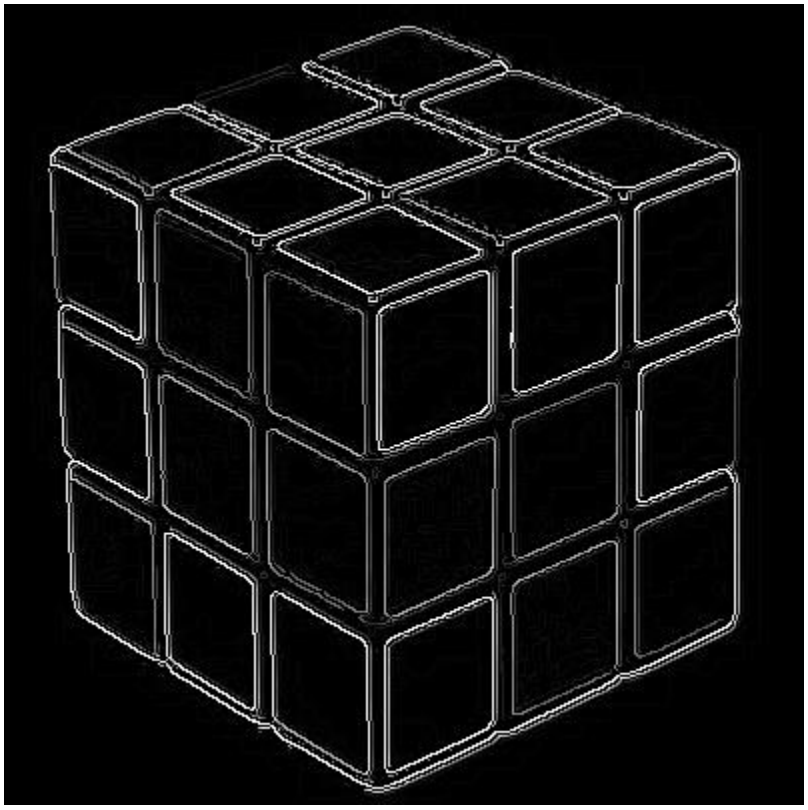

Fig 1.4: Image after applying Non-maximum Suppression

5. Double Thresholding:

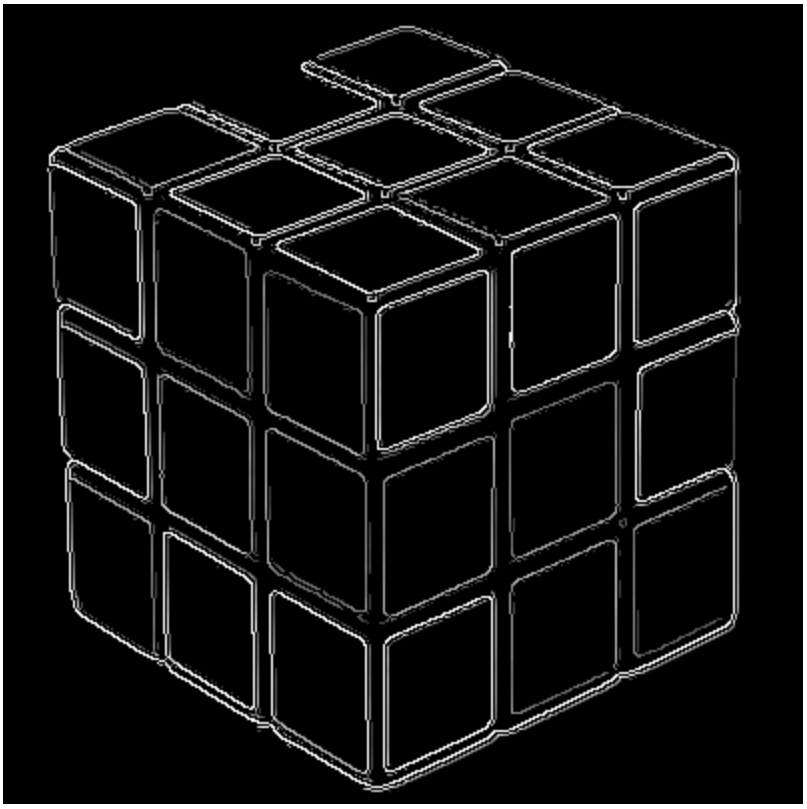

Fig 1.5: Image after applying Double Thresholding.

6. **Edge Tracking (Hysteresis):**

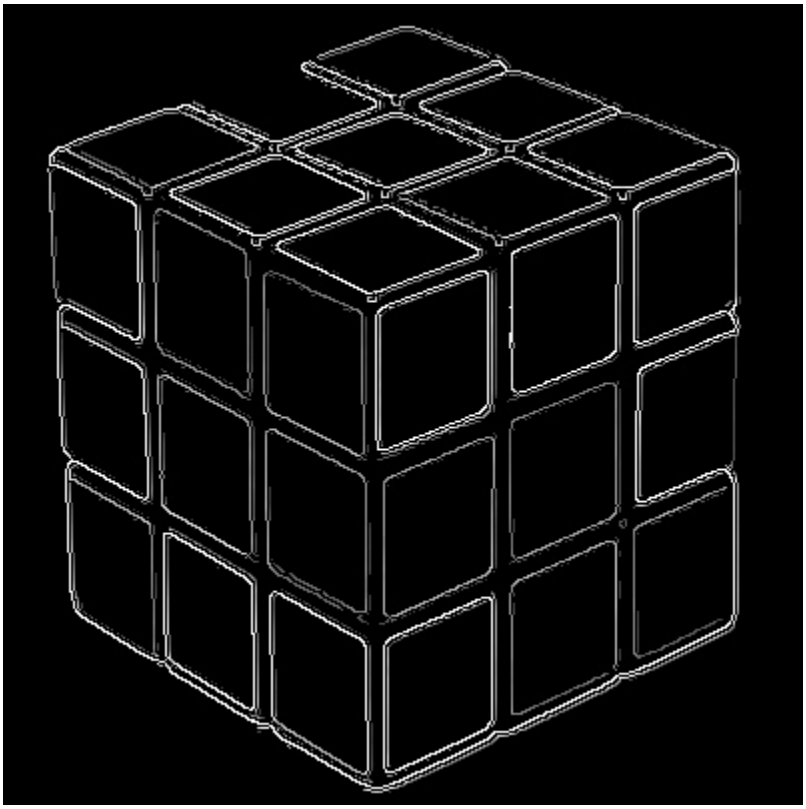

Fig 1.6: Output Image after Canny Detection

Code:

```
import numpy as np
import os
import cv2

def Canny_detector(img, weak_th=None, strong_th=None):
    # conversion of image to grayscale
    img = cv2.resize(img, (400, 400), interpolation = cv2.INTER_AREA)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    cv2.imshow('Grayscale', img)
    cv2.imwrite("Grayscale.jpg", img)

    # Noise reduction step
    img = cv2.GaussianBlur(img, (5, 5), 1)
    cv2.imshow('Noise Reduction', img)
    cv2.imwrite("Noise Reduction.jpg", img)

    # Calculating the gradients
    gx = cv2.Sobel(np.float32(img), cv2.CV_64F, 1, 0, 3)
    gy = cv2.Sobel(np.float32(img), cv2.CV_64F, 0, 1, 3)
    print("gx = ", gx)
    print("gy = ", gy)
```

```
# Conversion of Cartesian coordinates to polar
mag, ang = cv2.cartToPolar(gx, gy, angleInDegrees=True)
print("Magnitude = ", mag)
print("Angle = ", ang)

mag_max = np.max(mag)
print("Maximum magnitude = ", mag_max)

if not weak_th: weak_th = mag_max * 0.1
if not strong_th: strong_th = mag_max * 0.7
height, width = img.shape

for i_x in range(width):
    for i_y in range(height):

        grad_ang = ang[i_y, i_x]
        grad_ang = abs(grad_ang - 180) if abs(grad_ang) > 180 else abs(grad_ang)

        if grad_ang <= 22.5:
            neighb_1_x, neighb_1_y = i_x - 1, i_y
            neighb_2_x, neighb_2_y = i_x + 1, i_y

        # top right (diagnol-1) direction
        elif grad_ang > 22.5 and grad_ang <= (22.5 + 45):
            neighb_1_x, neighb_1_y = i_x - 1, i_y - 1
            neighb_2_x, neighb_2_y = i_x + 1, i_y + 1

        # In y-axis direction
        elif grad_ang > (22.5 + 45) and grad_ang <= (22.5 + 90):
            neighb_1_x, neighb_1_y = i_x, i_y - 1
            neighb_2_x, neighb_2_y = i_x, i_y + 1

        # top left (diagnol-2) direction
        elif grad_ang > (22.5 + 90) and grad_ang <= (22.5 + 135):
            neighb_1_x, neighb_1_y = i_x - 1, i_y + 1
            neighb_2_x, neighb_2_y = i_x + 1, i_y - 1

        # Now it restarts the cycle
        elif grad_ang > (22.5 + 135) and grad_ang <= (22.5 + 180):
            neighb_1_x, neighb_1_y = i_x - 1, i_y
            neighb_2_x, neighb_2_y = i_x + 1, i_y

        # Non-maximum suppression step
        if width > neighb_1_x >= 0 and height > neighb_1_y >= 0:
            if mag[i_y, i_x] < mag[neighb_1_y, neighb_1_x]:
                mag[i_y, i_x] = 0
                continue

        if width > neighb_2_x >= 0 and height > neighb_2_y >= 0:
            if mag[i_y, i_x] < mag[neighb_2_y, neighb_2_x]:
                mag[i_y, i_x] = 0


cv2.imshow("Non-maximum Suppression", mag)
cv2.imwrite("Non-maximum Suppression.jpg", mag)

for i_x in range(width):
    for i_y in range(height):

        grad_mag = mag[i_y, i_x]

        if grad_mag < weak_th:
```

```
                mag[i_y, i_x] = 0
            elif grad_mag >= strong_th:
                mag[i_y, i_x] = 255


    def hysteresis(img, weak, strong=255):
        M, N = img.shape
        for i in range(1, M - 1):
            for j in range(1, N - 1):
                if (img[i, j] == weak):
                    try:
                        if ((img[i + 1, j - 1] == strong) or (img[i + 1, j] == strong)
or (img[i + 1, j + 1] == strong)
                                or (img[i, j - 1] == strong) or (img[i, j + 1] ==
strong)
                                or (img[i - 1, j - 1] == strong) or (img[i - 1, j] ==
strong) or (
                                        img[i - 1, j + 1] == strong)):
                            img[i, j] = strong
                        else:
                            img[i, j] = 0
                    except IndexError as e:
                        pass
        return img

    hysteresis_img =  hysteresis(mag, weak_th)


    cv2.imshow("Double Thresholding", mag)
    cv2.imwrite("Double Thresholding.jpg", mag)
    cv2.imshow("Hysteresis", hysteresis_img)
    cv2.imwrite("Hysteresis.jpg", hysteresis_img)

    return hysteresis_img

img = cv2.imread("rubicks3.jpg")
img = cv2.resize(img, (400, 400), interpolation = cv2.INTER_AREA)
cv2.imwrite("input.jpg", img)
canny_img = Canny_detector(img)
cv2.imshow("original", img)
cv2. imshow("canny_output",canny_img)

if cv2.waitKey(0) & 0xFF == 27:
    cv2.destroyAllWindows()
```

Task 2 (a)

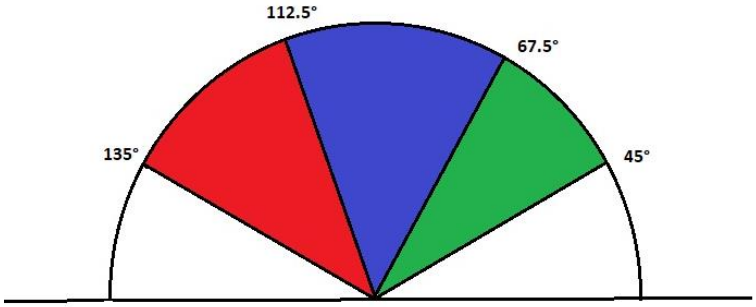

Fig 2.1: New gradient angel direction band.

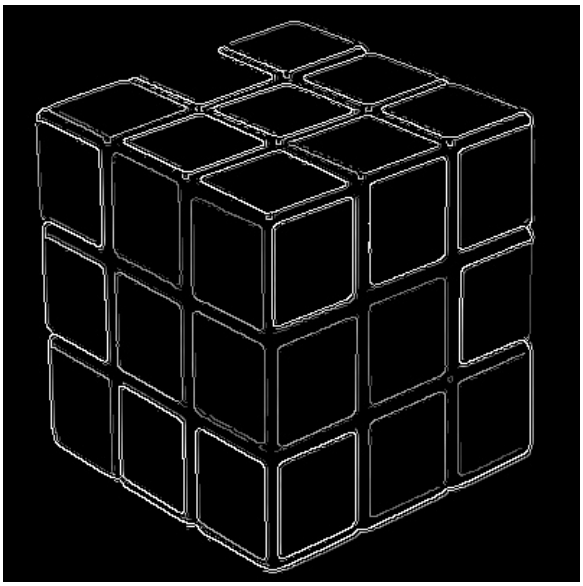

Fig 2.2: Output image using Canny Edge Detection

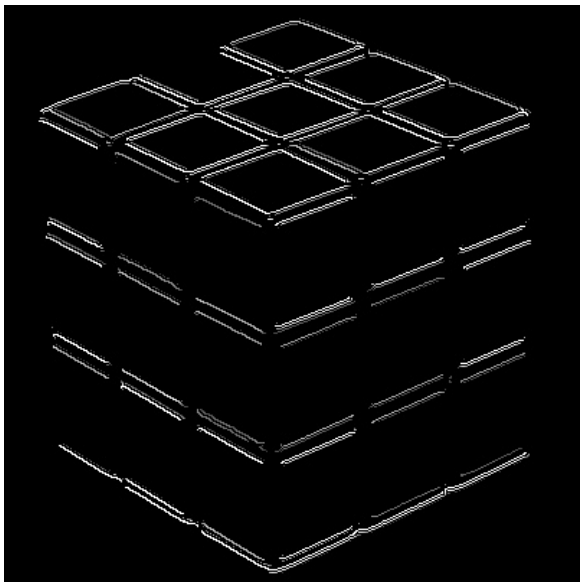

Fig 2.3: Output using new gradient angle direction band

We changed the gradient angle band to be between 45 and 135 degrees. Because of that we would detect edges that are oriented diagonally instead of vertically or horizontally. This would result in a different set of edges being detected in the image. As we can see from the image that, the horizontal and vertical lines are not detected. Only the lines in an angle between 45 and 135 degrees are detected as edges.

Task 2 (b)

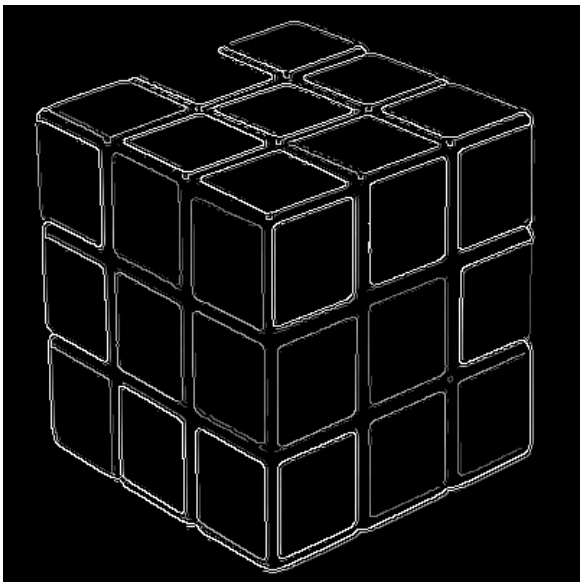

Fig 2.4: Output image using Canny Edge Detection

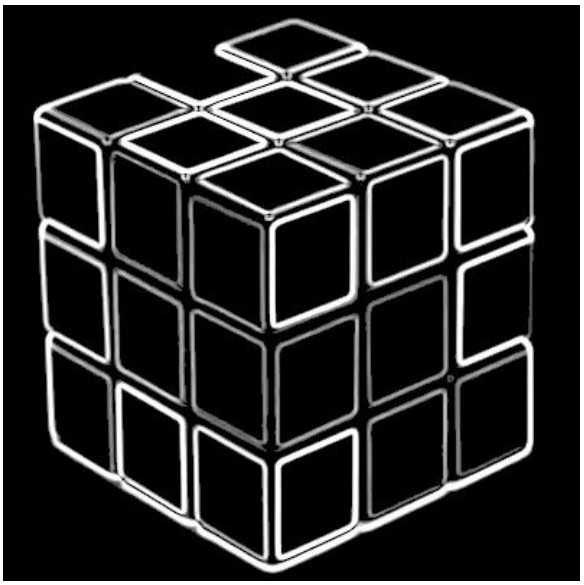

Fig 2.5: Output using Canny without Non-maximum Suppression

As we can see from Fig 2.5 applying Canny Edge detection without non-maximum suppression will give us a output with very thick edges. Non-maximum suppression is an essential step in the Canny Edge Detection algorithm that helps to thin the edges and keep only the strongest response along the edge. This would result in less precise localization of the edges, making it difficult to use the edge detection output for further image processing tasks such as object recognition or tracking.

Task 2 (c)

Task 3

The speed of the canny edge detection can be done by various ways. Some of them are,

1. **Image Down-sampling:** Down-sampling the input image reduces its size and thus the computational load of the edge detection algorithm. However, this approach may result in loss of detail and accuracy.

2. **Parallel Processing:** Canny Edge Detection can be parallelized to speed up the computation. By dividing the input image into smaller regions and processing them in parallel, the edge detection algorithm can take advantage of multi-core CPUs and GPUs to achieve faster processing times.

3. **Hardware Acceleration:** Edge detection algorithms, including Canny Edge Detection, can be implemented in hardware, such as Field Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), or Graphics Processing Units (GPUs). Hardware acceleration can provide significant speedups compared to software-based implementations, although it may require specialized hardware and expertise.

Task 4

The Features from Accelerated Segment Test (FAST) is a corner detection algorithm that is widely used in computer vision and robotics applications. The steps involved in the FAST algorithm are:

1. Choose a pixel p in the image.
2. Choose a threshold value t.
3. Choose a circle of 16 pixels around p.
4. Check if p is a corner by comparing its intensity value to the intensity values of the pixels on the circle. If more than n (typically 12) of the pixels on the circle are brighter or darker than p by at least the threshold value t, then p is considered a corner.
5. Repeat the above steps for every pixel in the image.

The results of the FAST algorithm can be visualized using the following steps:

1. Input image: Here is the original image that we will be using for corner detection.
2. Thresholding: We choose a threshold value and apply it to the image. Pixels with intensity values above the threshold are shown in white, while those below the threshold are shown in black.
3. Corner detection: We apply the FAST algorithm to the threshold image to detect corners. The detected corners are shown in red.
4. Final output: The detected corners are overlaid on the original image to produce the final output.

Code:

```
import cv2

# Load the input image
img = cv2.imread('rubicks3.jpg')

# Convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Set FAST parameters
fast = cv2.FastFeatureDetector_create(threshold=50)

# Detect corners using FAST algorithm
keypoints = fast.detect(gray, None)

# Draw detected corners on the input image
img_with_corners = cv2.drawKeypoints(img, keypoints, None, color=(0, 0, 255))

# Display the output image
```

```
cv2.imshow('FAST corners', img_with_corners)
cv2.imwrite("FASTcorners.jpg", img_with_corners)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Fig 4.1: Input Image

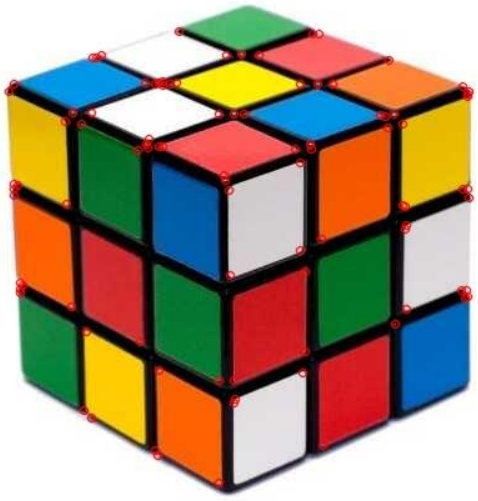

Fig 4.2: After applying FAST corner detection