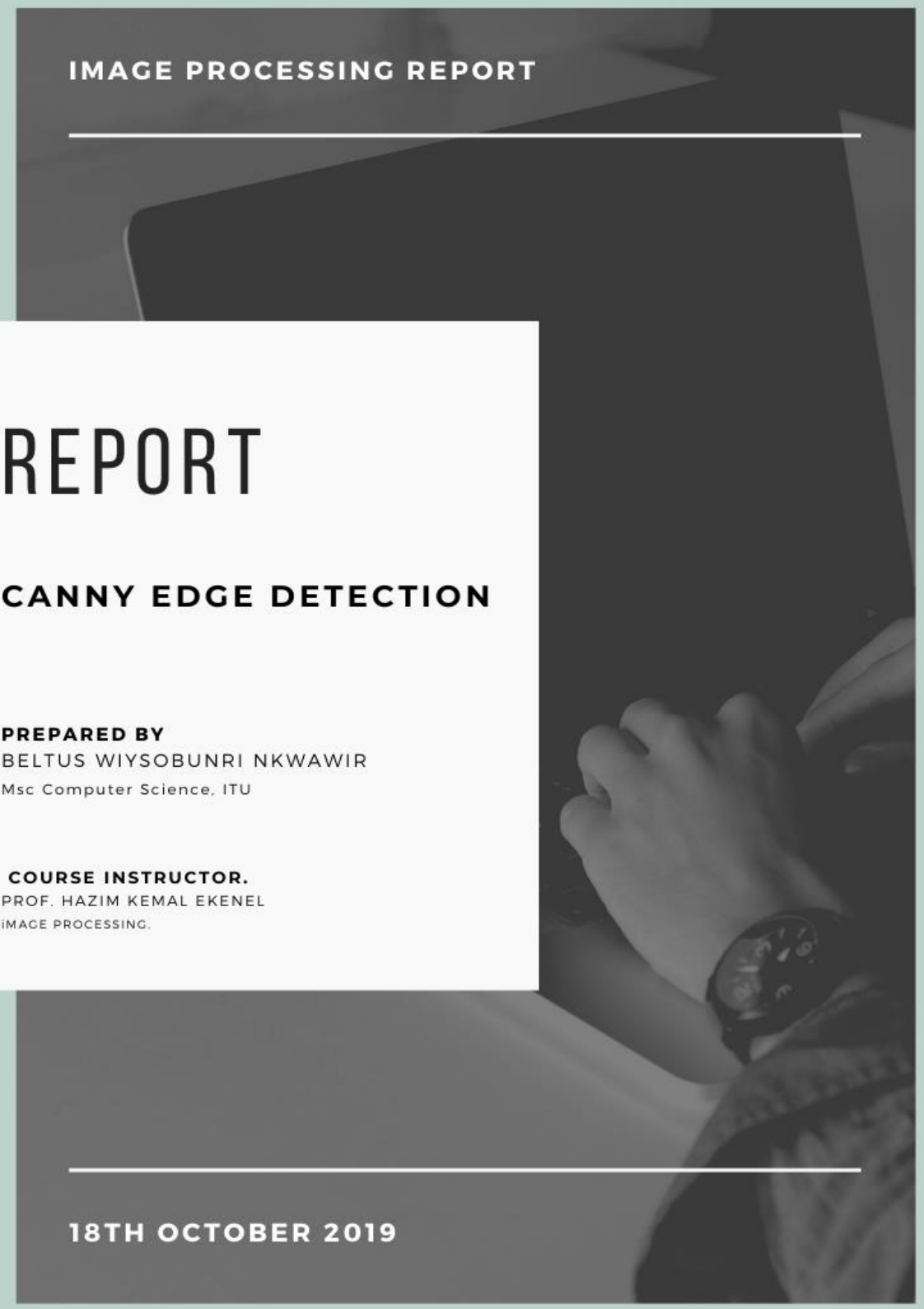
IMAGE PROCESSING REPORT

REPORT

CANNY EDGE DETECTION

PREPARED BY
BELTUS WIYSOBUNRI NKWAWIR
Msc Computer Science, ITU

COURSE INSTRUCTOR.
PROF. HAZIM KEMAL EKENEL
iMAGE PROCESSING.



18TH OCTOBER 2019

OBJECTIVES OF THE HOMEWORK PROJECT.

Implementation of the Canny Edge Detector which can be described by the below six cardinal steps.

General Outline of the Canny Edge Detection Algorithm.

1. Convert to Grayscale
2. Smoothing: Blurring the image to remove noise.
3. Finding gradients: The edges should be marked where the gradients of the image has large magnitudes.
4. Non-maximum suppression: Only local maxima should be marked as edges.
5. Double thresholding: Potential edges are determined by thresholding.
6. Edge tracking by hysteresis: Final edges are determined by suppressing all edges that are not connected to a very strong edge.

Requirements for Running My Canny Edge Program implementation.

This program was written using Ubuntu 18.04.3 LTS operation system and the programming language I used was python version 3.6. In order to successfully run the code of the algorithm the following requirements may be mandatory.

- Create a virtual python environment and ensure it is activated
- Install python 3.6.
- Install jupyter notebook in your environment
- Install the opencv, matplotlibrary, numpy , scipy libraries which where the cardinal backbone behind the success of this project.

Some Important Applications of the Canny Edge Detector.

The Canny edge detector is quite robust and can be used in a diverse number of fields and applications such as:

- In Robotics visions especially in the autonomous self-driving car industries.
- Classification of medical images especially in the finding of fingerprints for “diseases” such as tumors.
- In the future for intelligent traffic control it can be applied in the automatic detection of license plates.
- Also used in the enhancement of satellite images to generate satellite maps.

STEP ONE: CONVERTION OF IMAGE TO GRAY SCALE.

In this first step of the canny process, the images are converted to grayscale to obtain an 8-bit gray-intensity image. This is because for edge detection we are only interested in the changes of the intensity values across the image which are potential edges.

For this implementation I used the **opencv libary** to obtain grayscale images.

gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY).

These images shown below are the test images that were used throughout the project to test the functionality and robustness of the Canny edge algorithm.

STEP TWO : SMOOTHING USING THE GAUSSIAN BLUR FILTER.

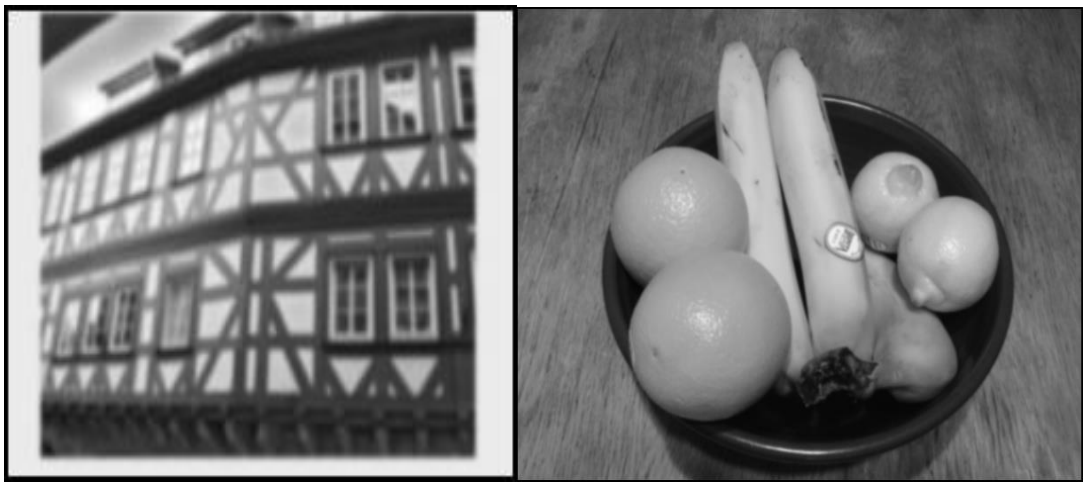
This step is very important in edge detection because the underlining algorithms that are involved in the detection process include derivatives. This means that, the algorithm itself is very sensitive and susceptible to image noise.

For this step I decided to implement the **5 x 5 Gaussian blur filter** in order to mitigate effect of any noise pixels that can be lingering in our input image. However, I noticed there was a trade-off between smoothing and noise reduction. Smoothing resulted in the loss of some potentially useful edges in the original images.

The 5 x 5 gaussian kernel that was convolved with the input images can be seen in the code section. Some images input and output images after applying the Gaussian blur filter can be seen below with an impressive amount of reduced noised.

The results were obtained using the 5 x 5 gaussian kernel below.

Gaussian = (1.0/57) * np.array([[0, 1, 2, 1, 0],[1, 3, 5, 3, 1],[2, 5, 9, 5, 2], [1, 3, 5, 3, 1], [0, 1, 2, 1, 0]])



The results of the gaussian blur stage were on average visually good, however, after performing the other steps of the Canny edge detection algorithm it is clear that the gaussian blur hampered the detection of some true edges in some of the images.

STEP THREE: GRADIENT FILTERING.

This step is a crucial step in the implementation of the canny filter as its output enables us to perform thresholding. In particular this step detects the edge intensity and direction by calculating the gradient of the image using **sobel operators** which is a perfect approximation of the derivative of discrete image points.

Here I applied the sobel kernel both in the x-direction and the y-direction independently and then calculated the magnitude and direction of each point in the image matrix. As previously mentioned, there is always a trade -off between smoothing and good edge localization.

X – Direction Kernel

-1	0	1
-2	0	2
-1	0	1

Y – Direction Kernel

-1	-2	-1
0	0	0
1	2	1

These are the resulting outputs that I got from the test images .

A) Smoothed Image *B) Gradient X-Direction* *C) Gradient Y-Direction.*

A) Smoothed Image *B) Gradient X-Direction* *C) Gradient Y-Direction.*

A) Smoothed Image *B) Gradient X-Direction* *C) Gradient Y-Direction.*

A) Smoothed Image *B) Gradient X-Direction* *C) Gradient Y-Direction.*

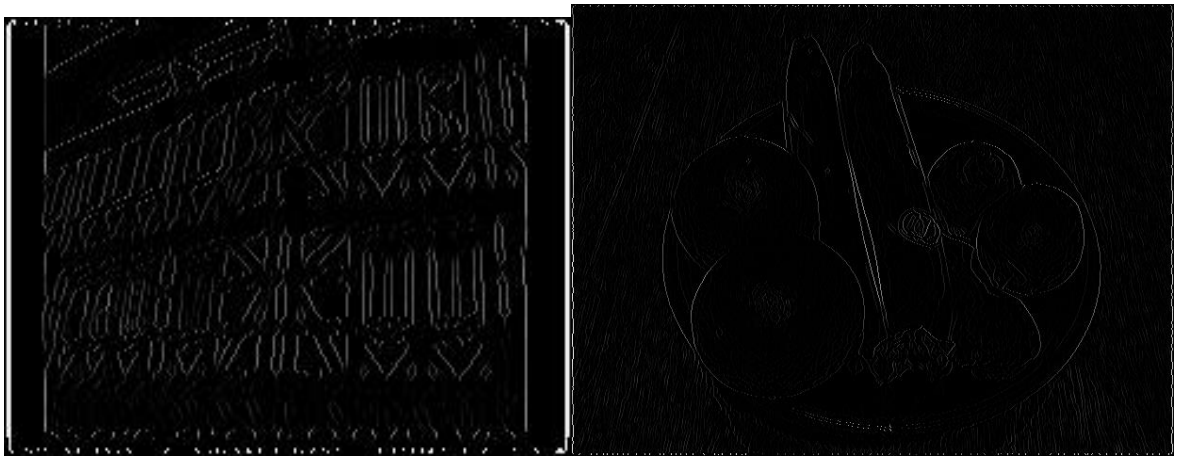
A) Smoothed Image *B) Gradient X-Direction* *C) Gradient Y-Direction.*

A) Smoothed Image *B) Gradient X-Direction* *C) Gradient Y-Direction.*

STEP 4 : NON - MAXIMUM SUPPRESSION.

The aim of this step is generally reduced or thin-out the edge pixel that are found in the Intensity image obtained from gradient derivative. This is because for edge detection we focus on the existence of an edge and not on its thickness. For non-maximum suppression the algorithm I used takes each point on the gradient intensity matrix and finds the pixels with the maximum value in the edge directions.

The results of my implementation with the test images is shown below:



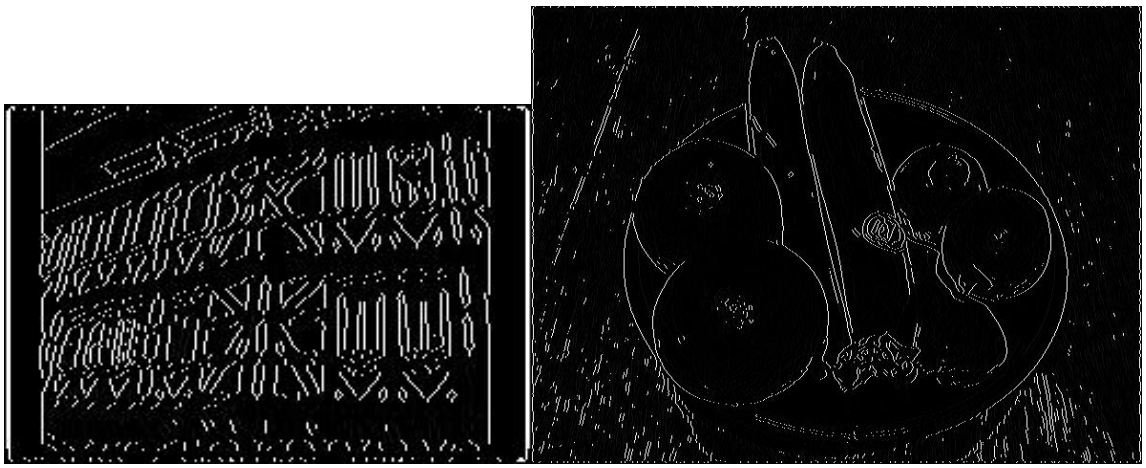
In the images above we can clearly see that the thick edges from the derivative process have been thinned out and just the outline of the edges are visible.

STEP 5: DOUBLE THRESHOLDING.

In this step I specified a fixed value for the low and high threshold values which was used across all test images, however, from the results, it was evident that, a fixed low and high threshold value was not the best option. This is because, with some images, double thresholding yielded good results while other input images, real edges with very low intensity values were considered as noise and set to zero. As a result, I decided to implement this process dynamically by choosing the low and high threshold values based on the maximum intensity of the image pixel value. This was done by setting a high and low threshold coefficients.

I did some test by changing both coefficient values while observing the outputs and at the end, I settled for a Low threshold coefficient of **0.01** and a high threshold coefficient of **0.1.**

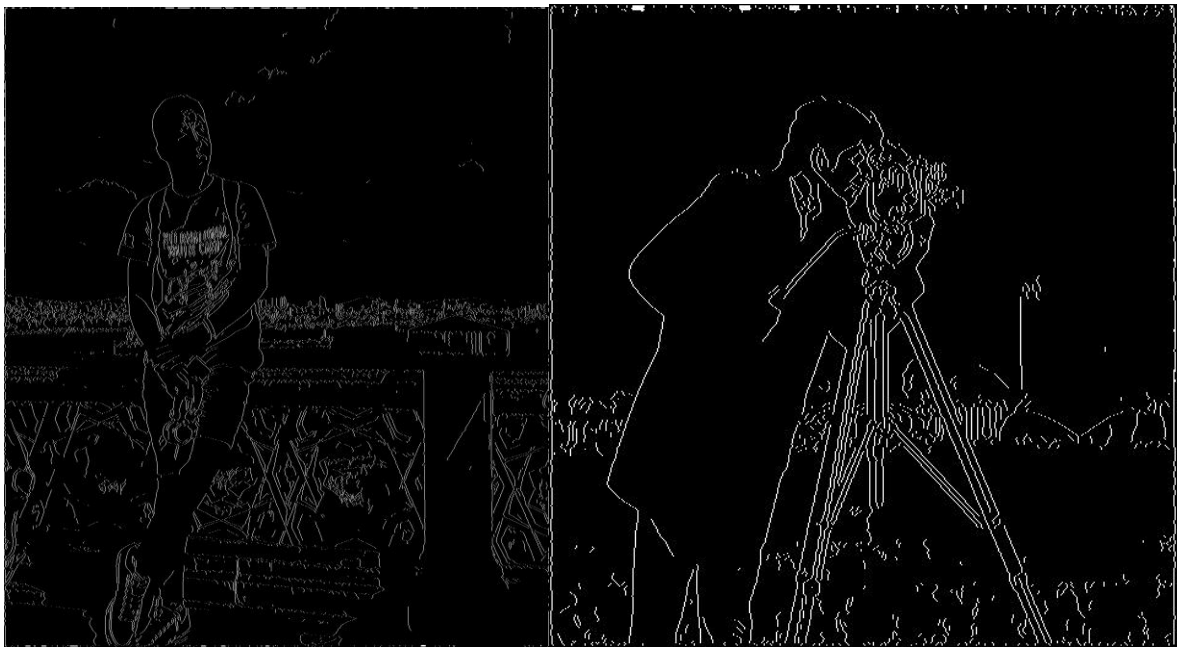
The result obtained with these values can be seen below.

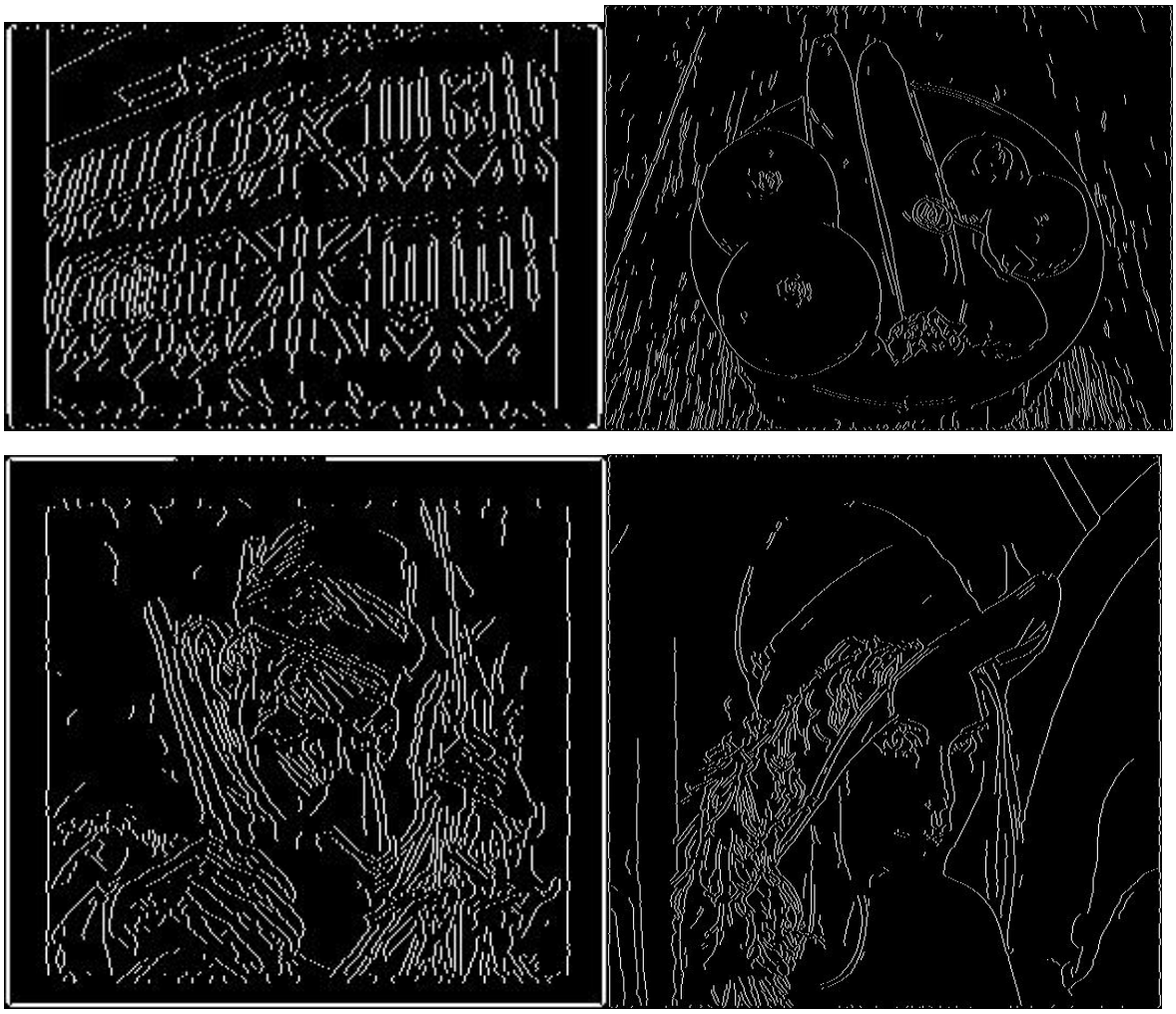


STEP 6: EDGE TRACKING BY HYSTERESIS.

In my implementation, the pixels that were classified as weak were used as input into the edge tracking by hysteresis method. Every weak pixel is compared with its 8 neighbours and if one of the neighbours is a strong pixel, this pixel is automatically classified as strong, therefore an edge, else it is set to zero.

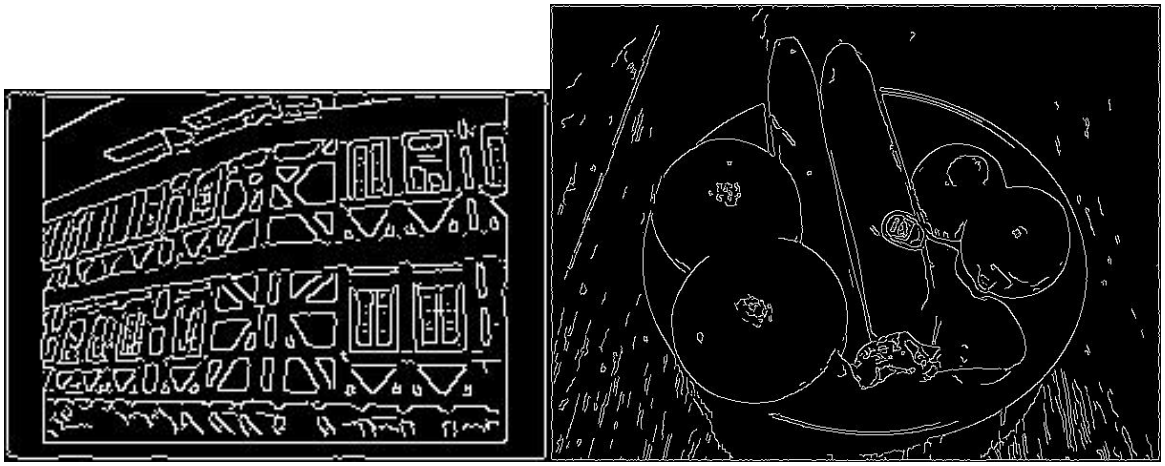
The final results of the canny edge detection algorithm yielded the following as can be seen below.





COMPARISON OF RESULTS WITH OPENCV CANNY FUNCTION.

In order to investigate and validate the performance of my implemented algorithm, I tested the same images using the opencv canny function. As recommended by John Canny himself, the ratio of low threshold to high threshold should be 1:2 or 1:3. Heeding to this recommendation, I chose a low threshold of 40 and high threshold of 80 for the opencv canny function to obtain the edged detected images below.



CONCLUSION.

Comparing the outputs from the opencv canny function to the output from my implementation, it unequivocally performs almost as good as the opencv implementation.

This marks the end of this report.