EE 417 COMPUTER VISION
LAB#2
POST-LAB REPORT

Goktug Korkulu
27026

This lab report consists of 2 main operations. Namely, **Linear Filtering** and **Edge Detection**. Specifically, Linear Filtering will be applied by using ***Gaussian Filtering*** and Edge Detection will be applied by ***Prewitt Operator***, ***Sobel Operator*** and ***Laplacian of Gaussian (LoG)*** operator.

As always, the functions that was written are as generic as possible, meaning that they all will not assume anything about the inputs such as size, type and color.

Linear Filtering

Gaussian Filtering

Description: Gaussian Filtering is a linear operator that involves the local convolution of a given image with a filter kernel of samples of the 2D Gaussian function. Gaussian Filtering is used for smoothing the pictures by eliminating outliers in image values which are considered as noise in a given context. Since Gaussian Function is continuous function and the pixels are discrete, we create 2D matrix in order to apply the function to discrete values. For instance, we will use below 5x5 kernel in order to apply Gaussian approach to our input images.

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

/273

Figure 1 : Approximated Gaussian Kernel

Implementation:

"Now write a function that takes an image as the input and returns the 'Gaussian smoothed' version of the image. You will use 5x5 approximated Gaussian Filter kernel in figure1."

```
1   %% Gaussian Filter
2   clear; close all; clc;
3   img = imread("Lab2 - Images/baboon.png");
4   lab2gaussfilt(img);
5
```

Figure 2 : the main function call for Gaussian Filtering function

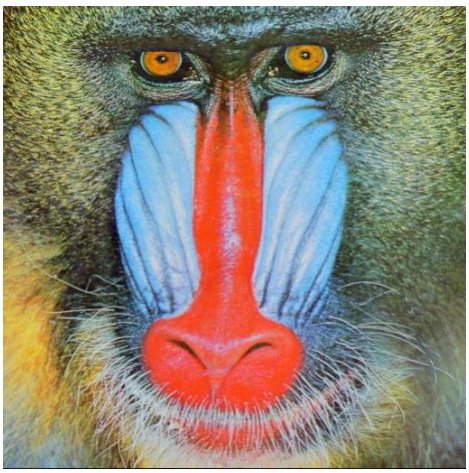

Figure 3 : baboon.png. input image for Gaussian Filtering function

- As it is seen above figure, I call the Gaussian Filtering function from main function by entering baboon.png image as input which is also shown above.

```
1  function I_new = lab2gaussfilt(img)
2
3  %Grayscaling Process
4  [row,col,ch] = size(img);
5  if(ch == 3)
6      img = rgb2gray(img);
7  end
8
9  %Doubling Process
10 img = double(img);
11
12 %kernel to be convolved with the image.
13 kernel = [1 4 7 4 1; 4 16 26 16 4; 7 26 41 26 7; 4 16 26 16 4; 1 4 7 4 1];
14 kernel = double(kernel);
15 kernel = kernel / 273;
16
17 I_new = zeros(size(img)); %create pure black new image with the size of input image.
18
19 %Convolution
20 k = 2;
21 for i = k+1 : (row -k)
22     for j = 3 : (col -k)
23         %subimage creation
24         sub_img = img(i-k:i+k,j-k:j+k);
25
26         %convolution
27         % M.*W where M is the subimage and W is the kernel
28         I_new(i,j) = sum(sum(sub_img.*kernel));
29     end
30 end
31
32 %Display
33 figure;
34 subplot(2,2,1), imshow(uint8(img)), title('Original Grayscale Image');
35 subplot(2,2,3), imhist(uint8(img)), title('Histogram of Original Grayscaled Image');
36 subplot(2,2,2), imshow(uint8(I_new)), title('Gaussian Smoothed Version');
37 subplot(2,2,4), imhist(uint8(I_new)), title('Histogram of Smoothed Image');
38
39 end
```

Figure 4 : Gaussian filtering function

- Above is the Gaussian Filtering function that was coded in lab. First of all, the function name is *lab2gaussfilt* (shown in 2), the input name is *img* (shown in 3) and the output name is *I_new* (shown in 1).
- Part 4 is for dealing with the image color. It gathers the values of row, column and channel at line 4 and checks whether the input image is colored or grayscaled image. If the input image is grayscaled already, does nothing; if it is colored, converts it to grayscale.
- Part 5 is converting grayscale image pixel values from integer to double so that we values will not lose any decimal values while applying mathematical operations.
- Part 6 is creating the 5x5 kernel that was given in figure 1 by hardcoding. Also, after creating the 5x5 matrix, we convert the values to double so when we divide each item in the matrix to 273, we will not lose any decimal.
- Part 7 is creation of new image with full of pure black as the input image size so that we will put filtered values of each pixel from input image to new image later.

- Part 8 is convolution part. Since our hardcoded matrix is 5x5 so W = 5, by k = (W-1)/2, k is 2 in our case. We iterate this bunch of code (row-1)*(col-1) so do operation on almost each pixel.
- In part 9, we create sub image from our input image with the same size of kernel (5x5) and also located at (i, j).
- Part 10 is the most important code in my opinion. This is the one which we apply convolution to sub image that was created previously with our kernel that was created before.
- At the end, in part 11, we display Original Grayscaled, Gaussian Smoothed Version of the input image and their histograms.

Outcome of Implementation:

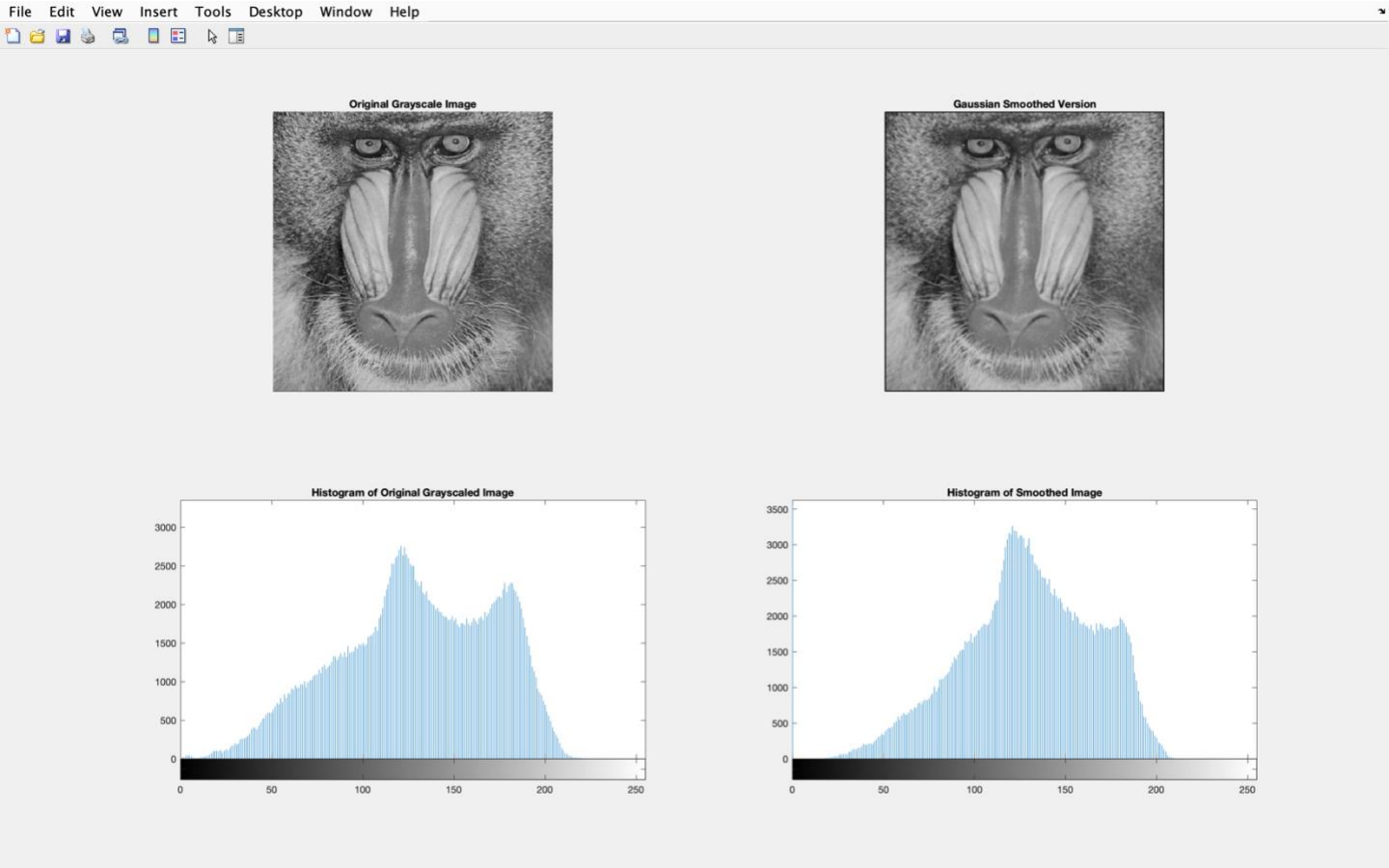

Figure 5 : OUTCOME of the gaussian filtering function

- As seen, Gaussian Smoothed Version of the image is more blurred than the original grayscaled image. Which means our code works properly. Also, I put the histogram tables just out of curiosity of mine.

Edge Detection

First Order Derivative Filters

1)Prewitt Operator

Description:

Prewitt filter is a discrete 2D first order derivative operation which can be applied with the following kernels

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

X Filter

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

Y Filter

Figure 6 : Prewitt Filters

In order to detect edges in grayscale image, the gradient image obtained by the horizontal and the vertical Prewitt operators is binarized by a threshold. The gradient image is calculated as

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

Implementation:

"Now write a function which takes an image and a threshold value as inputs and utilize 'Prewitt Filters' to return a binary image of detected edges."

```
6   %% Prewitt Operator
7   clear; close all; clc;
8   img2 = imread('Lab2 - Images/house.png');
9   lab2prewitt(img2, 100);
10
```

Figure 7 : the main function call for Prewitt Filtering function

Figure 8 : house.jpg. input image for Prewitt and Sobel Filter functions.

- I call lab2prewitt function in main function by entering hause.png image and 100 threshold value as inputs.

```
1  function I_new = lab2prewitt(img, th)
2
3  %Grayscaling Process
4  [row,col,ch] = size(img);
5  if(ch == 3)
6      img = rgb2gray(img);
7  end
8
9  %doubling
10 img = double(img);
11
12 %kernel to be convolved with the image
13 x_filter = [-1 0 1; -1 0 1; -1 0 1];
14 x_filter = double(x_filter);
15
16 y_filter = [-1 -1 -1; 0 0 0; 1 1 1];
17 y_filter = double(y_filter);
18
19 % convolutions
20 k = 1;
21 % horizontal gradient
22 I_new_h = zeros(size(img));
23 for i = k+1 : (row-k)
24     for j = k+1 : (col-k)
25         %subimage creation
26         sub_img_h = img(i-k:i+k , j-k:j+k);
27
28         %convolution
29         I_new_h(i,j) = sum(sum(sub_img_h .* x_filter));
30     end
31 end
```

Figure 9 : the first half of the Prewitt Operation function.

- This is the Prewitt Operation function whose name is *lab2preqitt* and takes *img* and *th* as inputs then returns *I_new* as output. What this function does is basically taking an arbitrary image and a threshold value, apply the kernel in figure6 one by one and create one binarized output image by utilizing the equation given above.
- Part 4 is for dealing with the image color. It gathers the values of row, column and channel at line 4 and checks whether the input image is colored or grayscaled image. If the input image is grayscaled already, does nothing; if it is colored, converts it to grayscale.
- Part 5 is converting grayscale image pixel values from integer to double so that we values will not lose any decimal values while applying mathematical operations.
- Part 6 does 2 things. It creates x_filter and y_filter then converts their pixel values to double to save the decimals in case of mathematical operations applied to those kernels. Note that kernels are given in figure6 above.
- Part 7 is related to x_filter convolution with input image. First, I set k as 1 since the window size of kernels are 3, so k = 1. Then in part 8, we create sub image with the size of kernel and convolved that sub image with x_filter then put that value to the pixel at (i,j) (at part 9) of new image that was created in line 22.

```
32
33  % vertical gradient
34  I_new_v = zeros(size(img));
35  for i = k+1 : (row-k)
36      for j = k+1 : (col-k)
37          %subimage creation
38          sub_img_v = img(i-k:i+k , j-k:j+k);
39
40          %convolution
41          I_new_v(i,j) = sum(sum(sub_img_v .* y_filter));
42      end
43  end
44
45  %GRADIENT
46  %creation of gradient image
47  I_new_grad = sqrt(I_new_v.^2 + I_new_h.^2);
48
49  %applying treshold to the gradient image
50  %Binarized Image
51  I_new = zeros(size(img));
52  for p = 1 : row
53      for q = 1 : col
54          if (I_new_grad(p,q) < th)
55              I_new(p,q) = 0;
56          else
57              I_new(p,q) = 255;
58          end
59      end
60  end
61
62
63  %Displays
64  figure;
65  subplot(3,1,1), imshow(uint8(img)), title('Original Grayscaled Image');
66  subplot(3,2,3), imshow(uint8(I_new_h)), title('X Filtered Image with Prewitt');
67  subplot(3,2,4), imshow(uint8(I_new_v)), title('Y Filtered Image with Prewitt');
68  subplot(3,2,5), imshow(uint8(I_new_grad)), title('Prewitt Gradient Image');
69  subplot(3,2,6), imshow(uint8(I_new)), title("Binarized Image with Prewitt");
70
71  end
72
```

Figure 10 : the other half of the Prewitt Operator function.

- In part 10, I do the same operation as part 7. First create new image as size of input image and create sub image with the size of the y_filter kernel. Then convolve that sub image with y_filter and put the value to the (i,j) index of newly created image in line 34.
- Part 11 is to combine those two newly created images called I_new_h and I_new_v in order to create new I_new_grad by using the equation above.
- Part 12 is dealing with binarizing I_new_grad which was created one previous step. While doing that we first create new image as size of input image. I use threshold value here in order to get rid of the weak edges. I do that by putting a level which the above pixel values will be pure white and below pure black so that edges are more precise and shown.
- Part 13 is to display all the images newly created and the original grayscaled image.

Output of Implementation:

Figure 11 : output images with th=50

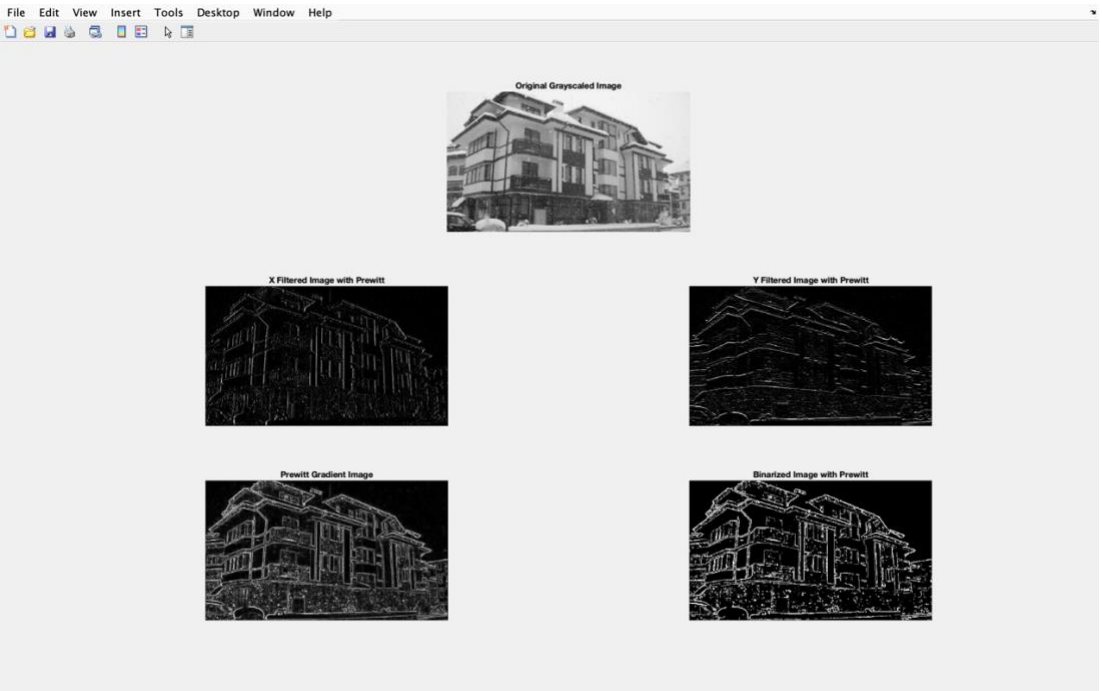

Figure 12 : output images with th=100

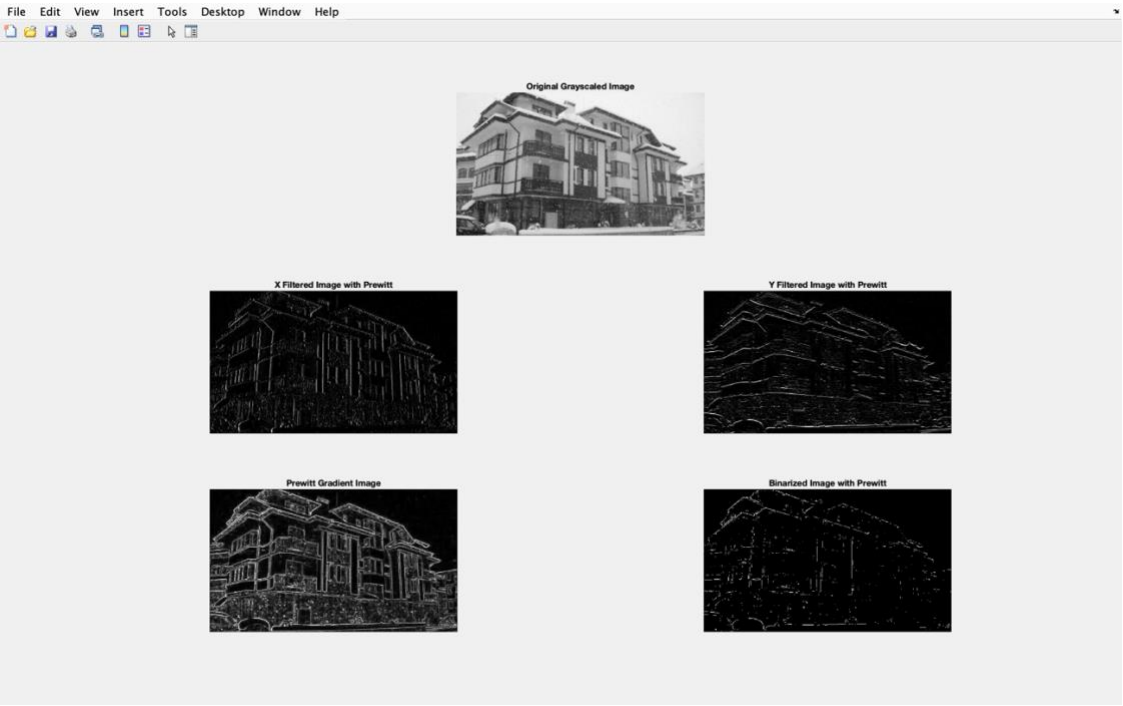

Figure 13 : output images with th=200

- The interpretation of mine for those outputs are those; when I put the threshold value too low, as in figure 11, the edges are over detected. On the other side, when I set the threshold value too high, I obtain over detected edges such as in figure 13. However, the threshold in the middle is way effective and proper to be used in these examples. So, I would prefer threshold value 100.

2)Sobel Operator

Description:

Sobel filter is a discrete 2D first order derivative operation which can be applied with the following kernels

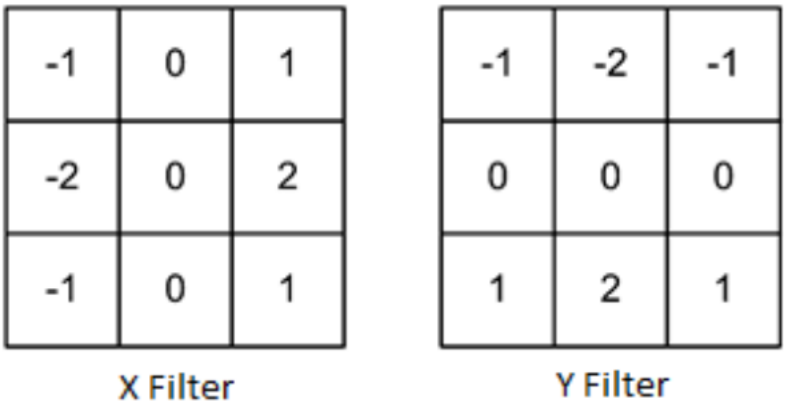

Figure 14 : Sobel Filter

In order to detect edges in grayscale image, the gradient image obtained by the horizontal and the vertical Sobel operators is binarized by a threshold. The gradient image is calculated as

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

Implementation:

"Now write a function which takes an image and a threshold value as inputs and utilize 'Sobel Filters' to return a binary image of detected edges. Function name should be 'bal2sobel.m'"

```
11    %% Sobel Operator
12    clear; close all; clc;
13    img3 = imread("Lab2 - Images/house.png");
14    lab2sobel(img3, 100);
15
```

Figure 15 : the main function call for Sobel Operation function

- I call lab2sobel function by entering house.png image and 100 as threshold value which will be used in order to limit on detecting edges.
- Note that I clear and close all previously written and opened data before this part so that no other unrelated data will cause any problem.

```
function I_new = lab2sobel(img, th)

%Grayscaling Process
[row,col,ch] = size(img);
if (ch == 3)
    img = rgb2gray(img);
end

%doubling
img = double(img);

%kernel to be convolved with the image
x_filter = [-1 0 1; -2 0 2; -1 0 1];
x_filter = double(x_filter);

y_filter = [-1 -2 -1; 0 0 0; 1 2 1];
y_filter = double(y_filter);

%convolutions
k=1;

%horizontal gradient
I_new_h = zeros(size(img));
for i = k+1 : (row-k)
    for j = k+1 : (col-k)
        %subimage
        sub_img_h = img(i-k:i+k, j-k:j+k);
        %convolution
        I_new_h(i,j) = sum(sum(sub_img_h .* x_filter));
    end
end

%vertical gradient
I_new_v = zeros(size(img));
for i = k+1 : (row-k)
    for j = k+1 : (col-k)
        %subimage
        sub_img_v = img(i-k:i+k, j-k:j+k);

        %convolution
        I_new_v(i,j) = sum(sum(sub_img_v .* y_filter));
    end
end
```

Figure 16 : First half of the lab2sobel.m function.

```
44
45
46  %GRADIENT
47  I_new_grad = sqrt (I_new_v.^2 + I_new_h.^2);
48
49  %Binarize
50  I_new = zeros(size(img));
51  for i = 1:row
52      for j = 1:col
53          if (I_new_grad(i,j) < th)
54              I_new(i,j) = 0;
55          else
56              I_new(i,j) = 255;
57          end
58      end
59  end
60
61  %Display
62  figure;
63  subplot(3,1,1), imshow(uint8(img)), title('Original Grayscaled Image');
64  subplot(3,2,3), imshow(uint8(I_new_h)), title('X Filtered Image with Sobel');
65  subplot(3,2,4), imshow(uint8(I_new_v)), title('Y Filtered Image with Sobel');
66  subplot(3,2,5), imshow(uint8(I_new_grad)), title('Sobel Gradient Image with Sobel');
67  subplot(3,2,6), imshow(uint8(I_new)), title("Binarized Image with Sobel");
68  end
```

Figure 17 : Second half of the lab2sobel.m function.

- This function takes *img* and *th* values as inputs which correspond the input image and threshold value respectively and return *I_new* as output after the entire processes. (shown in 1, 2, 3).
- Part 4 is for dealing with the image color. It gathers the values of row, column and channel at line 4 and checks whether the input image is colored or grayscaled image. If the input image is grayscaled already, does nothing; if it is colored, converts it to grayscale.
- Part 5 is converting grayscale image pixel values from integer to double so that we values will not lose any decimal values while applying mathematical operations.
- The first 2 line of part 6 is to create X Sobel filter and the other 2 lines of part 6 is to create Y Sobel Filter. As given in figure 14, I have hardcoded the filter by my hand and converted their values to double so that when we perform any mathematical operation to or with them, will not be any loss of decimals.
- In part 7, k=1 since window size is 3x3 on those kernels and k = (W-1)/2.
- Part 8 is to calculate horizontal gradient. First, it creates new image called *I_new_h* with the size of original image and sets its pixel values as pure black. Then we iterate the code (col*row) times by creating sub image centered at (i,j) of the input image with window size 3 and convolve it with the X Sobel Filter.
- In part 9 we do almost the same thing with the only difference by convolving the sub image with Y filter of Sobel.
- Part 10 is to combine horizontal and vertical gradients by applying the above formula that was given in the description part. At the end of this step, we have the combined gradient of the original image.

- However, since we are asked to give binarized version of gradient of the original image, we perform part 11 in order to apply threshold to the gradient values so the values under the threshold value will be turned to 0 and above will be turned to 255.
- Finally, we display all the outcomes of each steps one by one.

Output of Implementation:

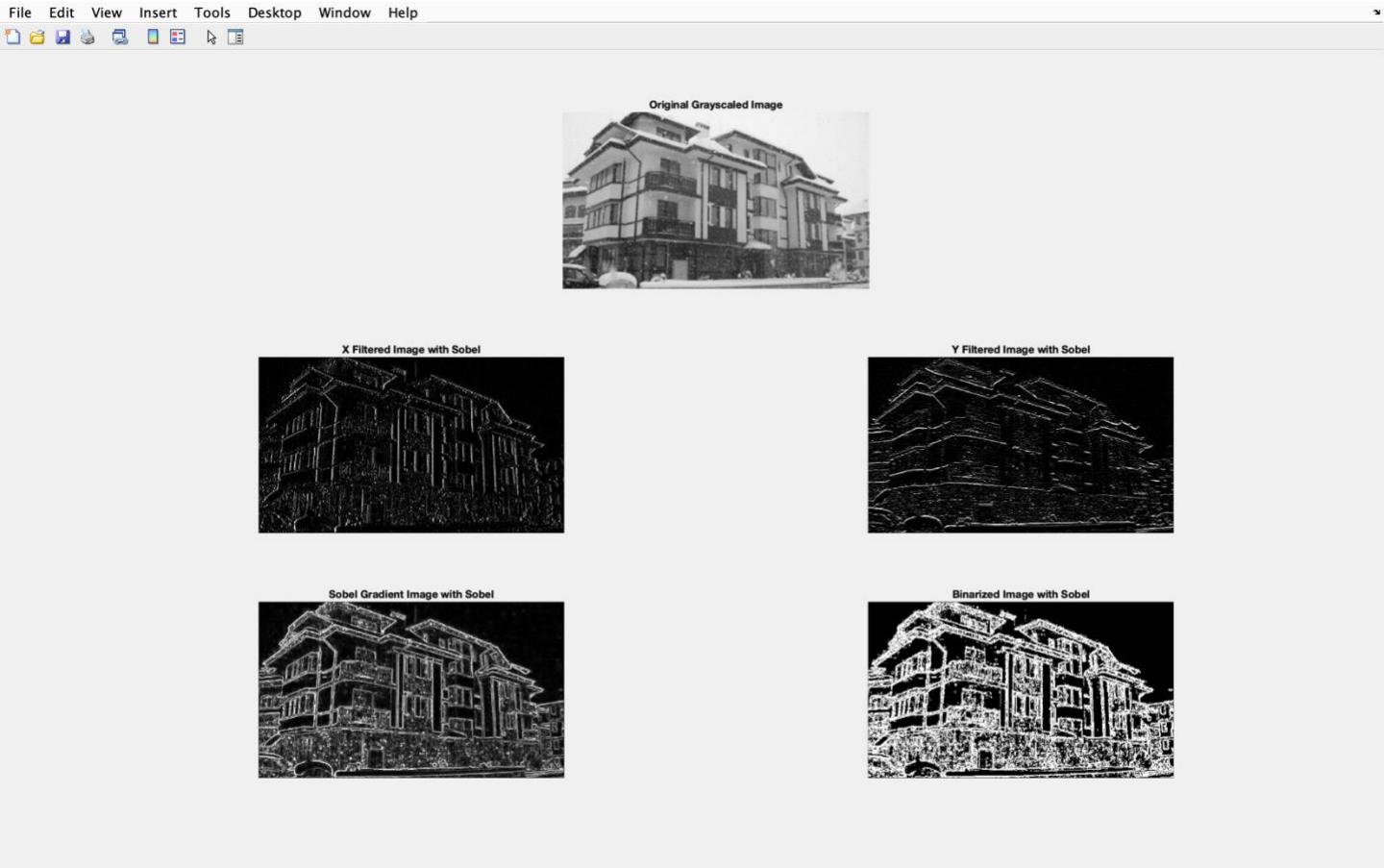

Figure 18 : th = 50

Figure 19 : th = 100

Figure 20 : th = 200

- As it is seen from the different outcomes according to the th values that was entered to the function; th = 50 is not filtering almost anything since the threshold value is too low. However, when we increase the threshold value to the 100, we easily are able to identify the edges. But, if we over increase the threshold value like 200, we lose some edges that was a real edge and needed for detection. Thus, th = 100 seems like the most optimal value within those 3 values.

Second Order Derivative Filter

Laplacian of Gaussian Smoothed Image

Description: We have already detected edges with first order derivative methods such as Sobel and Prewitt Operations. Now, we will utilize Laplacian of Gaussian (LoG) filter in order to detect the edges and see the difference between first order and second order derivative edge detectors. The important thing that we should keep in mind about LoG filter is that it may detect noises as well as edges of an image. So that's why we must smooth the image first by using Gaussian Filter. Applying the Laplacian to a Gaussian filtered image can be done in a single step of convolution with the kernel of ;

0	1	0
1	-4	1
0	1	0

Figure 21 : LoG Kernel

The main difference between first order edge detectors and LoG is that zero crossing points represent the edge pixel rather than pure white lines. Thus, extract the gradient profile from a line segment of the LoG filtered image belonging to an edge region and investigate the zero-crossing behavior.

Implementation:

"Write a function which takes an arbitrary image as the input and utilize 'Laplacian of Gaussian' to return the LoG filtered image. Show the LoG filtered image along with a sample gradient profile from a pre-selected line segment of the filtered image. Name the function as 'lab2log.m'"

```
16    %% Laplacian of Gaussian Smoothed Image
17    clear; close all; clc;
18    img4 = imread("Lab2 - Images/Object_contours.jpg");
19    lab2log(img4);
```

Figure 22 : the LoG call on main function

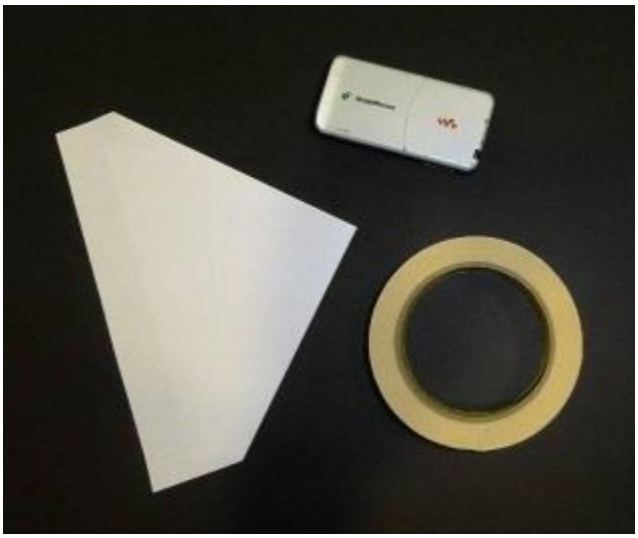

Figure 23 : Object_contours.jpg

- Notice that I called the lab2log function to ‘Object_contours.jpg’ image that is shown above in figure 23. Also, cleared and closed all the data previously used and altered so that the implementation will be isolated from outsider implementations.

```
1  function I_new = lab2log(img)
2
3  %Grayscaling Process
4  [row,col,ch] = size(img);
5  if (ch == 3)
6      img = rgb2gray(img);
7  end
8
9  %doubling
10 img = double(img);
11
12 %kernel to be convolved with the image
13 kernel = [0 1 0; 1 -4 1; 0 1 0];
14
15 %convolutions
16 k=1;
17 I_new = zeros(size(img));
18 for i = k+1 : (row-k)
19     for j = k+1 : (col-k)
20         %subimage
21         sub_img_v = img(i-k:i+k, j-k:j+k);
22
23         %convolution
24         I_new(i,j) = sum(sum(sub_img_v .* kernel));
25     end
26 end
27
28
29 %Display
30 figure;
31 subplot(1,3,1), imshow(uint8(img)), title('Original Grayscaled Image');
32 %subplot(1,3,2), imshow(uint8(I_new)), title("LoG applied Image");
33 subplot(1,3,2), imshow(I_new, []), title("LoG applied Image");
34 subplot(1,3,3), plot(I_new(130, 30:60));
35 %disp(Im2(190:200,320:330));
36
37 end
```

Figure 24 : The implementation of lab2log function

- This function which is called ***lab2log*** is taking an arbitrary image as ***img*** and returns another image called ***I_new*** by performing some operations such as filtering and revealing the edges. (shown in 1,2,3)
- Part 4 is for dealing with the image color. It gathers the values of row, column and channel at line 4 and checks whether the input image is colored or grayscaled image. If the input image is grayscaled already, does nothing; if it is colored, converts it to grayscale.

- Part 5 is converting grayscale image pixel values from integer to double so that we values will not lose any decimal values while applying mathematical operations.
- Part 6 is to create the kernel (figure 21) that will be used to filter each of every pixel values of input image.
- Part 7 is the most integral part of the implementation. First, k = 1 since the kernel that is going to be used has W=3. Then, we create ***I_new*** with the size of input image by full of zeros, meaning pure black color to each pixel. Then we iterate that code for col*row times by creating sub image centered at (i,j) of input image and with window size of kernel, 3. Then it convolves that sub image with kernel so that for each pixel of new image is filtered and mapped to ***I_new***. After row*col iteration, we have LoG filtered new image.
- Part 8 is to display all the images that has been created within the processes. Also, LoG applied image is displayed as gray colored for ease of seeing the edges. The final display is prechosen plot to show the gradient profile of edge on a specific area of the output image.

Output of Implementation:

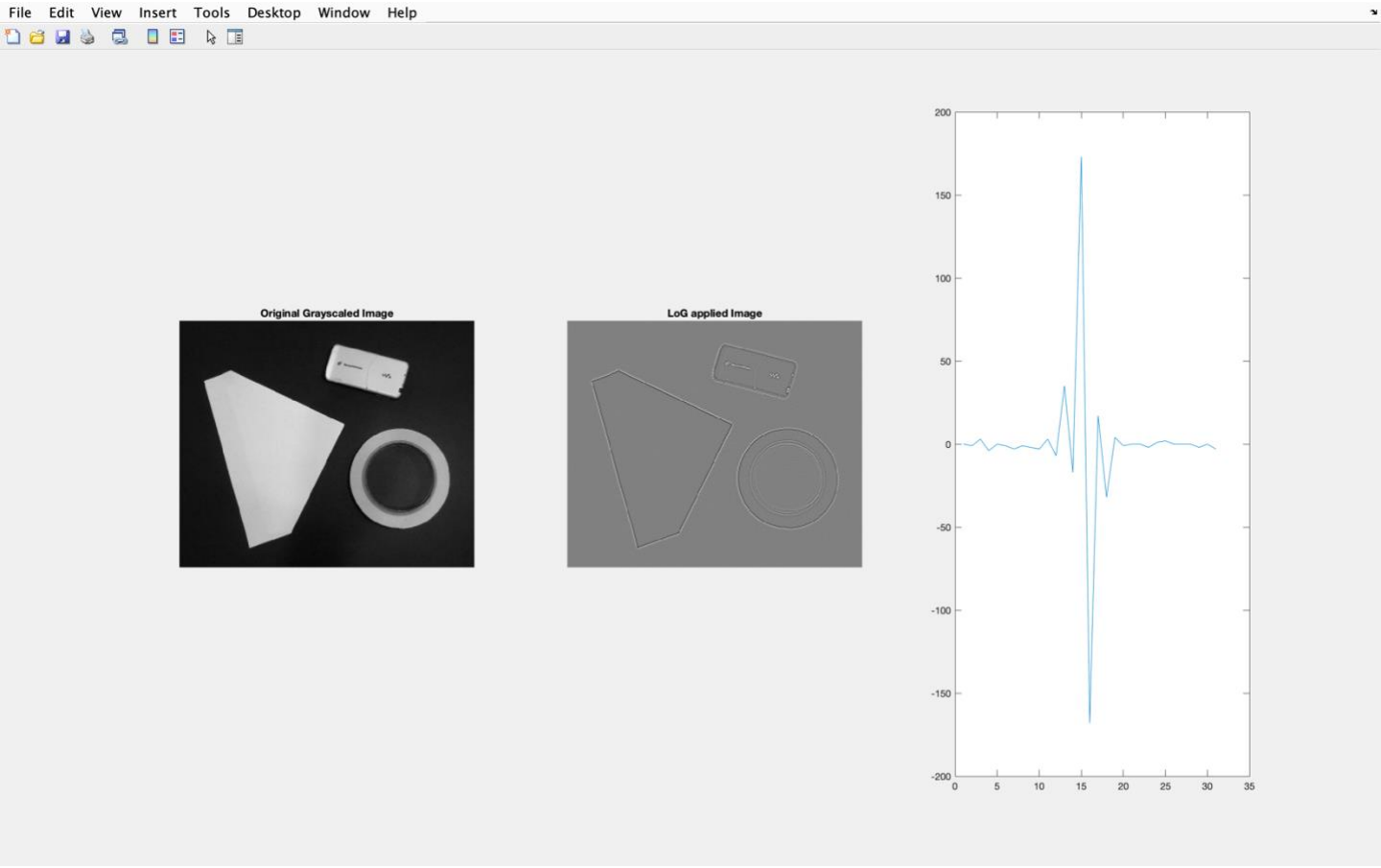

Figure 25 : from left to right; original image, edge detected image and the gradient profile between prechosen pixels.

- Here, we can see the edges are perfectly detected and the gradient profile of the pixels between (130,30) and (130,60). The gradient profile implies that there is a zero crossing which means edge because of our implementation. And notice that before and after the zero crossing, there is a sign change so that we make sure it is an edge.

------END OF THE POST LAB REPORT------

Thanks for reading

Goktug