INTRODUCTION

We implemented different Edge Detecting algorithms such as Prewitt, Sobel and Laplacian of Gaussian Operator in this lab.

Edges are the one of the examples of images' features which are the local, detectable parts of the image. Edges correspond to parts which pixel values have sharp variations. Edge detectors are basically trying to approximate first or second order derivatives.

Rest of the report will demonstrate the implementations of these operators and explanations. Then, their results will be shown. At the end of each part, operators will be discussed.

1. Prewitt Operator

Prewitt Operator is one of the *Edge Detecting* operators implemented during the lab. Prewitt operator includes 2D derivative with two different kernels. These two different can be seen below:

-1	0	1
-1	0	1
-1	0	1

X Filter

-1	-1	-1
0	0	0
1	1	1

Y Filter

Each kernel has own duty: X filter is used for detecting Vertical Edges. Y filter is used for detecting Horizontal Edges.

After convolving our input image with the above filters, we can calculate our gradient image with the following formula:

$$G(p) = \sqrt{G_x(p)^2 + G_y(p)^2}$$

After obtaining gradient image, we need to binarize it to detect edges in the greyscale image. This binarization is done according to parameter of threshold value (user gives this value). If a pixel value is higher than threshold, then its value defined as 255 (white). If the pixel value is lower than threshold, then its value defined as 0 (black).

1.2 My Prewitt Operator Function Explanation

In my implementation, I started as always by ensuring that the image is grey scale.

```
% Ensuring grey scale image
[row,col,ch] = size(img);
if(ch == 3)
    img = rgb2gray(img);
end
```

Then, I defined my Filter Kernels which is shown in the upper part. Since we have 3x3 kernel size, our k value is 1. Then, I converted my image pixel data to double format to be able to perform mathematical operations. I created a matrix with the same size of input image with zeros.

Then, I used double for loop to iterate on the image while doing the convolution operations for both filter kernels.

```
13 -  data = double(img);
14    % % Double for loop for sliding window
15 -  new_img_data_X = zeros(row, col);
16 -  new_img_data_Y = zeros(row, col);
17 -  window_size = 1;
18 -  for i = window_size+1:row-window_size %Edges are zero
19 -      for j = window_size+1:col-window_size
20 -          my_window = data((i-window_size):(i+window_size), (j-window_size):(j+window_size)); %take the window
21 -          value_X = sum(sum(my_window .* x_filter)); %Convolution operation
22 -          value_Y = sum(sum(my_window .* y_filter)); %Convolution operation
23
24 -          new_img_data_X(i, j) = value_X;
25 -          new_img_data_Y(i, j) = value_Y;
26 -      end
27 -  end
28 -  new_img_X = uint8(new_img_data_X);  %Prewitt X filtered image
29 -  new_img_Y = uint8(new_img_data_Y);  %Prewitt Y filtered image
```

In the for loop part, I decided to leave borders as zero. I could also do zero padding as I did in the previous lab. Since I decided to leave borders as zero, I started to iterate from the index (k+1, k+1) in the image. In each iteration, I took the 3x3 image window to convolve with my filter kernel (this step corresponds to *code line 20*). Then, I did the convolution operations (these steps correspond to *line 21 & 22*). After convolutions, I put the results to the centers of my window. After loops, I converted my images to uint8 format back again to visually show them.

Then, I applied the Gradient Image formula with these two images as following way:

```
31 -    new_img_data_G = sqrt(new_img_data_X.^2 + new_img_data_Y.^2);
```

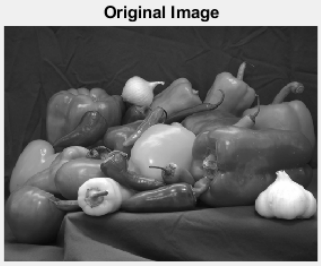
Then, I used “find” function of the MATLAB to find indexes of pixel values which are higher than threshold. Then, I made this indexes as 255 (white) to show edges. Other pixels are equal.

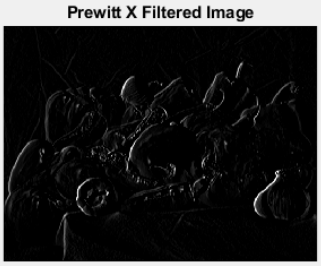
```
37 -    indexes = find(new_img_data_G>threshold);
38
39 -    new_img_data_Edges = reshape(new_img_data_Edges, [row2*col2, 1]);
40 -    new_img_data_Edges(indexes) = 255;
41 -    new_img_data_Edges = reshape(new_img_data_Edges, [row2, col2]);
```

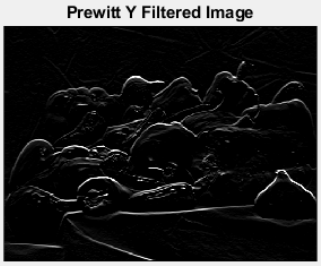
At the end, I showed all processed images in the same plot. These images can be seen in the result part.

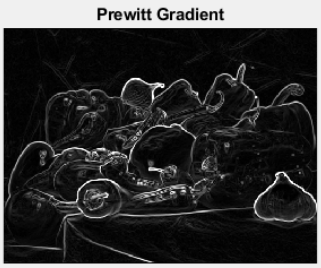
1.3 Prewitt Operator Results

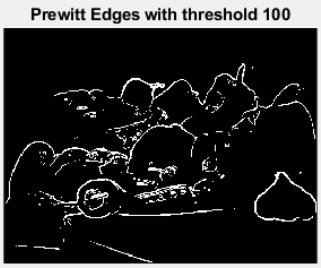
Peppers Images with different threshold values:

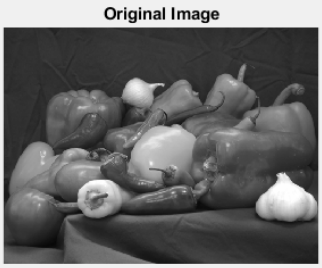


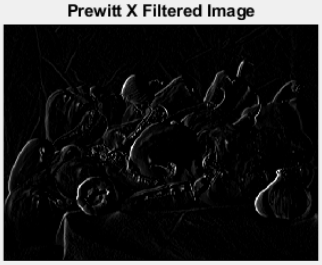


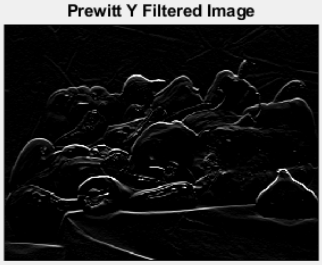


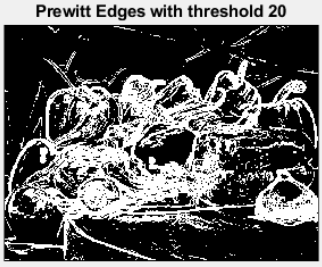






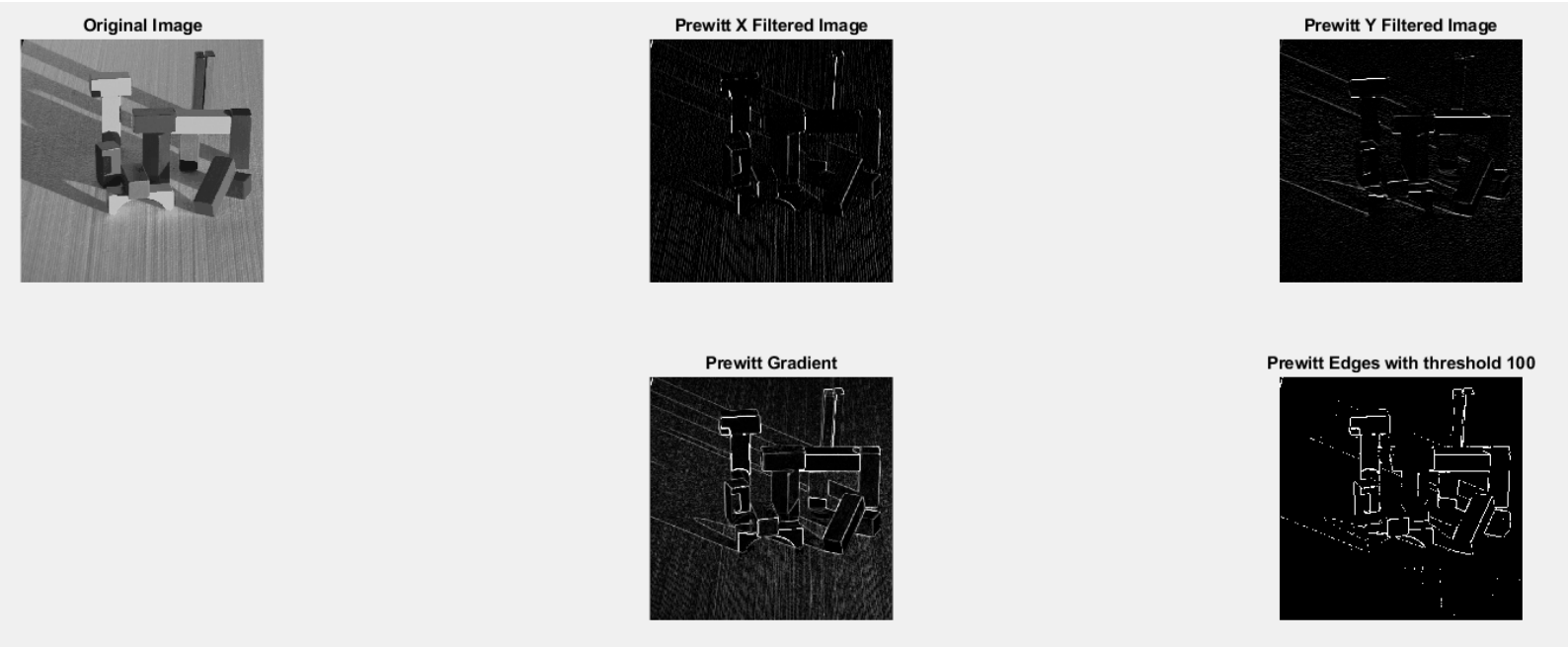


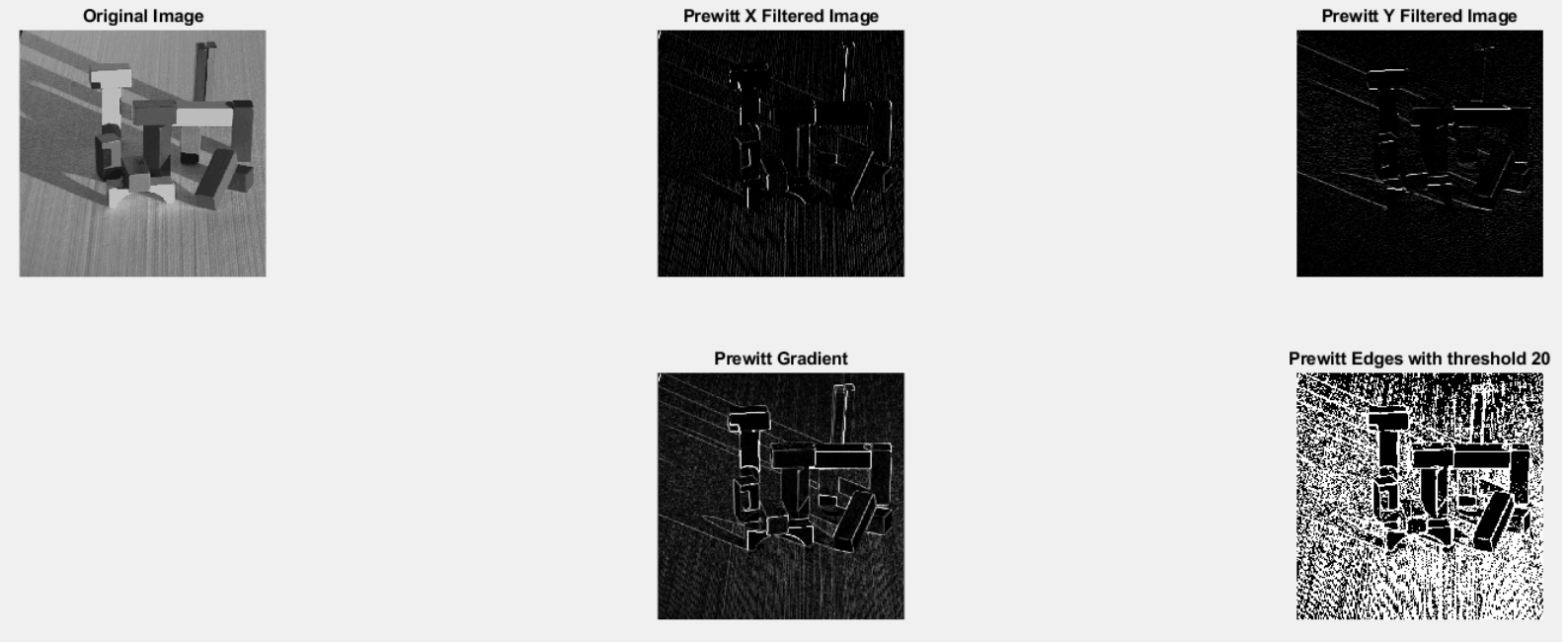




Ege Erdoğan 25331

Blocks Images with different threshold values:





1.4 Prewitt Operator Discussion

We can observe that Prewitt Operator can detect edges in the images. We can also observe that our threshold parameter has a crucial role for the algorithm. When we set our threshold lower, we are observing extra white pixels in the output image. On the other hand, if we set threshold too high, we can not capture all edges in the image. This threshold value can be different for various images. Therefore, we need to find optimal threshold value for each input images.

We can also observe that X filter catches vertical edges, and Y filter catches horizontal edges in the input images.

2. Sobel Operator

Sobel Operator is one of the *Edge Detecting* operators implemented during the lab. Sobel Operator following same procedures with the Prewitt Operator. The only difference is that Sobel Operator's filter kernels. These filter kernels can be seen below:

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

X Filter

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

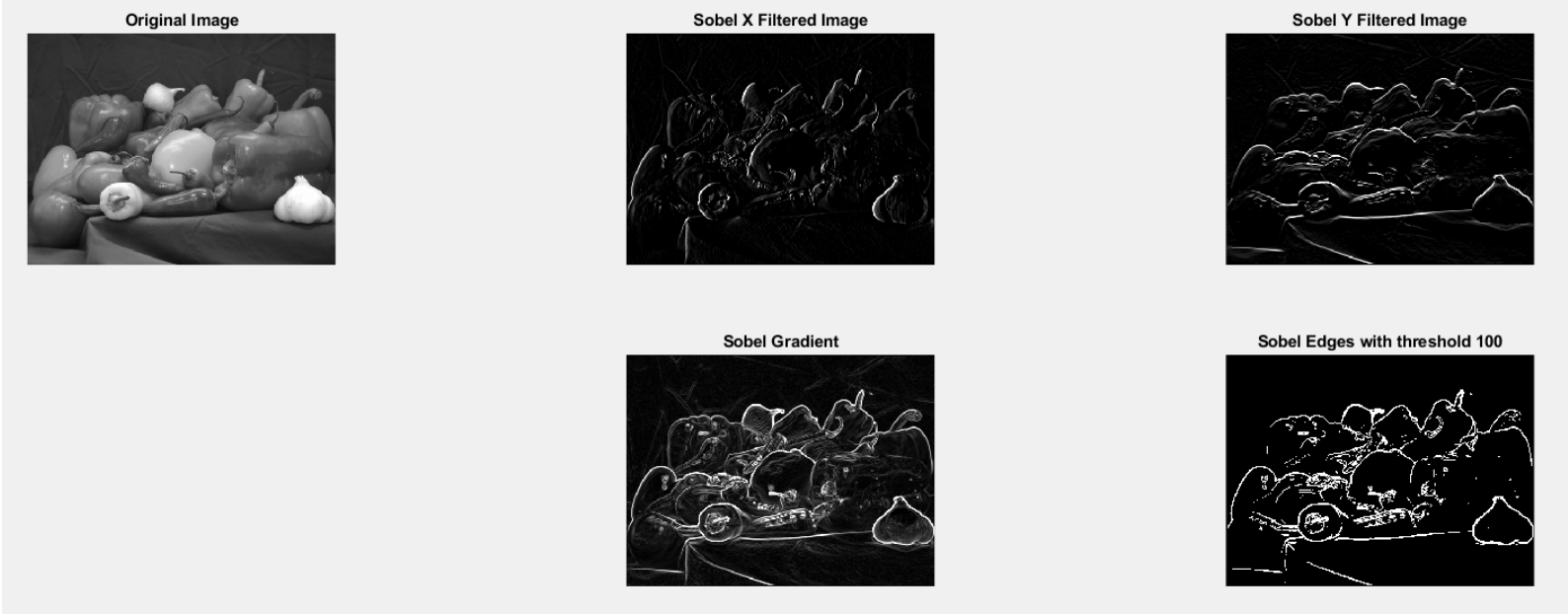
Y Filter

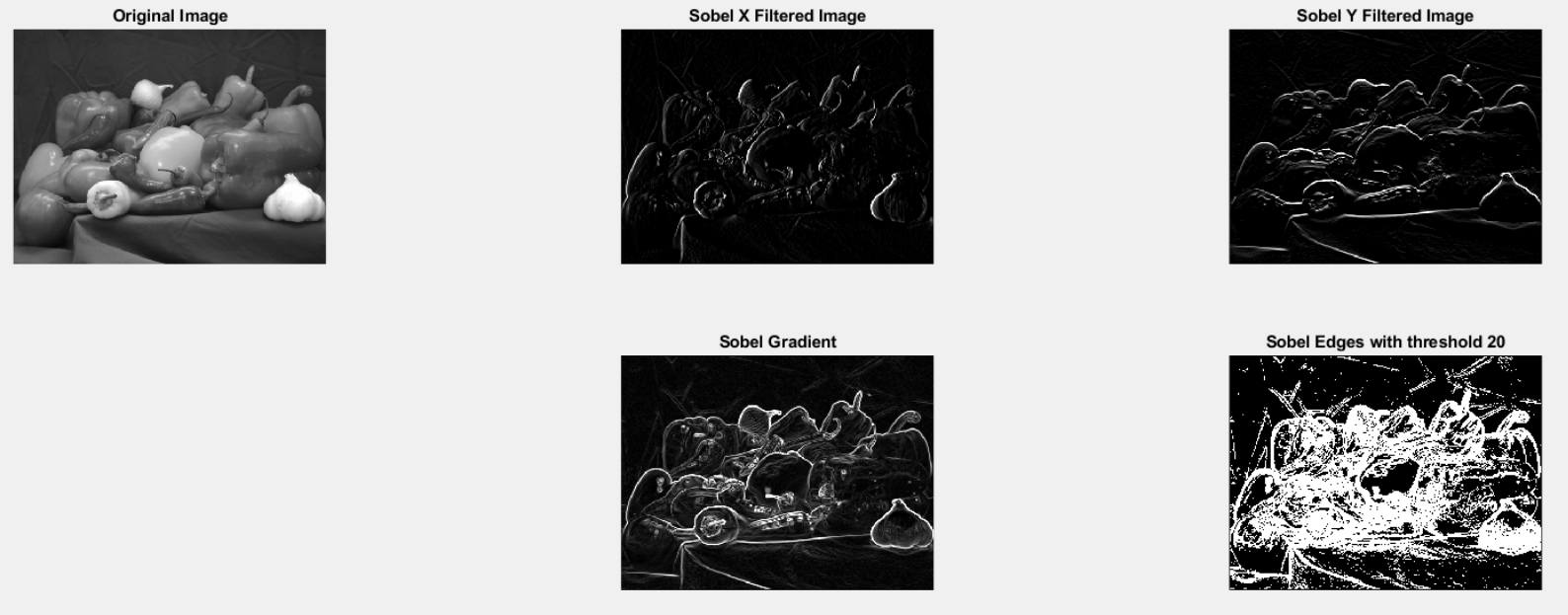
2.2 My Sobel Operator Function Explanation

My function is totally same with Prewitt Operator. I just changed my filter matrices.

2.3 Sobel Operator Results

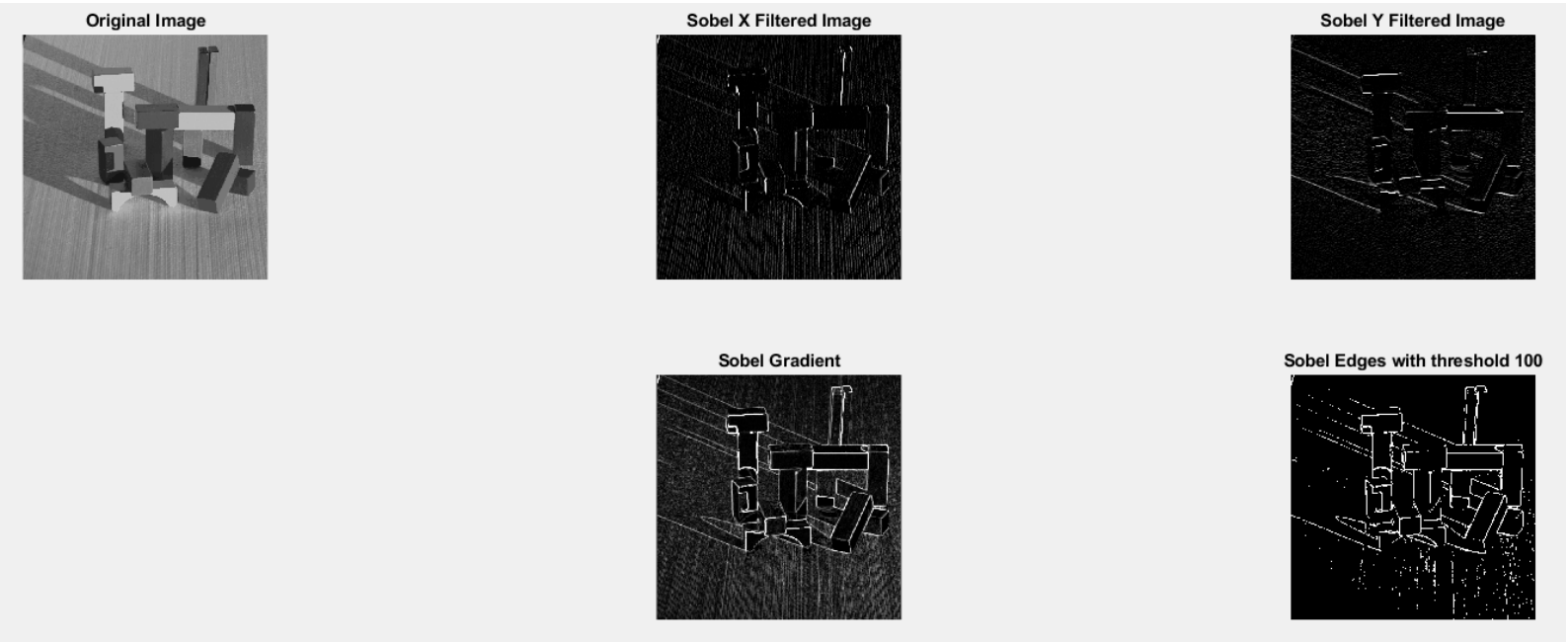
Peppers Images with different threshold values:

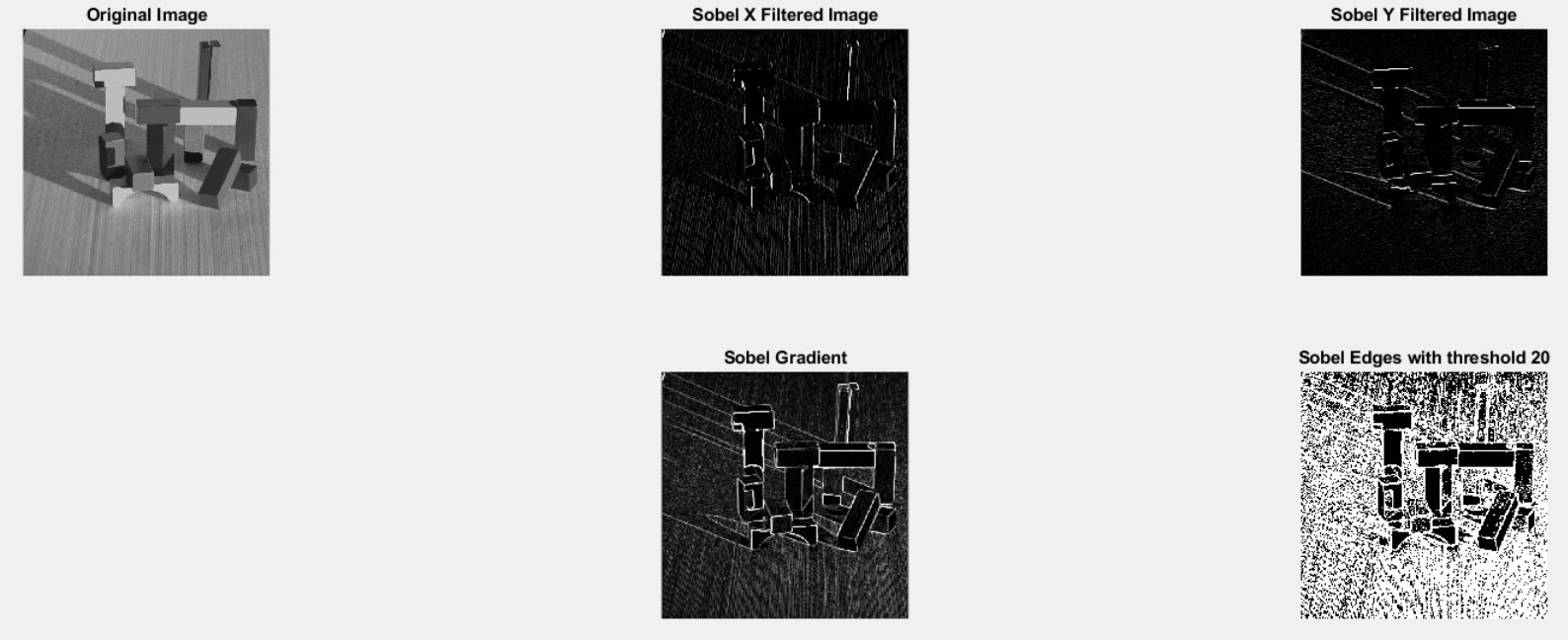




Ege Erdoğan 25331

Blocks Images with different threshold values:



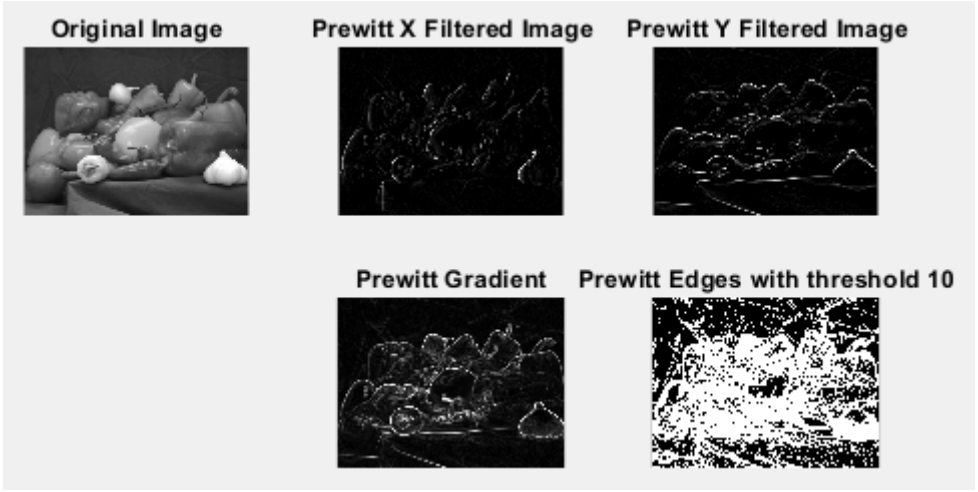


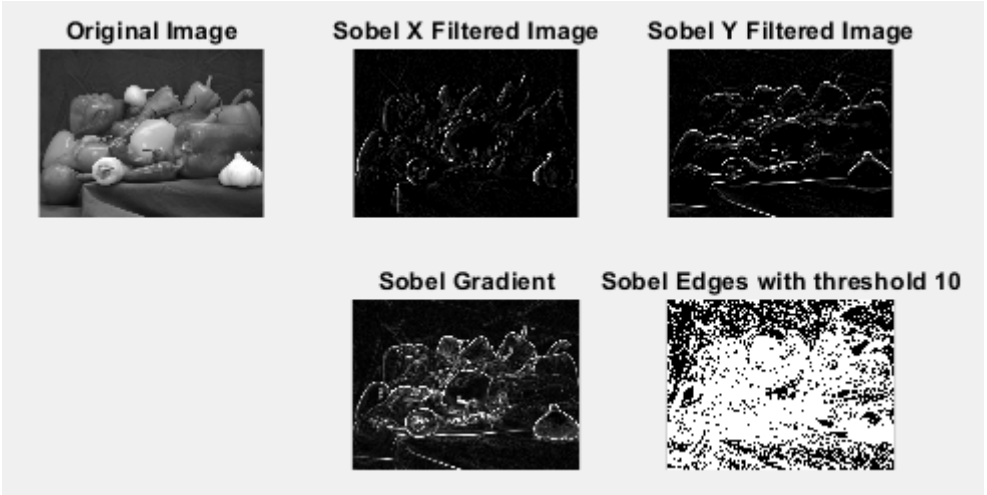
2.4 Sobel Operator Discussion and Comparison with Prewitt

We can observe that Sobel Operator can detect edges in the images. Threshold value plays crucial role in the Sobel Operator.

We can observe that threshold values have different impacts on the Prewitt and Sobel operators. **Sobel Operator have higher weights for the pixels that are closer to centre. On the other hand, Prewitt Operator have same weights for the window.**

We can observe that Sobel Operator have higher number of white pixels compared to Prewitt for low threshold values. This can be observed in the below images:





3. Laplacian of Gaussian Smoothed Image (LOG)

Laplacian of Gaussian (LOG) is one of the *Edge Detecting* operators implemented during the lab. Laplacian approximates, uses second order derivative of the image. Therefore, noise in the image have higher influence on the operator compared to Sobel and Prewitt. Thus, we need to reduce the noise by applying smoothing algorithm Gaussian filter before Laplacian operator. Hoca showed the derivation of the filter kernel which is used to calculate Laplacian of image in the lecture. Laplacian uses the following kernel by convolving with input image to calculate Laplacian of image:

0	1	0
1	-4	1
0	1	0

After convolution, we can observe zero crossings in the image.

3.1 My LOG Function Explanation

In my implementation, I started as always by ensuring that the image is grey scale. Then, I smoothed my input image with the Gaussian filter as LOG filter sensitive to noise in the image. Then, I defined my above kernel and convolved with my smoothed image.

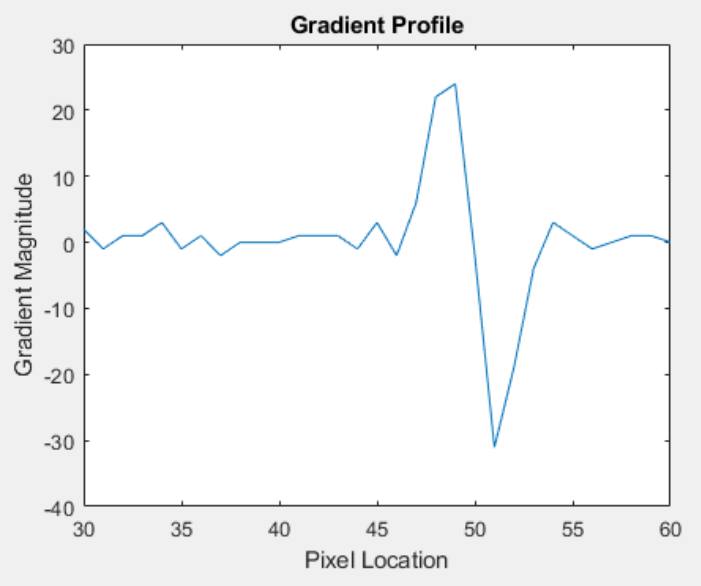
```
16 -    window_size = 1;
17 -    new_img_data = zeros(row, col);
18 -  for i = window_size+1:row-window_size %Edges are zero
19 -      for j = window_size+1:col-window_size
20 -          my_window = data((i-window_size):(i+window_size), (j-window_size):(j+window_size)); %take the window
21 -          value = sum(sum(my_window .* kernel)); %Convolution operation
22 -          new_img_data(i, j) = value;
23 -      end
24 -  end
25 -  new_img = uint8(new_img_data);
```

At the end, we have LOG filtered image.

We can choose one line from this filtered image and plot its magnitude. I took the 150th line between 30th and 60th columns with the following code:

```
27 -    segment = new_img_data(150, 30:60);
```

When we plot this segment, we can see the below result:



As we can see above, we have a zero crossing (edge) in this segment.

Now we must find zero crossings. Firstly, I created an image with all zeros with same size LOG filtered image. Again, I used double for loops to iterate on my image. I recorded the values of center and neighbours of pixels. We need to look pixels that have non-zero gradient magnitudes. However, there may be noise effect. Therefore, we need a threshold value to reduce this noise effect. This threshold value may vary between -5 and 5. Then, we need to look for sign change between pixels to see if there is any Edge or not. This sign change can happen in three different scenarios. There may be sign change between left and right pixels of center, or upper and down pixels of center, or diagonals of center. Even, we detect sign change, it may a minor change. Therefore, we need to have another threshold value to detect

significant sign changes. In other words, magnitude changes between pixels should be higher than predefined threshold value to be considered as Edge. Following algorithm implements the above ideas:

We do not include corner pixels such as (2, 2) because corner pixel cannot have all neighbours. While we are checking diagonal sign change, if we compare it with center pixel and its diagonal, the edges are becoming smoother. I could not find the reason of it. Therefore, I checked sign change with center and its diagonal neighbours one by one.

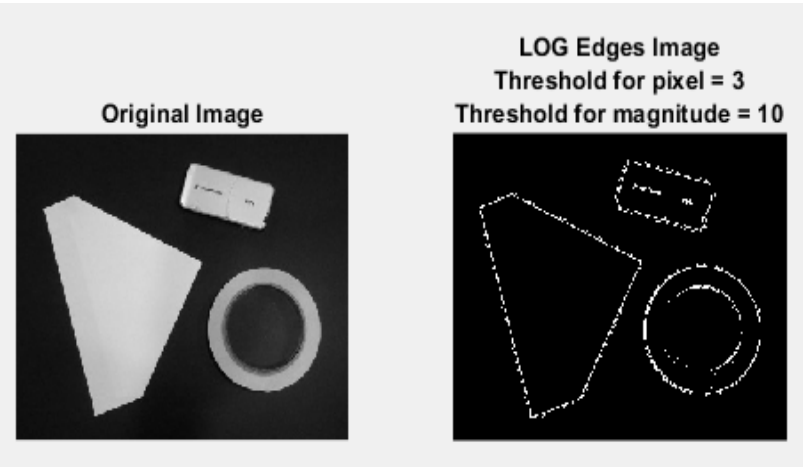
```
new_img_data_Edges = zeros(row2,col2);
for i = 2:row-1
    for j = 2:col-1
        center = new_img_data(i, j);
        left = new_img_data(i, j-1);
        right = new_img_data(i, j+1);
        upper = new_img_data(i-1, j);
        down = new_img_data(i+1, j);
        upper_left = new_img_data(i-1, j-1);
        upper_right = new_img_data(i-1, j+1);
        down_left = new_img_data(i+1, j-1);
        down_right = new_img_data(i+1, j+1);

        if (center > Threshold2)
%       Case 1 left/right zero crossing
        if (left*right < 0) %Sign Change
            if (abs(left - right) > Threshold)
                new_img_data_Edges(i, j) = 255;
            end
 %      Case 2 up/down zero crossing
        elseif (upper*down < 0) %Sign Change
            if (abs(upper - down) > Threshold)
                new_img_data_Edges(i, j) = 255;
            end
  %     Case 3 Diaganols
        elseif (center*down_right < 0) %Sign Change
            if (abs(center - down_right) > Threshold)
                new_img_data_Edges(i, j) = 255;
            end
        elseif (center*down_left < 0) %Sign Change
            if (abs(center - down_left) > Threshold)
                new_img_data_Edges(i, j) = 255;
            end
        elseif (center*upper_left < 0) %Sign Change
            if (abs(center - upper_left) > Threshold)
                new_img_data_Edges(i, j) = 255;
            end
        elseif (center*upper_right < 0) %Sign Change
            if (abs(center - upper_right) > Threshold)
                new_img_data_Edges(i, j) = 255;
            end
        end
        end
```

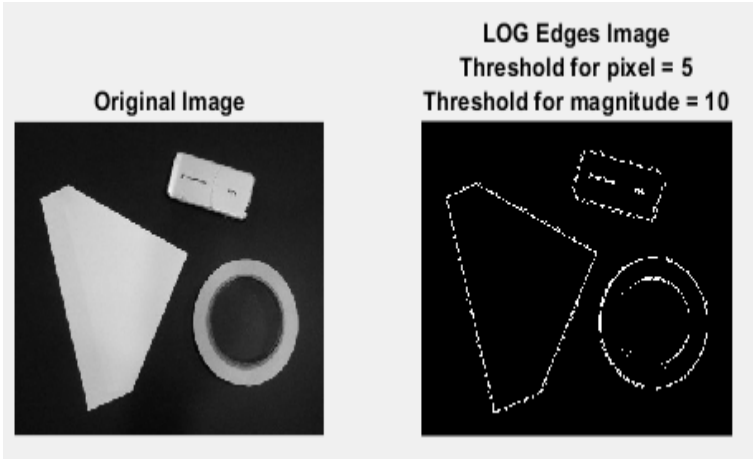
3.2 LOG Operator Results

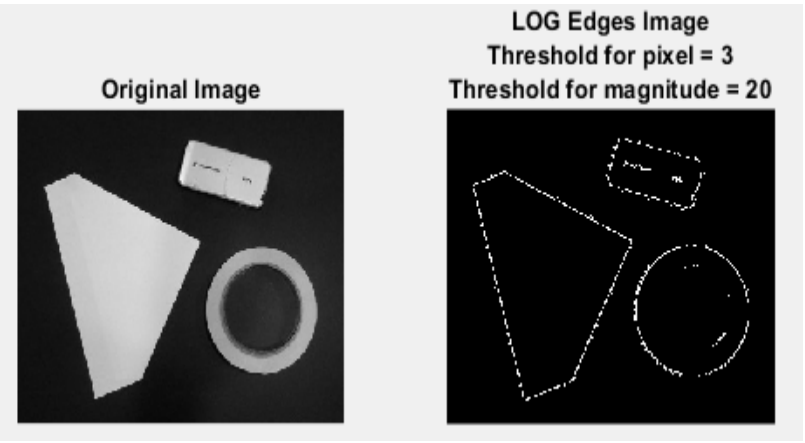
I tried different threshold values and observe their effects on the output image.

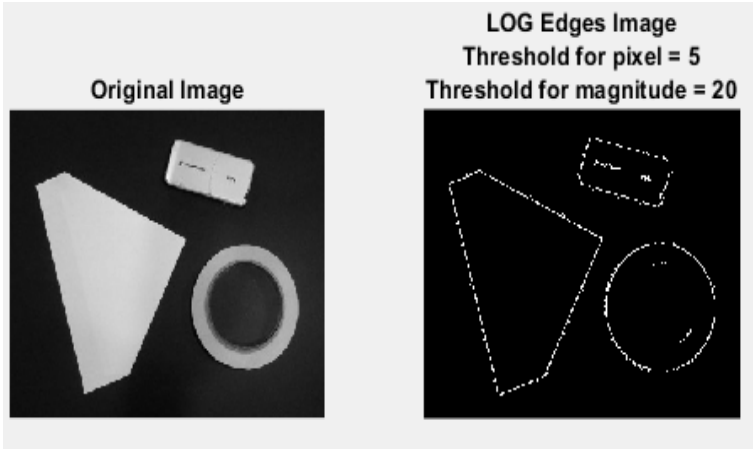
Effects of Different Thresholds on Image Contours

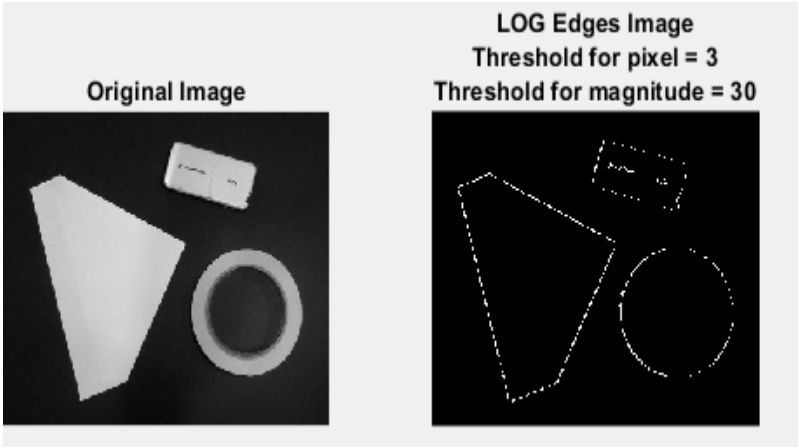
1. Threshold for pixel's gradient magnitude; 2. Threshold for magnitude of sign change

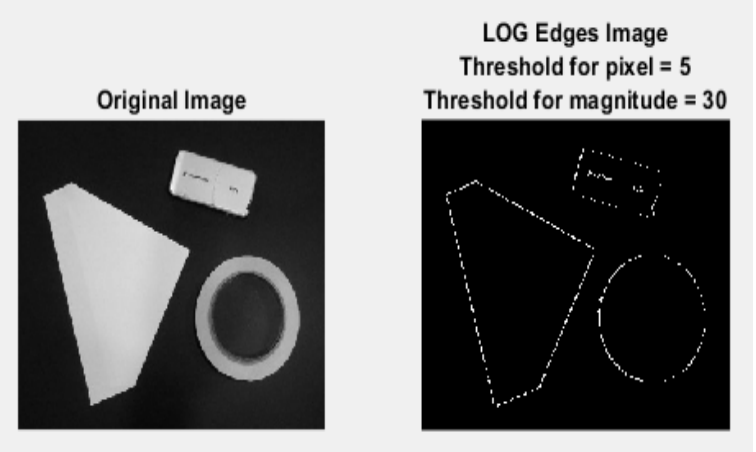










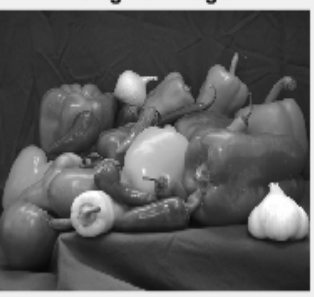


Ege Erdoğan 25331

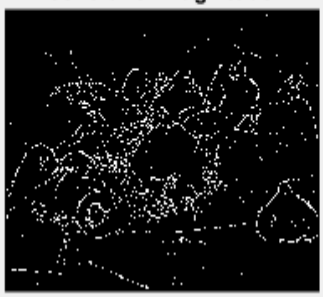
Effects of Different Thresholds on Image Peppers

1. Threshold for pixel's gradient magnitude; 2. Threshold for magnitude of sign change

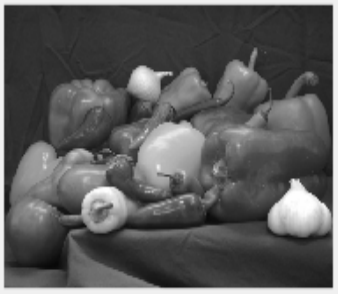
Original Image



LOG Edges Image
Threshold for pixel = 3
Threshold for magnitude = 2



Original Image



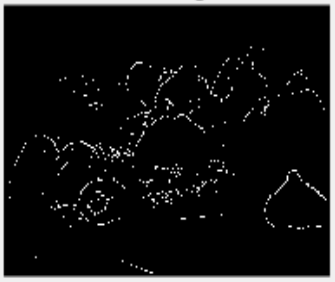
LOG Edges Image
Threshold for pixel = 5
Threshold for magnitude = 2

Original Image

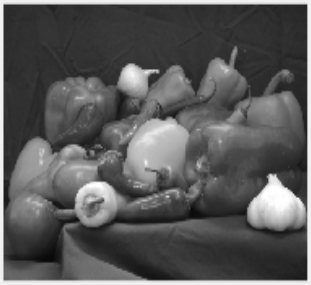
LOG Edges Image
Threshold for pixel = 3
Threshold for magnitude = 10

Original Image

LOG Edges Image
Threshold for pixel = 5
Threshold for magnitude = 10

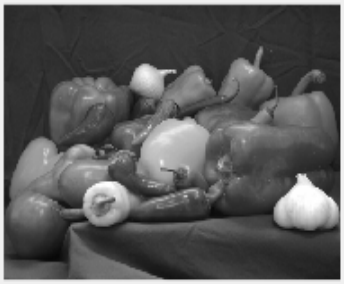


Original Image



LOG Edges Image
Threshold for pixel = 3
Threshold for magnitude = 20

Original Image



LOG Edges Image
Threshold for pixel = 5
Threshold for magnitude = 20

3.3 LOG Operator Discussion

We are able to detect Edges with the LOG filter. However, it is sensitive to noise. Therefore, we need a smoothing before using LOG filter.

We have two different threshold values for different purposes. One of them for the pixel's Gradient Magnitude as there may be noise which cause to non-zero pixel value. Other threshold value for the magnitude of sign change. We can observe that both of them plays critical role. If we do not carefully choose these threshold values, we may miss the Edges or we may catch non edge pixels in the output image. For instance, when we go from threshold value 10 to 30 in the Object Contour Image, we lost some edges in the output image.

3.4 Comparison with Sobel and Prewitt

LOG Operator is more **sensitive** to noise. Therefore, it should have the Smoothing operation beforehand. We have only one Filter Kernel for the convolution operation to calculate 2^{nd} derivative in LOG operator. On the other hand, we have 2 Filter Kernels in the Sobel / Prewitt Filters. While LOG operator calculates the change in the gradient, Sobel / Prewitt filters calculate the gradient. Therefore, they are suitable for different scenarios. For instance, if the gradient is stable for the part of image, LOG operator will output 0 as it does not change. On the other hand, Sobel / Prewitt Filters will output the value of gradient.