NANYANG TECHNOLOGICAL UNIVERSITY

CZ4003 COMPUTER VISION

LAB Deliverables

Assignment 2

Edge Detection

Hough Transform

3D Stereo

Sam Jian Shen (U1821296L)

Date of Submission: 17th of November 2020

School of Computer Science and Engineering

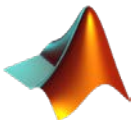
Nanyang Technological University

Contents

Setup and Execute

This section details can be found in README.md

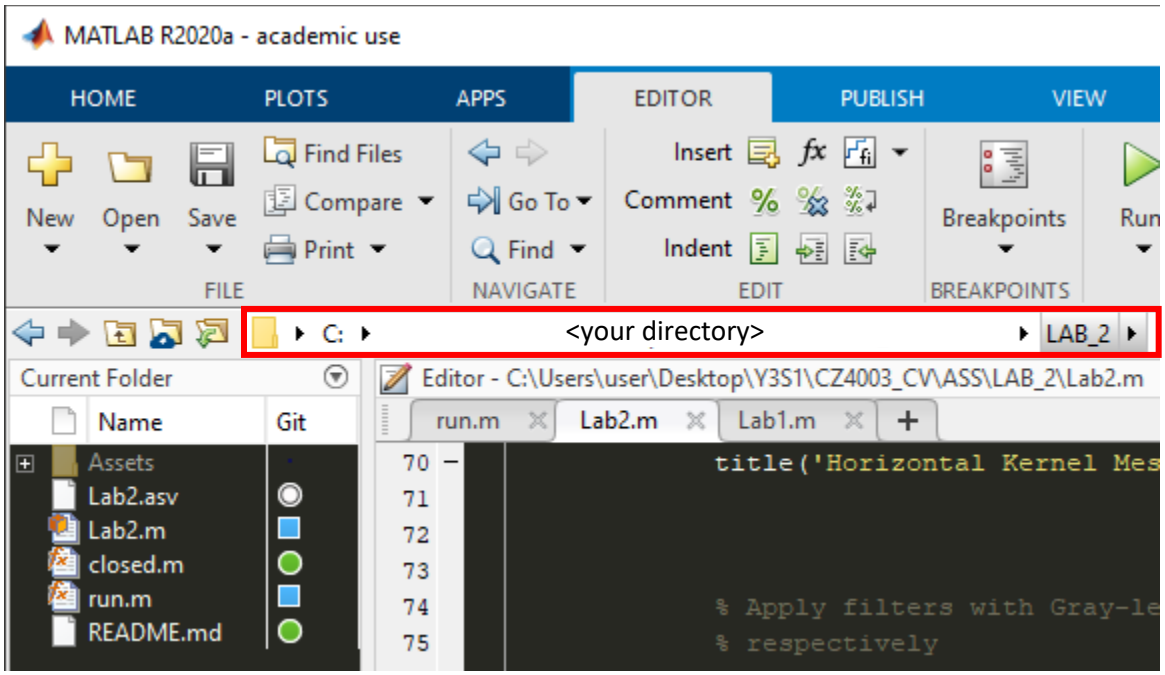
Pre-Requisite



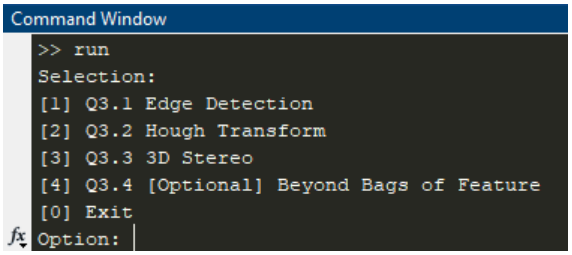
- Recommended MATLAB version **R2020a** and above installed.
- Installed **Image Processing Toolbox** software package from MATLAB Extension

How to Use?

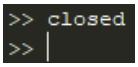
1. Run **MATLAB** program
2. Please ensure the current directory is at **LAB_2** file



3. At the 'Command Window' type '**run**' and enter



   a. The Command Window various **options** should be visible by then
4. **Follow the prompt instruction(s)** in Command Window accordingly
5. [Optional] At 'Command Window' type '**closed**' to close all opened window



2.1 Edge Detection

a) Display ‘macritichie.jpg’ in Grayscale

Code:

```
URL = 'assets\macritchie.jpg';
```

- Set directory path

```
Pc = imread(URL);
```

- Read image into 3 dimension arrays with value range 0-255

```
Name         Size                Bytes  Class

Pc         290x358x3            311460  uint8
```

```
img_Grayscale = rgb2gray(img_Origin);
```

- Convert RGB image to gray-level image

```
imshow(img_Grayscale);
```

- Display grayscale image

Output:

Note: the left side is the raw image and the right side is the gray-level image

b) Create 3x3 horizontal and Vertical Masks and Filter Image using Sobel masks

Given Sobel masks:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

X-Direction Y-Direction

Code:

```
kernel_x = [-1 0 1; -2, 0 2; -1 0 1];
kernel_y = [-1 -2 -1; 0 0 0; 1 2 1];
```

- Define Sobel mask

```
img_GreyScale = evalin('base','img_GreyScale');
```

- Retrieve gray image from workspace

```
img_edge_x = conv2(img_GreyScale, kernel_x,'same');
img_edge_y = conv2(img_GreyScale, kernel_y,'same');
```

- Filter them with conv2, this result in both the vertical edge and the horizontal edge of the image
- To ensure padding result was not included we use properties called ‘same’ to ensure the image size is not compromise.

```
img_edge = img_edge_x + img_edge_y;
```

- Combine both of the edges by sum them up together, this will result in the resultant edge image

```
% Normalize to uint8 0-255 range
img_edge_x_uint8 = uint8(img_edge_x);
img_edge_y_uint8 = uint8(img_edge_y);
img_edge_uint8 = uint8(img_edge);
```

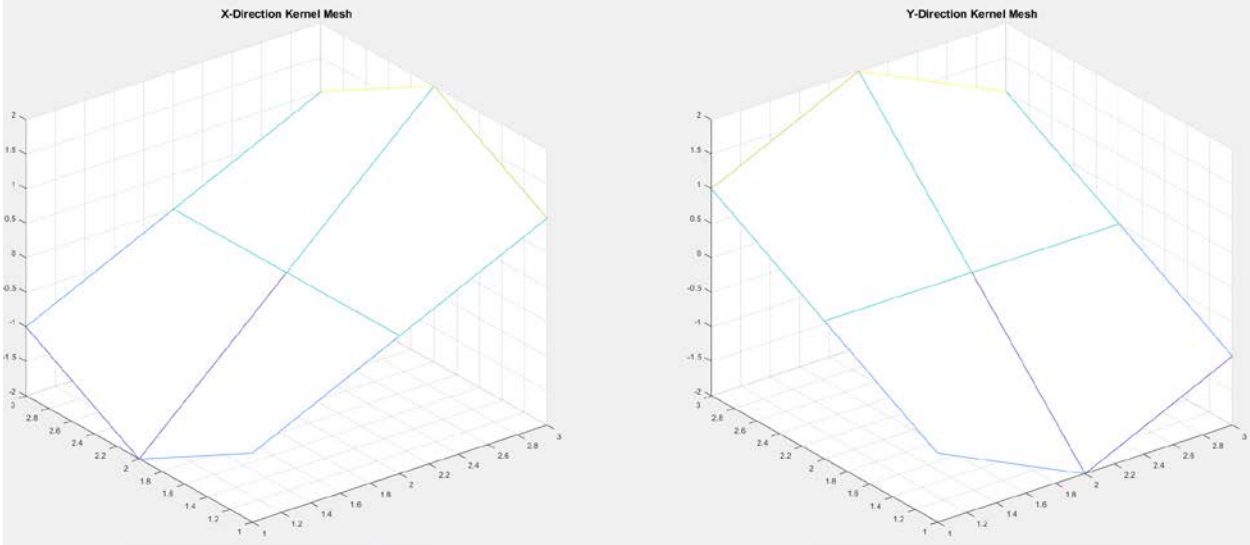
- Normalize to 0 to 255 to display the image properly

```
% Display horizontal edge
subplot(1,3,1), imshow(img_edge_y_uint8);
title('Y-Direction Sobel Kernel','FontSize', 14);
% Display vertical edge
subplot(1,3,2), imshow(img_edge_x_uint8);
title('X-Direction Sobel Kernel','FontSize', 14);
% Display all edge
subplot(1,3,3), imshow(img_edge_uint8);
title('Combine','FontSize', 14);
```

- Display the horizontal, vertical, and combine edges of images respectively

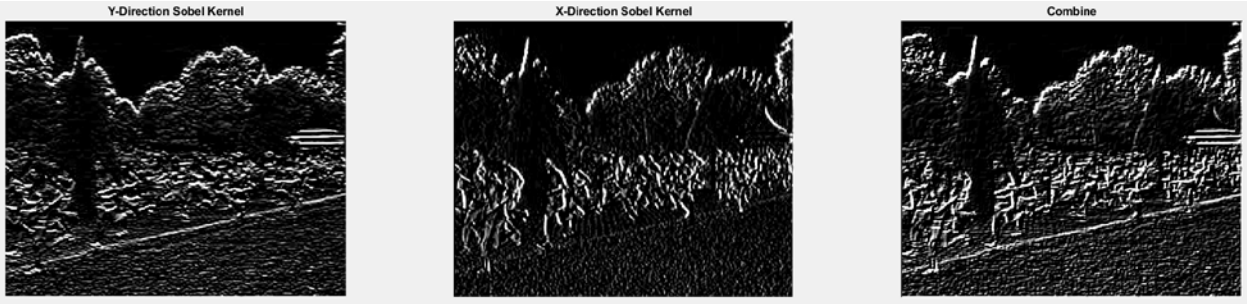
Output:

Sobel Kernel Mesh View



I can see that the weights are given more depend on the direction of the edge especially the center left-right or top-down of the Sobel Kernel

Filtered Images

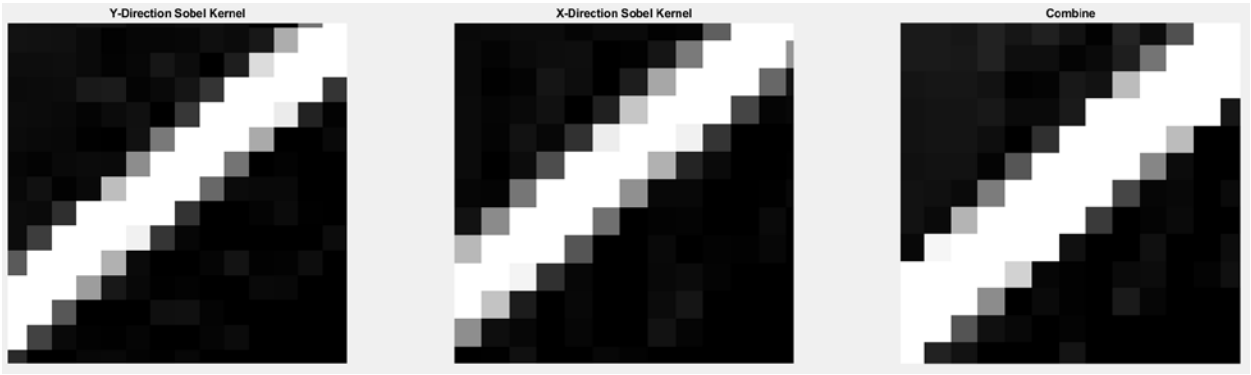


From the image, we can tell on the left image which is the Y-Direction Sobel kernel can detect horizontal edges, the middle image which is X-Direction Sobel kernel can detect vertical edges, the right image is a combination of both edges together to show all possible edges in the image.

What happens to an edge which is not strictly vertical nor horizontal?

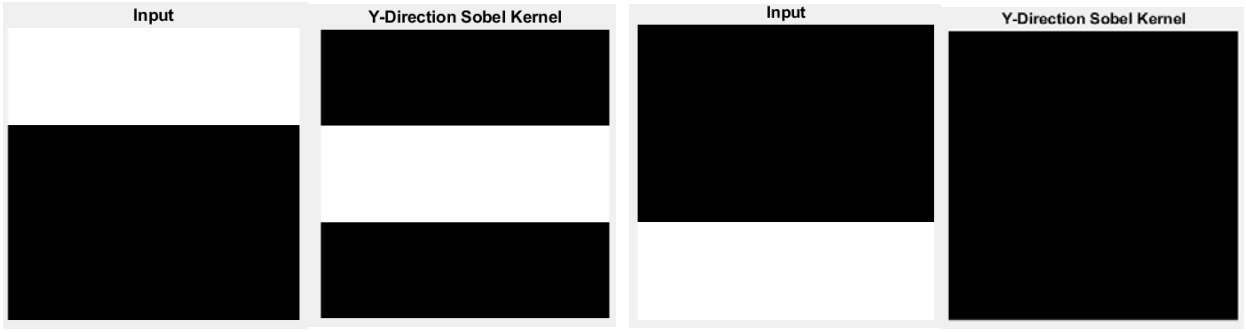
Base on the given image, it may be hard to tell since it is a low-resolution image with many curvatures. To answer better this question. I use a diagonal object image. In this case, I am using 'chopstick.jpg' instead.

From this, I can prove that even though the edge is not vertical nor horizontal we can still able to detect the edge to a certain extent. The main reason is that a diagonal is composed of a vertical and horizontal staircase-like pattern. To illustrate this better. Below is the zoom-in of the diagonal line of the object.

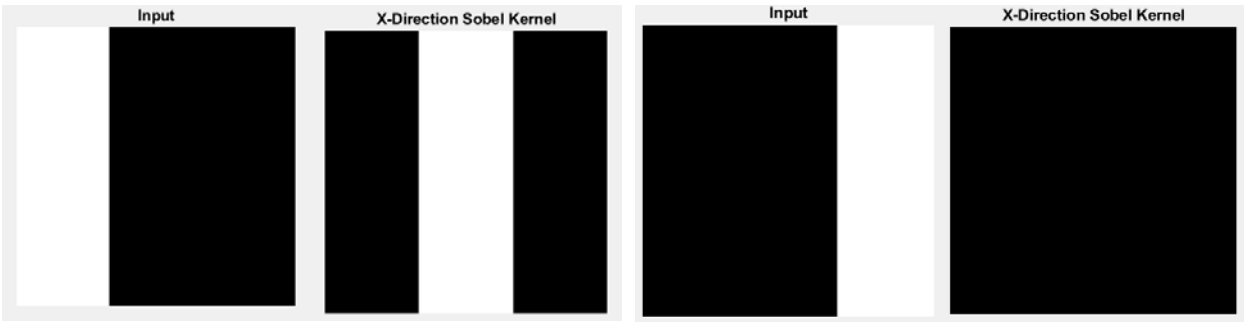


However, there is a certain part of the chopstick's lack of visible edges, the reason is that when I scale down the image to 3 by 3 pixels. It can detect a vertical/horizontal depending on the direction of the change in the gray-level intensity. This may explain the inconsistent edges that appear in 'macritichie.jpg' especially in diagonal curves of the roof on the right-hand side of the image, where it appears on one side but not on the other side when detecting vertical edge. To illustrate better below is the 3 by 3 input and see what happens when the Sobel mask is used.

Horizontal Edge Detection



Vertical Edge Detection



Notice that depending on where the direction of the edge is detected. The top part of the horizontal edge, the edge is detected during the bottom-up edge, not for the top-down edge. Likewise, for the vertical edge, the edge is detected during right-left edges, not for the left-right edge. Why does it appear like this? To understand better I need to delve deeper, to see the math process of the convolution.

Horizontal Edge Detection

Input			Before normalize Y-Direction Edge			After normalize Y-Direction Edge		
255	255	255	0	0	0	0	0	0
0	0	0	765	1020	765	255	255	255
0	0	0	0	0	0	0	0	0

Input			Before normalize Y-Direction Edge			After normalize Y-Direction Edge		
0	0	0	0	0	0	0	0	0
0	0	0	-765	-1020	-765	0	0	0
255	255	255	0	0	0	0	0	0

From here, I can clearly understand why the edge behaves strangely is due to negative values are not consider when to normalize the image. This explains the reason for missing edges. This applies similarly to the vertical edge as well.

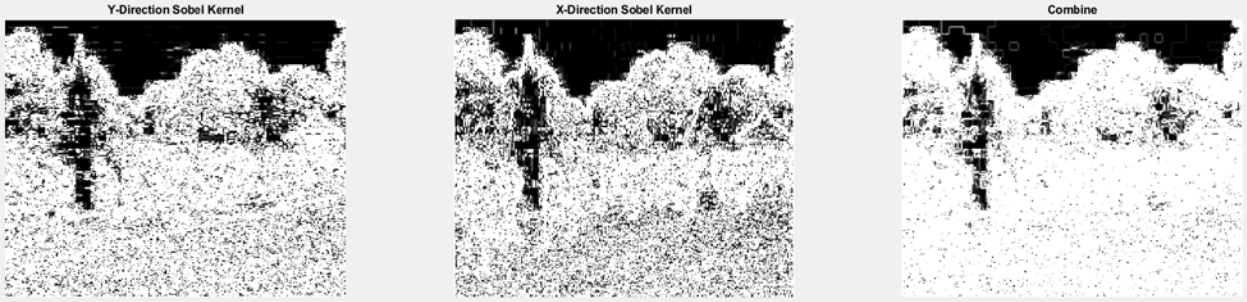
c) Combine Edge Image with squaring

Code:

```
img_edge_x = conv2(img_GreyScale, kernel_x,'same');
img_edge_y = conv2(img_GreyScale, kernel_y,'same');
img_edge = img_edge_x + img_edge_y;
```

Squared after convolution, to display both vertical and horizontal edge simply sum them up

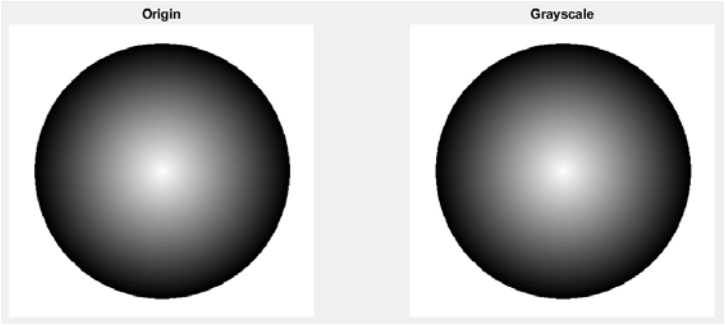
Output:



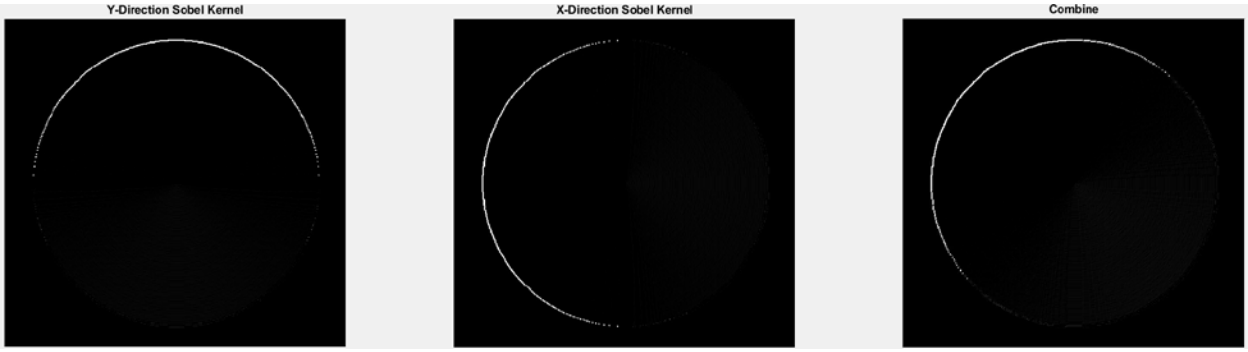
The image appears to be detecting a lot of edges than before. To understand better what is happening I use gradient gray image 'circular_gradient_gray.jpg'. Why I choose this, is because circle provide a universal understanding of how the filter work in a way that does not discriminate the direction of the edge

Why square it?

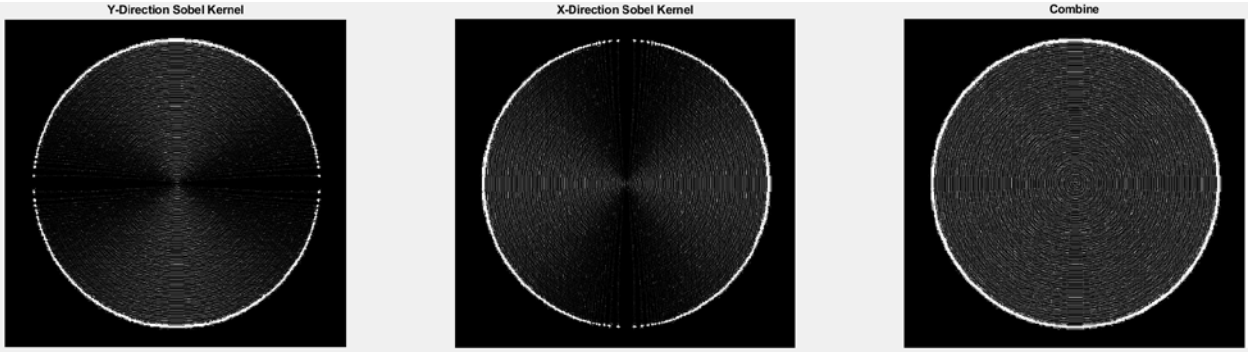
Origins



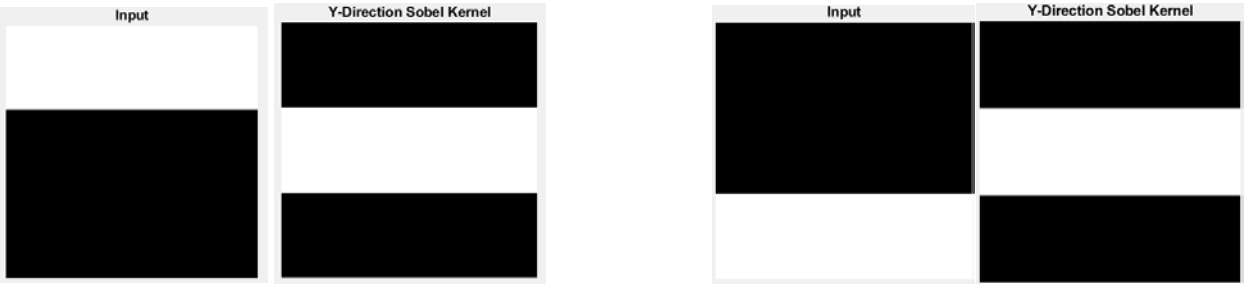
Without Squared



With Squared



Based on observation, I can see that image without a square strong edge (great change of intensity) was shown and in one direction due to the negative value not consider as mentioned earlier. On the other hand, the squared image switches all negative to positive, this results in all possible changes in intensity to be shown regardless of which direction. The brighter indicates a strong change while the dimmer indicates a small change in intensity.




```
Input
    0     0     0
    0     0     0
  255   255   255

Before normalize Y-Direction Edge
         0         0         0
    585225   1040400    585225
         0         0         0

After normalize Y-Direction Edge
    0     0     0
  255   255   255
    0     0     0
```

The above has shown how it has managed to detect the edge regardless of which direction the edge is located. This concludes the reason why 'macritichie.jpg' appear in this manner because there is so much change of intensity in the image result in all kind of edges been shown. A way to select the edge I want to see if by threshold edges.

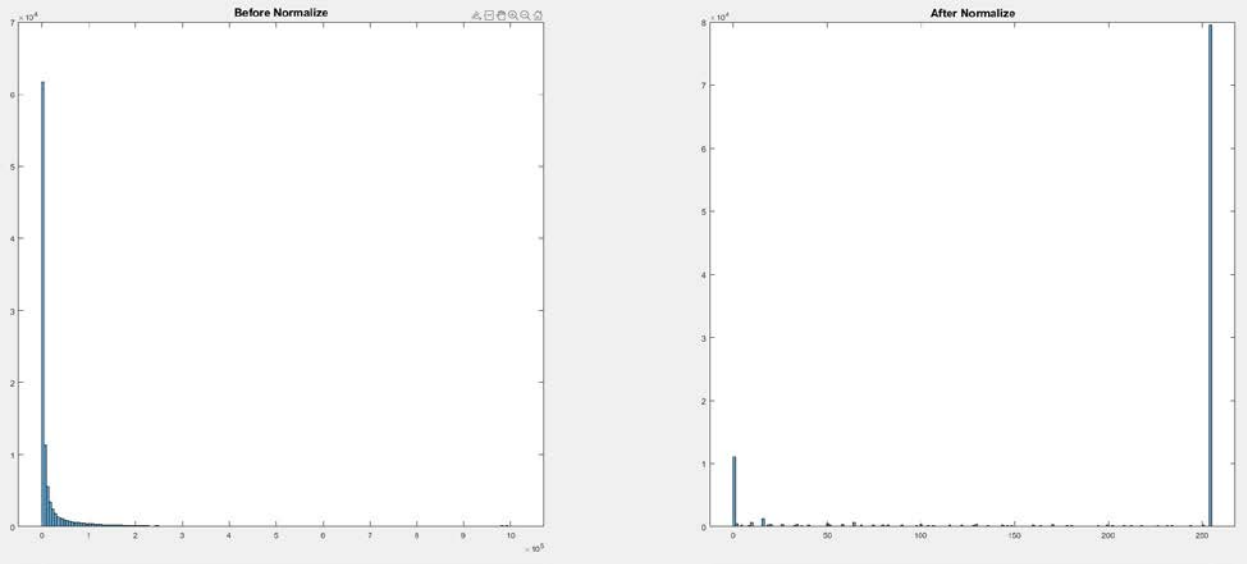
d) Threshold hold image

To threshold the image we need to understand the values we are considering.

```
Image Min Value before normalize:
    0

Image Max Value before normalize:
    1019592
```

As expected the value to be huge due to the power factor. Thus, our threshold requirement would be within this range.

Squared Image before and after normalizing



To analysis how is the image intensity being distributed we analyze the histogram of the image intensity value before and after normalize? Based on observation, I can see outlier values have from the left-hand side has greatly influenced the outcome of the normalization process on the right-hand side. This explains the reason why there is excessive brightness in the resultant image after squared. To mitigate the power, I square root the image which produces the result below.

Code:

```
img_edge_x = sqrt(conv2(img_GreyScale, kernel_x,'same').^2);
img_edge_y = sqrt(conv2(img_GreyScale, kernel_y,'same').^2);
img_edge = img_edge_x + img_edge_y;
```

- sqrt() to square root the value

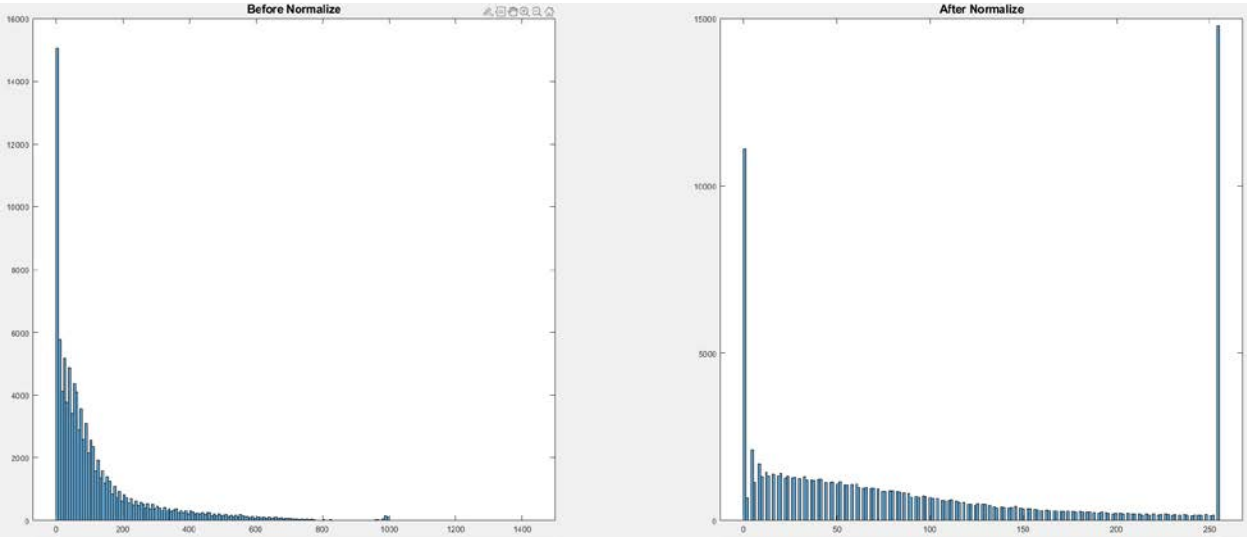
```
subplot(1,2,1), histogram(img_edge,200);
title('Before Normalize','FontSize', 14);
subplot(1,2,2), histogram(img_edge_uint8,200);
title('After Normalize','FontSize', 14);
```

- Display histogram with 200bins size

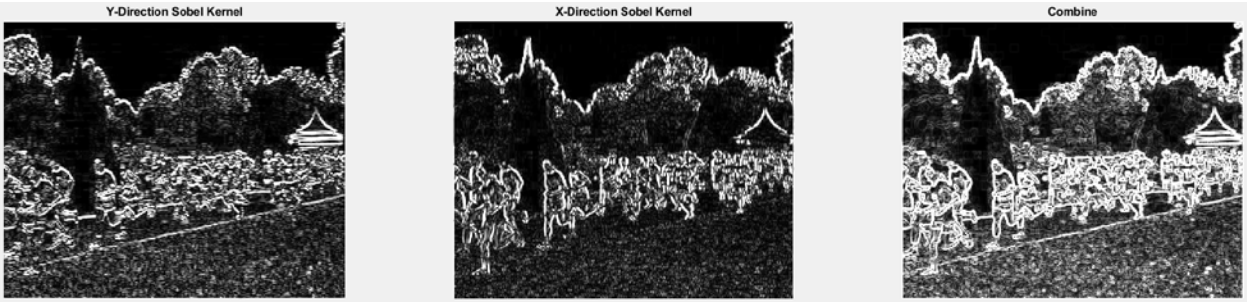
The rest of the code is display image output as mentioned above.

Output:

Histogram Before and After normalize



Filtered Image



It is a much better and observable range.

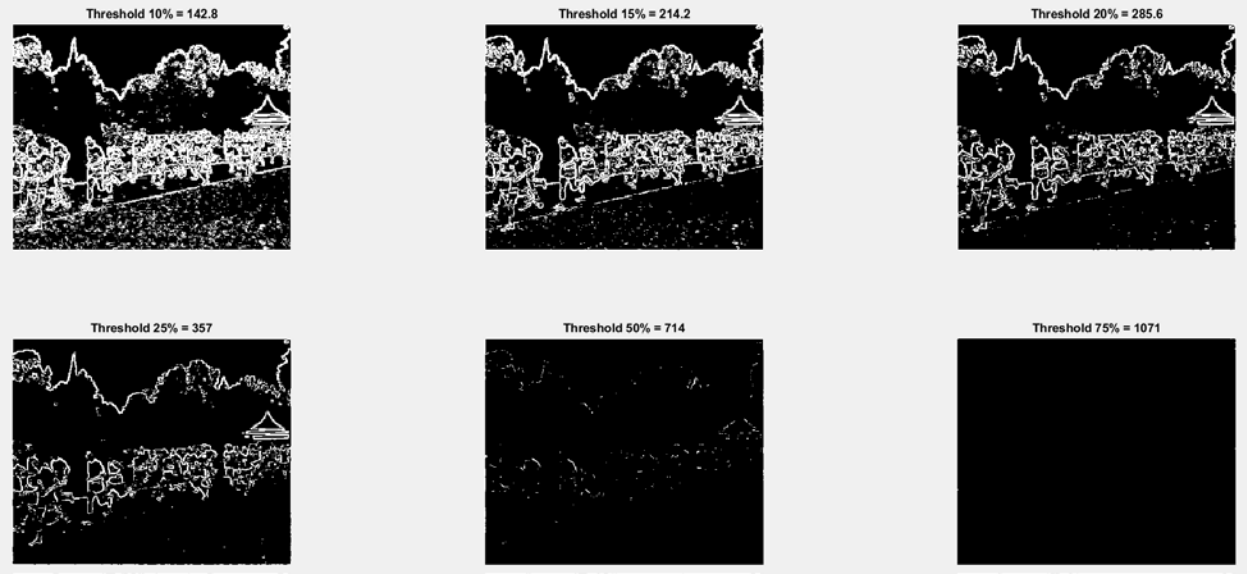
I go for thresholding base on the percentile of the image intensity range. The result is shown below.

Code:

```
% Apply threshold and return a binary image base on percentile
% of the image intensity
t10 = min_t + (range/100)*10;
t15 = min_t + (range/100)*15;
t20 = min_t + (range/100)*20;
t25 = min_t + (range/100)*25;
t50 = min_t + (range/100)*50;
t75 = min_t + (range/100)*75;

image_threshold1 = image > t10;
image_threshold2 = image > t15;
image_threshold3 = image > t20;
image_threshold4 = image > t25;
image_threshold5 = image > t50;
image_threshold6 = image > t75;

% Figure Settings
figure('Name','Threshold Image','NumberTitle','off') % change window name
set(gcf, 'Position', get(0, 'Screensize')); % set figure fullscreen
% Display image when threshold with different range of values
subplot(2,3,1), imshow(image_threshold1);
title('Threshold 10% = ' + string(t10),'FontSize', 14);
subplot(2,3,2), imshow(image_threshold2);
title('Threshold 15% = ' + string(t15),'FontSize', 14);
subplot(2,3,3), imshow(image_threshold3);
title('Threshold 20% = ' + string(t20),'FontSize', 14);
subplot(2,3,4), imshow(image_threshold4);
title('Threshold 25% = ' + string(t25),'FontSize', 14);
subplot(2,3,5), imshow(image_threshold5);
title('Threshold 50% = ' + string(t50),'FontSize', 14);
subplot(2,3,6), imshow(image_threshold6);
title('Threshold 75% = ' + string(t75),'FontSize', 14);
```

- First, find all the value for the threshold base on range and percentile
- Second, apply a threshold such that it returns min and max value which is 0 and 1428 in this case. The reason I do need further normalize because is the same 0 and 255 since it is a binary image, any value beyond 255 is always maxed intensity.
- Note, the 'image' is before normalizing.

Output:



Base on observation, I can see that having a high threshold, we barely able to get the edge we want. While having a low threshold we may get an unwanted intensity (noise) which is not an edge. It is hard to identify the optimal threshold value, the only way to find out is to go through the tedious trial and error work which is not the most efficient way to do. Imagine if there are n number of images, the better algorithm is needed for such complexity.

Advantage and Disadvantage of using different threshold

	High Threshold	Low Threshold	Optimal Threshold
Advantage	Robust to Noise	More Edges	Best possible Edge
Disadvantage	Fewer and missing Edges	Noise Detected	Likely to have noise and missing edges existing at the same time.

Therefore, there is no best threshold, there is always a trade-off either for noise or edges

e) Recompute the edge using Canny Edge Detection

Given

- lower threshold (tl) = 0.04

- higher threshold (th) = 0.1

- sigma = 1

Code:

After reading the origin image and converted to grayscale

```
th = 0.1;
tl = 0.04;
```

- Declare the threshold given value

```
for sigma = 1:1:5
```

- Declare sigma value to be tested later

```
image_canny = edge(img_GreyScale,'canny',[tl th],sigma);
```

- Invoke canny edge function with all it's parameters

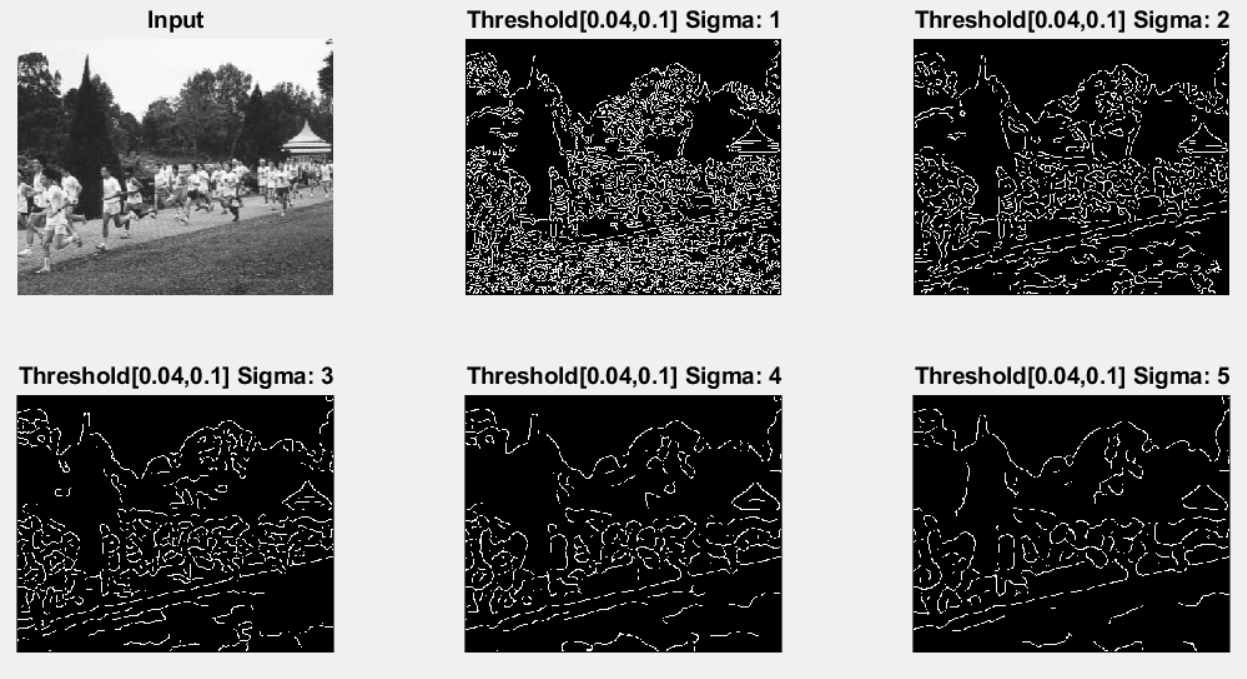
e.i) Test different sigma value between 1.0 to 5.0

Code:

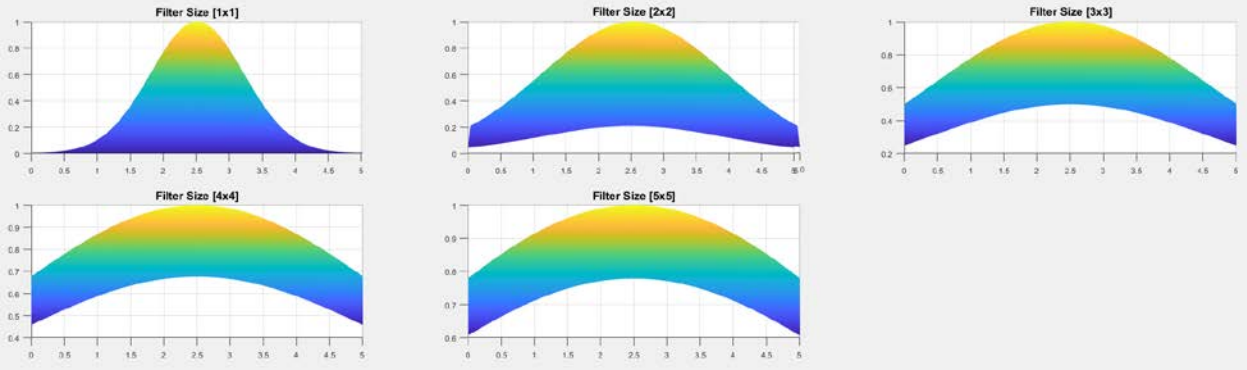
```
for sigma = 1:1:5
    image_canny = edge(img_GreyScale,'canny',[tl th],sigma);
    % Display Image after canny edge detection algorithm
    subplot(3,3,sigma+1), imshow(image_canny);
    title('Threshold[' + string(tl) + ',' + string(th)+ '] Sigma: ' + string(sigma),'FontSize', 14);
end
```

- A for loop that increment by 1, sigma value from 1 to 5.

Output:



Notice as sigma goes higher, less edge and noise detected vice versa. The reason for such a case was due to the Gaussian filter process in the Canny Edge Detection Algorithm. Base on edge function code located in 'edge.m'. It stated that the size of the filter is chosen automatically, based on Sigma value. To illustrate better I compute the gaussian distribution base where the filter size is base on the sigma value.



In this case, we fixed the filter size of 5 to study the distribution of the weight compared to different sigma values. Base on observation, the higher the sigma value wider and gentle the gradient of Gaussian distribution, while the lower the sigma value the less wide and steeper the gradient of Gaussian distribution. Since the size is proportional to the sigma, the larger the sigma value the more weightage is given to the surrounding. This means sharper edges are more blurred out which result in loss of details, the noise would be suppressed and removed as the sigma value increase. Thus, a smoother image. This will affect the later part of the canny edge detecting process of the algorithm. Such as determine if the feature is true edgel, true edge, noise, noise edgel, or noise edge.

How sigma is suitable?

Noise edgel removal

- In our context, the higher value of sigma in the range of 3 to 5 is more likely noise edgel tolerance. For sigma value 1, 2 noise edgel is still visible result in unwanted edge detected.

Location accuracy of edgels

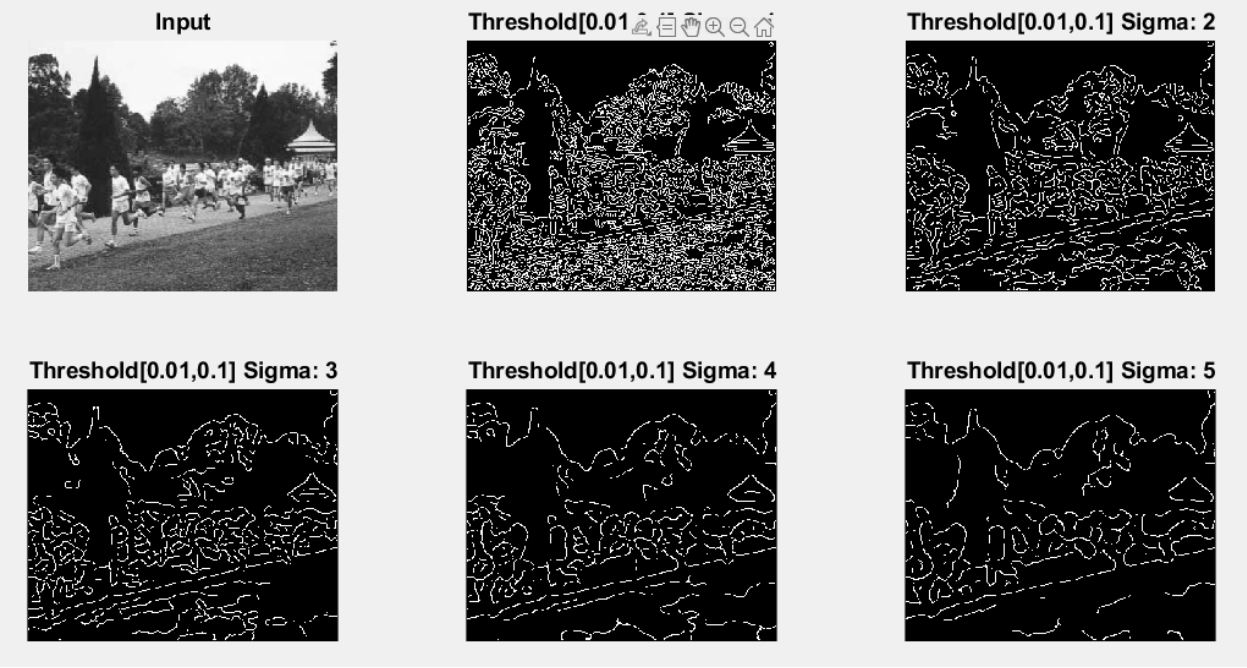
- In our context, the smaller value of sigma in the range of 1 and 2 is more likely to have better location accuracy of edgels. For sigma above 2, have worse location accuracy of edgels result in loss of details.

Therefore, there is a trade-off to choose when deciding to have noise edgel removal or location accuracy of edgels.

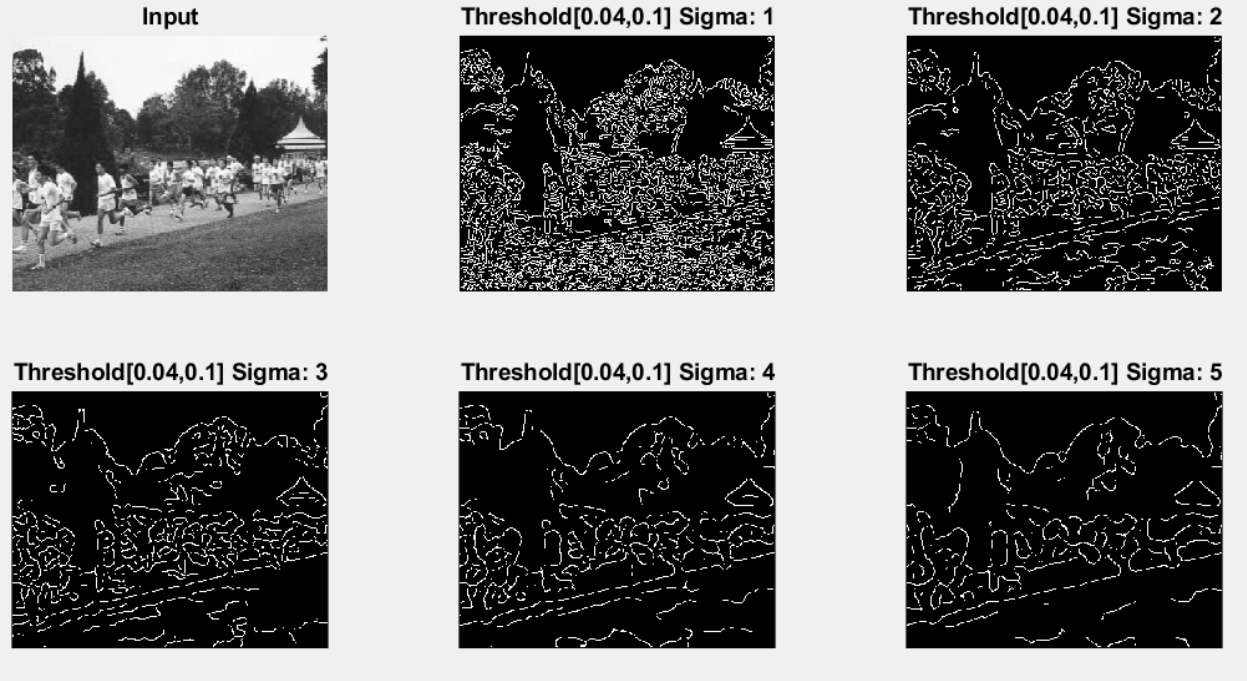
e.ii) Test different lower threshold value

Output:

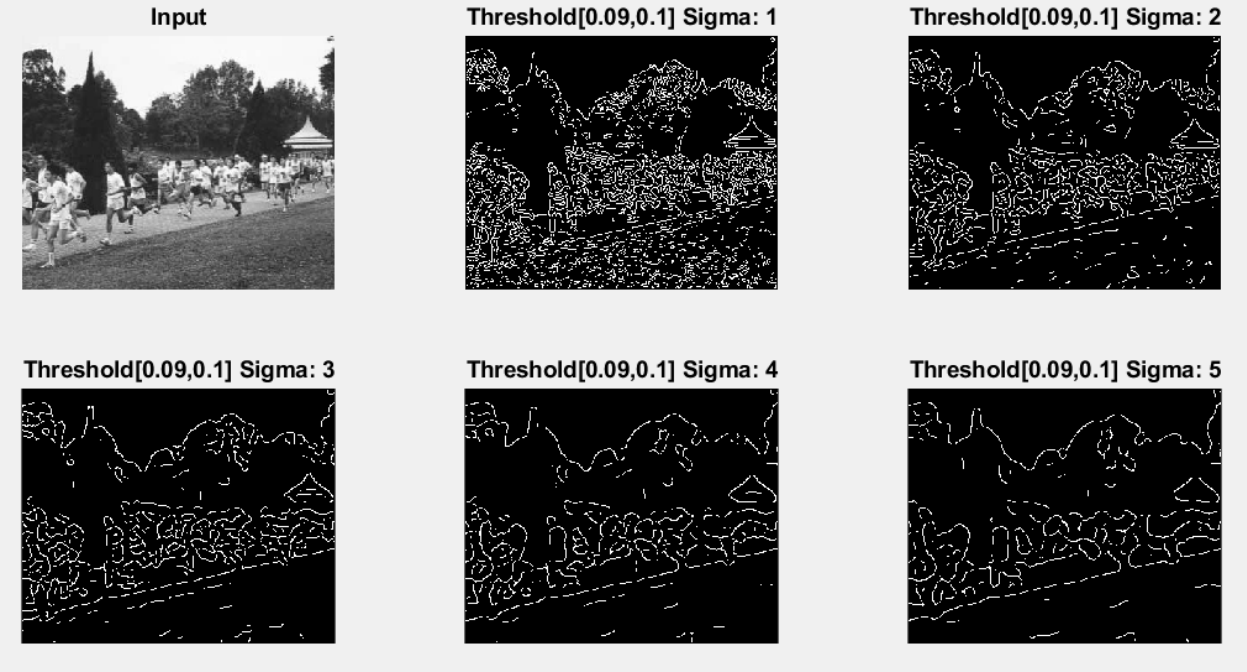
tl = 0.01



tl = 0.04



tl = 0.09



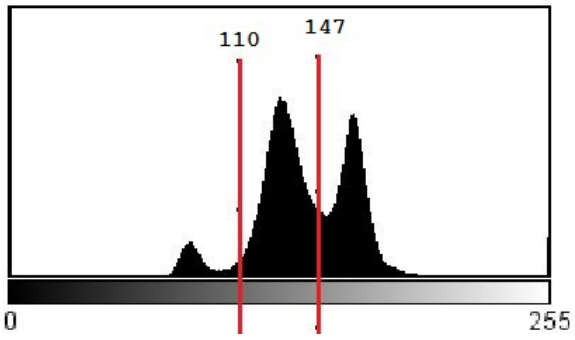
What does it do?

Base on observation, more edges appear on the lower threshold as compare to the higher threshold. The reason is that a lower threshold allows lower pixel intensity values to be part of the final output. It is to identify and determine which pixel is considered an edge or not.

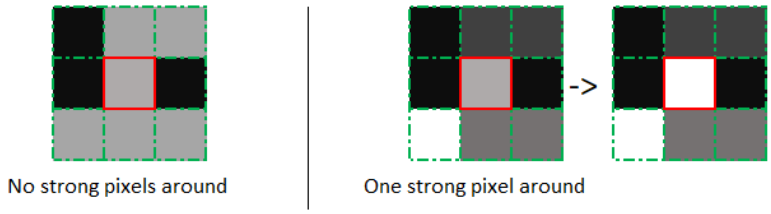
How does it relate to the Canny Algorithm?

The process in the Canny Edge Detector is known as Hysteresis Thresholding. The goal is to determine if the pixel intensity value classifies as an edge. There is 3 different condition to determine if the pixel is ultimate an edge.

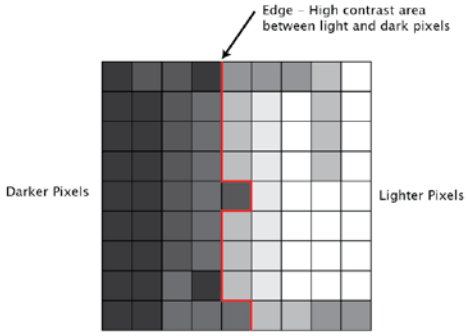
- Firstly, the intensity of the pixel that is not between the defined threshold will be ignored. For example, the threshold between 0.09 to 0.1, which means for a given pixel intensity range between 0.09 to 0.1 value will consider an edge. Below is an illustration with an image pixel intensity histogram, assume 110 is 0.09, and 147 is 0.1 range.



- Secondly, the pixel value within the threshold will select an edge base on neighboring pixels.



- Thirdly, if the neighboring pixels are an edge, it will be an edge vice versa.



2.2 Hough Transform

a) Reuse edge image computed via the Canny algorithm with sigma = 1.0

Since the threshold is not specific, we choose tl and th as 0.09 and 0.1 respectively for this section. As shown below.

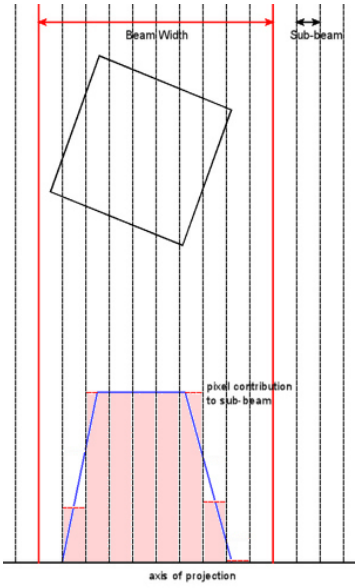
Code:

- Convert to grayscale
  - `img_Grayscale = rgb2gray(img_Origin);`
- Run Canny Algorithms
  - `image_canny = edge(img_GreyScale,'canny',[tl th],sigma);`
- Note: The code is similar to the previous canny implementation. Therefore, not needed to go into detail.

Output:

b) Radon Transform as a substitute for Hough transform. Display H as an image.

Concepts of Radon Transform:



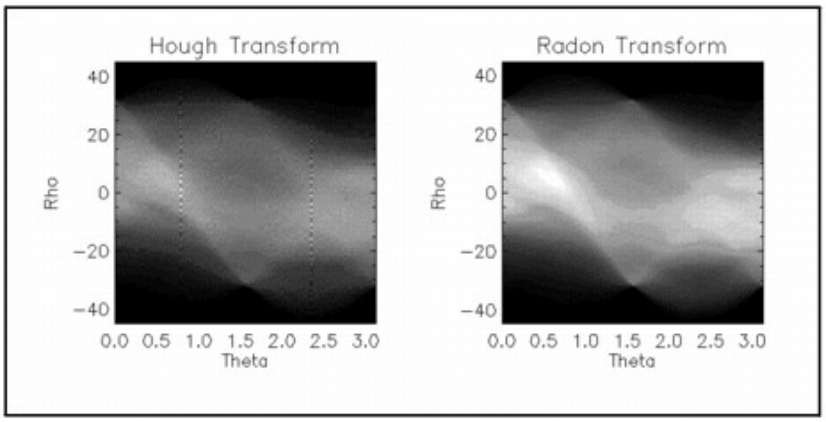
Relevance

By definition, Radon transform is a mathematical integral transform defined for continuous functions on R^n on hyperplanes in R^n. As seen on the figure on the left-hand side, it a projection at an angle. During the actual process of the integration, all the detection function which refer to the 'beam' will rotate for all projection angles (0-179) to generate the parameter space.

On the other hand, 'Hough' transform is inherently a discrete algorithm that detects a line in an image by voting.

Both Radon and Hough are a mapping from the image space to a parameter space of p and theta, but they differ in their point of view. Radon transform derives a point in parameter space from image space, the Hough transform explicitly maps data points from image to parameter space.

Key differences theoretically



Parameter space resolution

For each theta in parameter space, Radon projects the image points on a line described by its angle delta using buckets of size change of p.

Hough takes each image (x,y) and adds the appropriate intensity to all corresponding parameter space bins

This result in hough suffers certain features whereas Radon allows higher resolution in parameter space.

Performance

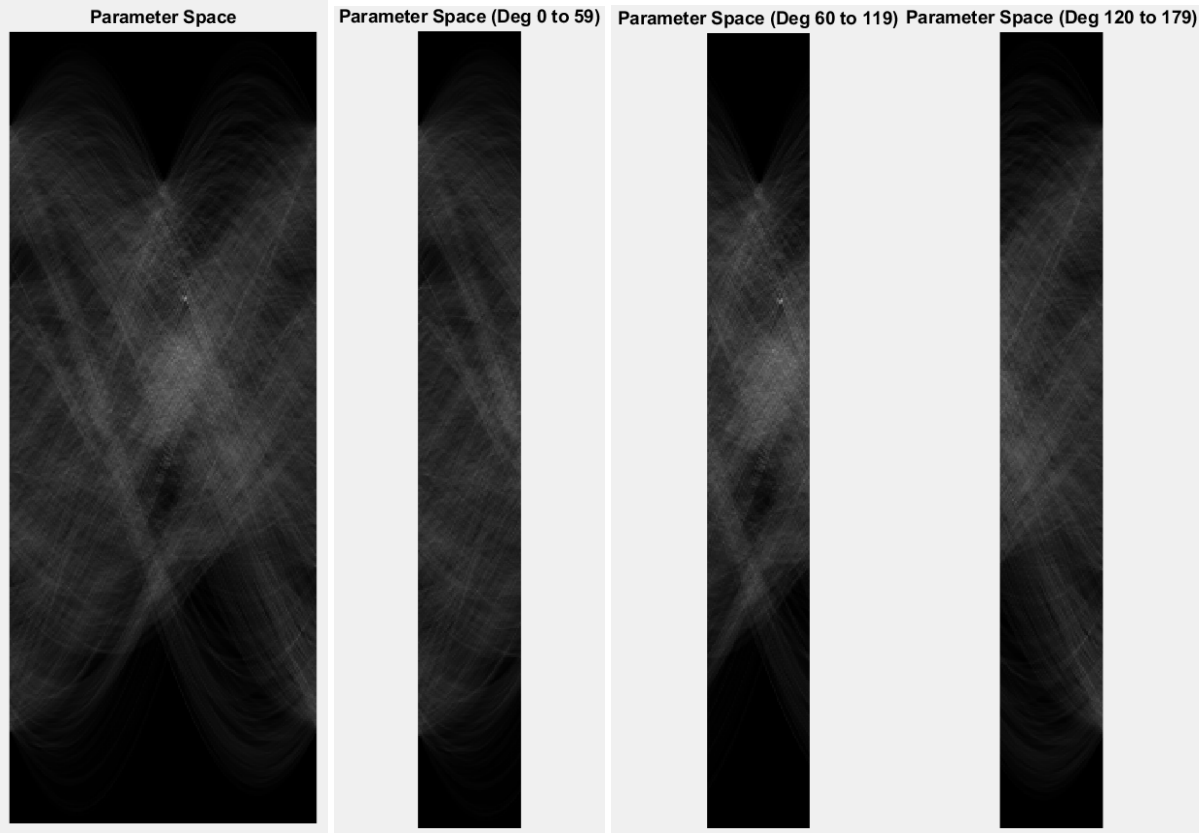
Mathematically, hough is usually faster with 'sine' function being plotted in the parameter space while Radon allows computing only a short interval of the theta in the parameter space due to transform derives a point in parameter space from image space. This can significantly reduce computation time.

Code:

```
[H, xp] = radon(img_Canny);
```

```
imshow(uint8(H))
```

Output:



The result show radon transform will appear similar to how hough transforms in terms of its intensity and pattern in the parameter space. However, the sinusoid line density in by radon transform is likely to be different as compare to hough transform.

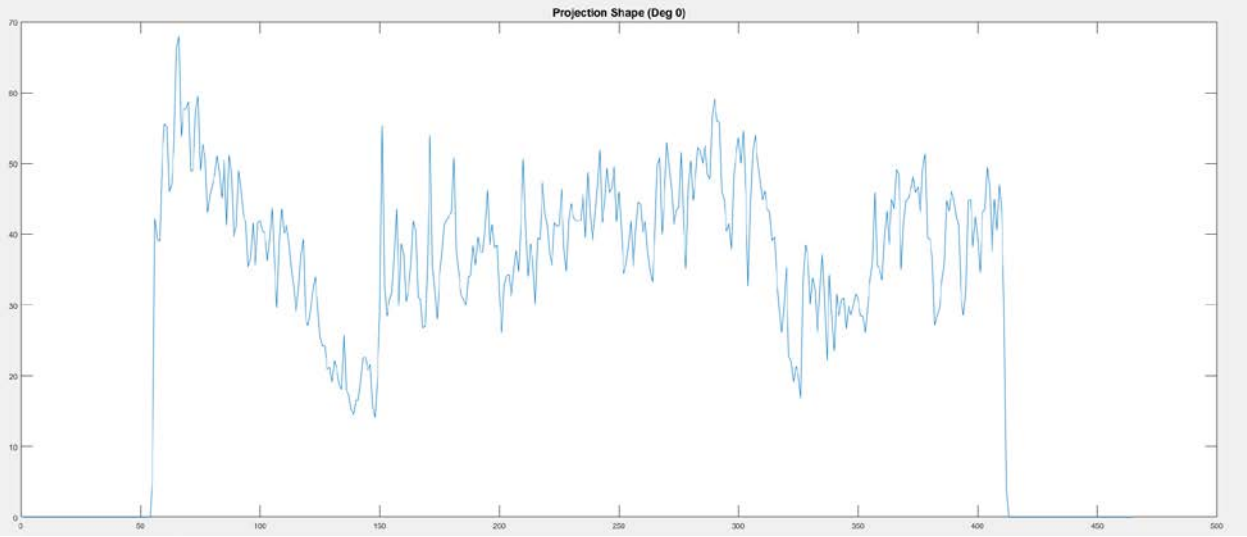
To understand how radon generates the parameter space, simply control the angle to show which also reflects how parameter space being generated. By changing the code below. The implementation is shown below.

```
[H1, xp1] = radon(img_Canny,0:1:59);
[H2, xp2] = radon(img_Canny,60:1:119);
[H3, xp3] = radon(img_Canny,120:1:179);
```

This is to show that the angle generates a column of pixels in the parameter space.

```
H4 = radon(img_Canny,0);
```

To understand how it generates the intensity for each parameter pixel space, run the code above and plot H4



This is the ‘stack of edges thickness’ intensity when projecting the image at 0 degrees. This will then converted into a line of arrays with different intensity levels as shown above as an illustration.

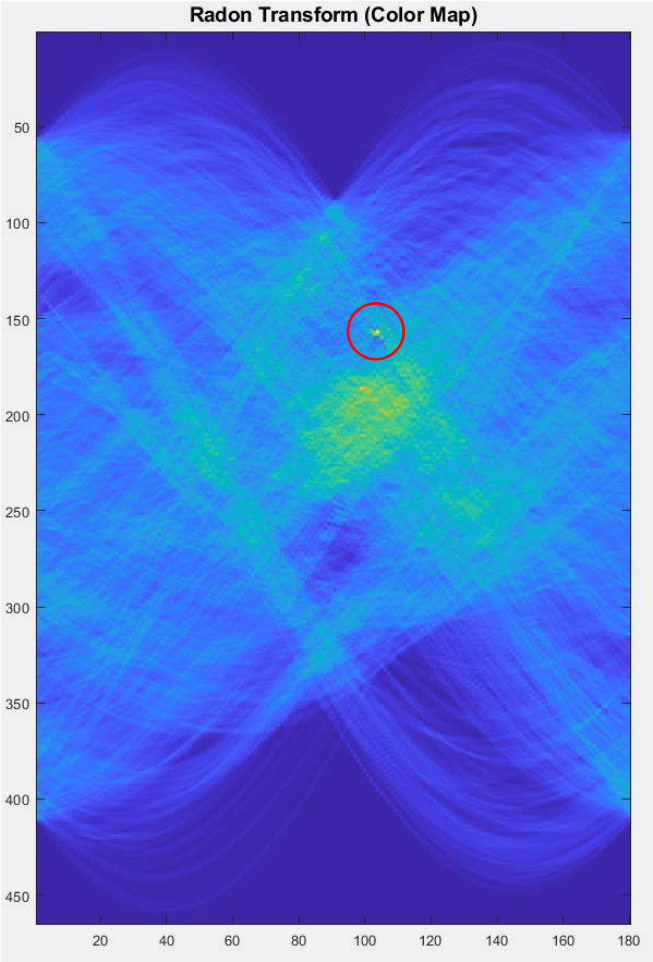
Repeat this for another 179 times will form the full Radon transform’s parameter space.

c) Find the location of the maximum pixel intensity in the Hough Image in terms of [theta, radius].

Code:

```
imagesc(uint8(H))
```

Generate a colormap version of the Radon transform parameter space, this is to observe where the high-intensity cluster is at.



As seen from the graph the approximate value lies around 100~120 theta and 150~160 radius.

```
[y,in] = max(uint8(H));
[value,col_maxI] = max(y);
[~,row_maxI] = max(H(:,col_maxI));
disp('Maximum Value for theta:');
disp(col_maxI);
disp('Maximum Value for radius: ');
disp(row_maxI);
```

```
Maximum Value for theta:
    104

Maximum Value for radius:
    157
```

To find the precise value we can compute and find the highest intensity using the code above. The result is shown on the right-hand side. The highest intensity at parameter space is 104 theta and 157 radius. This means the optimal angle is at 104 clockwise direction and the radius away from the origin point is 157.

d) Derive the equation to convert the [theta, radius] line representation to normal line equation form.

Code:

```
% Find Center of an Imagee
center=size(img_Canny)/2+.5;
```

```
y_off = center(1);
x_off = center(2);
```

- To find the equation we first have to compute the offset as the reference of the image is from the center point. Therefore, it needs to be shifted to the origin point to compute the correct line equation.

Outputs:

```
The center of the image coordinate (x,y)
  145.5000   179.5000
```

- The values of the center of the image as shown above y = 145.5 and x = 179.5

Code:

```
% Find the theta and radial coordinate value
theta = col_maxI;
radius = xp(row_maxI);
```

- To get the radial coordinate we have to find from an array of radial coordinates as shown above.

```
% Finding the line equation with given theta and radius
% Line equation is represent by Ax + By = C , Let C = 0
% Convert Polar to Cartesian Format
[A, B] = pol2cart(theta*pi/180, radius);
B = -B;
C = A*(A+x_off) + B*(B+y_off);
```

- A and B can be obtained using 'pol2cart' function. It converts polar form to cartesian form, the angle has to convert to radian form. The previous radial coordinate will be used in the 2nd argument.
- Variable 'B' is negated because the y-axis is point downward for image coordinate.
- With offset center value, A, and B values found we can derive C as shown above.

Outputs.

```
Line Equation's A is
   18.3861

Line Equation's B is
   73.7425

Line Equation's C is
   1.9806e+04
```

These are the values for A, B, and C of the line equation respectively.

e) Based on the equation of the line Ax + By = C obtained, compute yl and yr values for corresponding xl = 0 and xr = width of image - 1.

Code:

```
% Compute yl and yr with xl and xr
img_dimension = size(img_Canny);
y = img_dimension(2); % width
xl = 0;
xr = y - 1;
yl = (C - A* xl) / B;
yr = (C - A* xr) / B;
```

- We have to get the width image from the image dimension before evaluating corresponding values as shown above.

Output:

```
yl is
  268.5810

yr is
  179.5709
```

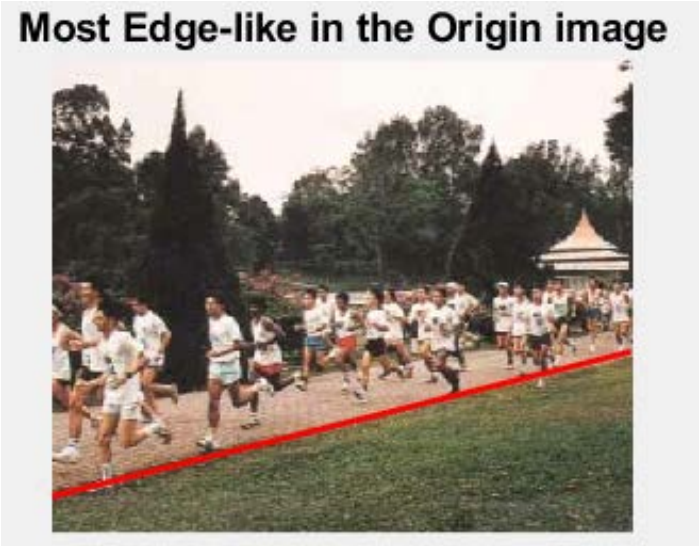
- These are the values for yl and yr

f) Display the original 'macritchie.jpg' image. Superimpose your estimated line.

Code:

```
% Plot the line with origin images
subplot(1,3,2),imshow(img_Origin);
line([xl xr], [yl yr],'Color','red','LineWidth',2);
```

Output:



As shown in the figure, we can see that the most intensity value from the parameter space comes from the most ‘edge’ or ‘strongest line’ feature which is the bottom line of the running path.

It matches the running path to a certain extent, but if look closely it is slightly off the few degree anticlockwise direction.

The reason for such possible error is the image path line may not be truly straight due to uneven terrain or engineer imperfections when constructing the running path. To improve this is using a non-linear function to derived the edge of the path.

Another reason for such inaccuracy is due to canny detection or radon transform process. The calculation here is in discrete function. However, Radon uses a continuous function, which resulted in value deviated from the ideal line. There are many ways to overcome this but depend on the situation, if let says it involves a few images correction, then manual corrective measure by tuning the line with added ‘bias’ is possible. Another method can be using similar images and supervise learn it to best predict the likelihood of an ideal line in the image.

2.3 Pixel Intensity SSD and 3D Stereo Vision

a) Disparity algorithm Implementation.

Code (Preprocess)

```
function D = op_stereo3d(imgL,imgR,Tx,Ty)
    %%----------------------------- Preprocess -----------------------------
    % Check if block size is odd number
    if(rem(Tx,2) == 0 || rem(Ty,2) == 0)
        disp('Please declare a odd number block size')
        return
    end

    % Find the dimension of the images
    [yl,xl] = size(imgL);
    [yr,xr] = size(imgR);
    disp('Image Dimension (Width,Height) is')
    disp(string(xl) + ',' + string(yl))

    % Check if left and right image size is the same
    if(xl ~= xr || yl ~= yr)
        disp('[Error] Please ensure left and right dimension is the same. Try Again!')
        return
    end

    % Find center index value of the block
    center = ceil(Tx/2); % Use ceiling since matlab index start from 1
    disp('Center Size is')
    disp(center)

    % Create a canvas filled with value 1 (remove corner pixels that will not be
    % process) (Dimension X or Y - Block Size) w
    D = ones(yl - (Ty - 1), xl - (Tx -1));
    disp('Canvas Size:')
    disp(size(D));
```

First thing is to ensure the inputs are suitable for SSD matching to process. Apart from that, few variables are declared such as image dimension, the center coordinate of the kernel, and set up canvas size. All these are there to support the SSD matching process and return the disparity map.

A few things I noticed was the given disparity images have a border that is not used for the process. To prevent such a case from happening is by removing the border (unused part) which base on the kernel width or height size subtracts by 1 for the center.

Outputs:

```
Image Dimension (Width,Height) is
256,256
Center Size is
     6

Canvas Size:
   246   246

X Start to End
6   251
Y Start to End
6   251
Limit Search Space (Pixels)
    14

ls Kernel for SSD Eql
     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1
```

This is the output for preprocessing.

Code (SSD Algorithm)

```
%%------------------------------ SSD Algorithms-------------------------------
% Define start and end pixel coordinate of SSD operation
start_x = center;
start_y = center;
stop_x = xl - center + 1;
stop_y = yl - center + 1;
disp('X Start to End');
disp(string(start_x) + '   ' + string(stop_x));
disp('Y Start to End');
disp(string(start_y) + '   ' + string(stop_y));

% Constrain Search Space
limitsearch = 14;
disp('Limit Search Space (Pixels)');
disp(limitsearch);

% Create a template size of all 1 for SSD Eq1
T_1 = ones(Tx, Tx);
disp('1s Kernel for SSD Eq1');
disp(T_1);

% Tranverse y-axis (top-down)
for cur_y = stop_y:-1:start_y
    % Tranverse x-axis (left-right)
    for cur_x = stop_x:-1:start_x
        % Extract the template from left imag  M = N(1:n,end-n+1:end)
        T = double(imgL(cur_y-5:cur_y+5,cur_x-5:cur_x+5));

        % Rotate by 180 degrees with 2 rot90 function
        Tr = rot90(T,2);

        % Reset/Initialize minimum differences
        min_diff = inf;

        % Initialize starting cordinates to be compare with ref
        % coodinates
        min_coor = cur_x;
        % Search for similarity in the right image
        %(14 pixels to the left and right)
        for cur_s = cur_x - limitsearch:1:cur_x + limitsearch
```

```
            % Out of bound search area
            if(cur_s > stop_x || cur_s < start_x)
                continue
            end

            % Get Right side as reference image
            cur_r = double(imgR(cur_y - 5:cur_y+5,cur_s-5:cur_s+5));

            % Calculate SSD with Decompose Formula (Note: One flip
            % kernel will do)
            SSD_eq1 = sum(conv2((cur_r).^2,T_1,'valid'),'all');
            SSD_eq2 = sum(sum(((T).^2)));
            SSD_eq3 = sum(2*conv2(cur_r,Tr,'valid'),'all');

            % Calculate SSD with all required equation
            SSD = SSD_eq1 + SSD_eq2 - SSD_eq3;

            % Find the minimum differences amongst all the SSD
            % matching values
            if(SSD < min_diff)
                min_diff = SSD;
                min_coor = cur_s;
            end
        end

        % Return the offset differences
        D(cur_y-center+2,cur_x-center+2) = -(cur_x - min_coor);
    end
  end
end
```

Details can be found on the source code file called ‘Lab2.m’

b) Download ‘corridorl.jpg’ and ‘corridorr.jpg’, converting both to grayscale.

Code:

```
img_Left = imread(URL_L);
img_Right = imread(URL_R);
```

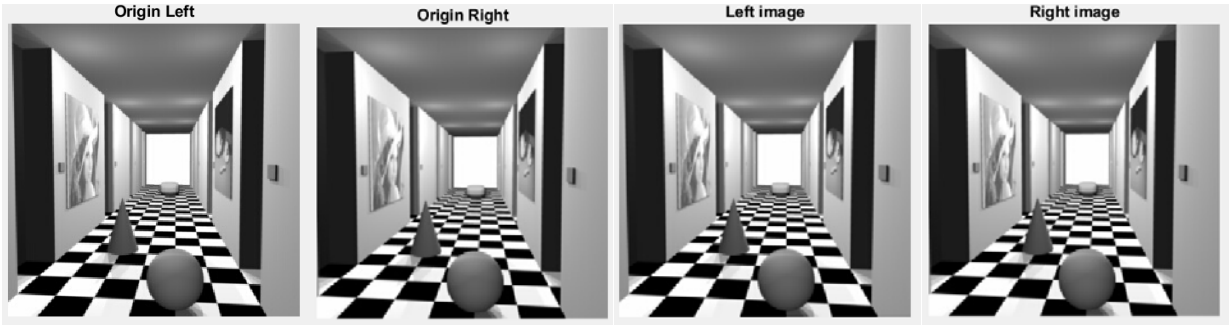
```
% Convert to grey-scale image for left
img_LeftGray = rgb2gray(img_Left);
```

```
% Convert to grey-scale image for right
img_RightGray = rgb2gray(img_Right);
```

Directory to the images is read and convert to the grayscale respectively.

Output:

Origin **Grayscale-Level**



It is hard to tell if the image is at the greyscale level. One way to find out is by showing the properties of the images.

```
Left Image Properties
  Name            Size                Bytes  Class    Attributes

  img_Left      256x256x3            196608  uint8

Right Image Properties
  Name             Size                 Bytes  Class    Attributes

  img_Right      256x256x3             196608  uint8

Sample Outcome Image Properties
  Name           Size             Bytes  Class    Attributes

  img_Ans      251x251            63001  uint8
```

We can identify if it is from multiple channels to one channel from grayscale as shown. Check the images’ properties respectively as shown above to prove the origin is not a fully grayscale image as it contains 3 channels.

c) Run your algorithm on the two images to obtain a disparity map D.

Code:

```
% Run 3D Stereo Functions
dmap = Lab2.op_stereo3d(imgL,imgR,temp_x,temp_y);

% Show the 3D Stereo Result
Lab2.display_dmap(dmap);
```

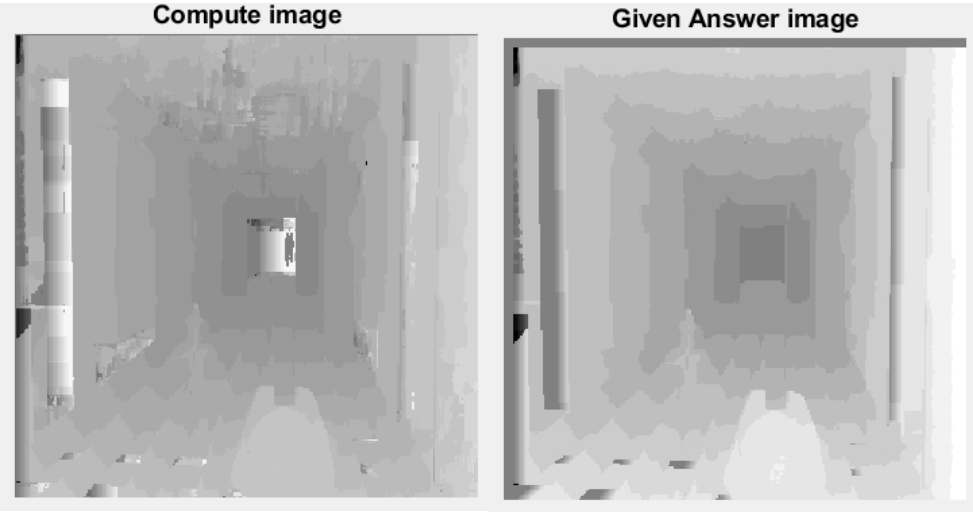
- The code is run via op_strero3d() function with the respective arguments, it will return a disparity map which then parses into display_dmap() function to display the images in the figure below.

```
imshow(-imgS,[-15,15]);
```

- The image was inverted and set a normalized interval range between -15 to 15

Output:

Limit search = 14 **Given Disparity Map**



Quality of the disparity

Generally, the computed disparity map does show the key depth of the image, where the further object has a smaller rate of change while the closer the object is the greater rate of change also known as displacement. This can be proven with 2 fingers tricks with our eyes. The image also shows that the higher change can be seen as brighter while small change appears darker.

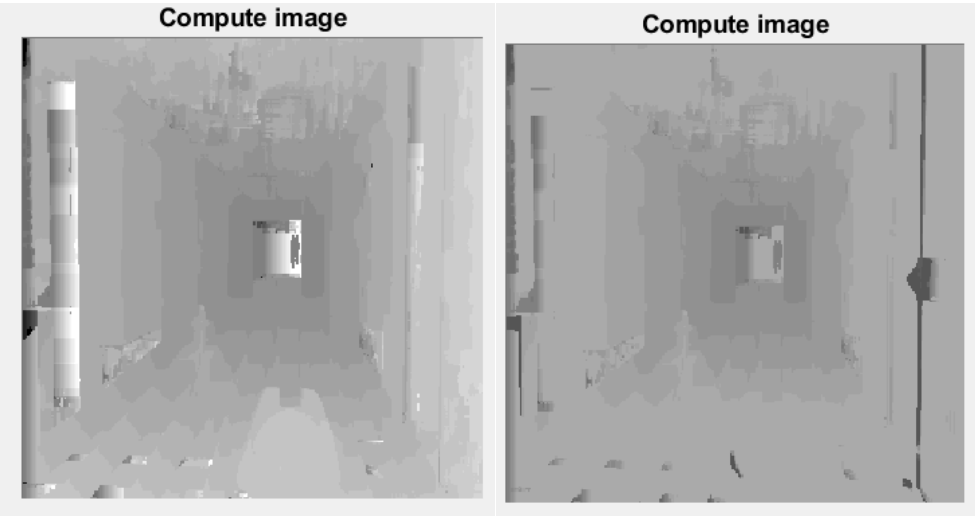
However, the computed disparity map is not as good as compared to the given disparity map 'corridor_disp.jpg'. It is noisy especially the ceiling of the alley and the pillars. One possible reason is that it is sensitive to faint lighting change, the ceiling shows a faint small gradient in grayness level.

There are a few shapes and objects in the images. Unfortunately, only a few of them are visible. The object that is wide enough within the search limit can be seen such as the sphere. Other objects cant be seen such as the switches, cone, paintings, the cuboid. This shows that an object can be easily viewed from us if it is near while the object further away is barely visible. This is due to narrow displacement in

the far object as our image is limited pixel by pixel level, the higher the depth, the lower the disparity it can be. This results in a bad estimation of the disparity value.

Notice a few weird distortions at the end of the alley and the left and right walkway walls. This shows the unstable or unreliable this algorithm can be. The given disparity image is the enhancement algorithm or methods which helps a smoother and consistent image.

Limit search = 20 **Limit search = 5**



I try different limited space to see if it works, technically it does not improve the image, it has less level depth with lower limit search space as compared to higher-value search space. This also depends on the acceptable range of value to be shown as well as the display of image code.

d) Rerun the algorithm on real images of 'triclops-i2l.jpg' and 'triclops-i2r.jpg'.

Code:

Same as the previous part.

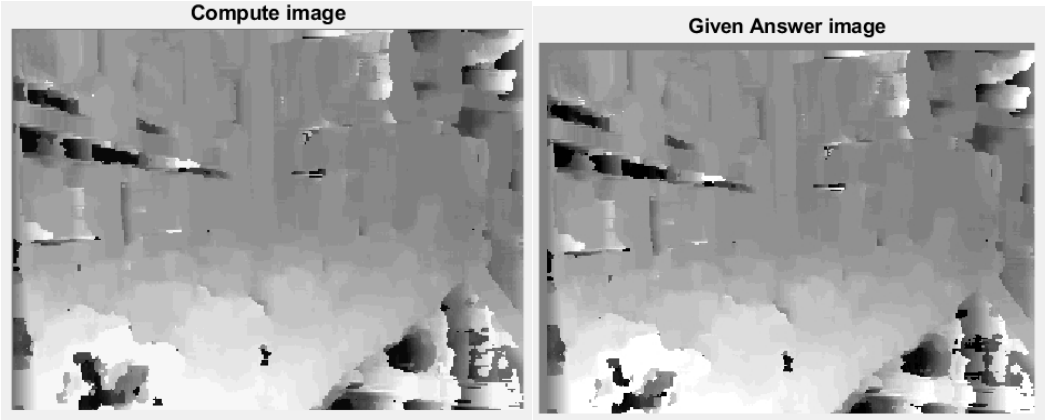
Output:

```
Left Image Properties
  Name            Size                Bytes  Class    Attributes

  img_Left      240x320x3             230400  uint8

Right Image Properties
  Name             Size                 Bytes  Class    Attributes

  img_Right      240x320x3              230400  uint8

Sample Outcome Image Properties
  Name           Size             Bytes  Class    Attributes

  img_Ans      235x315            74025  uint8
```

Notice the aspect ratio of the image is no longer 1:1 as compare to the previous part.

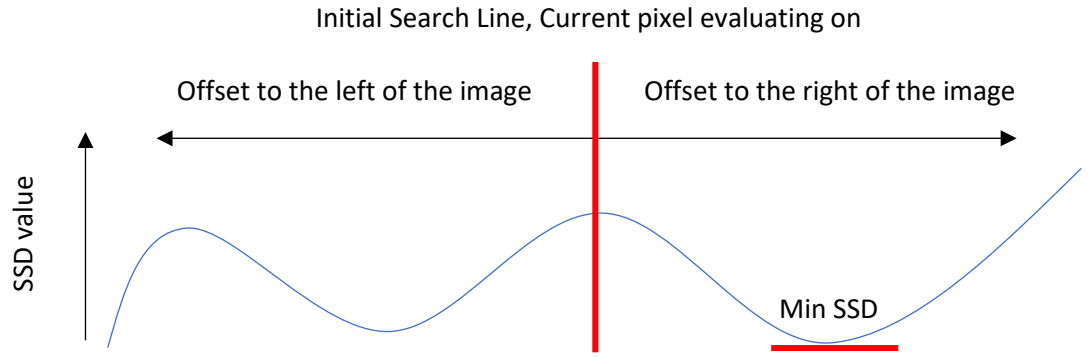
```
Image Dimension (Width,Height) is
320,240
Center Size is
     6

Canvas Size:
   310   230

X Start to End
6   315
Y Start to End
6   235
Limit Search Space (Pixels)
    14
```

These are the respective parameter to be used for the SSD matching process.



Accuracy of the estimated disparity

The computed disparity map is almost identical to the given answer image. In comparison with the corridor, the result can be said to be worst. As there is a distinctive mixed depth of the near object in the images.



There is noise on the walking path of the images which makes it impossible to find the right match. This also true for near bushes.

As illustrated above the result is chosen when SSD finds the global minima value which may be hard to distinguish when the contrast is low. These images are also blurry compare to the corridor images. This result feature barely visible due to the computation of disparity is not reliable.

One possible solution is changing the template dimension to see if it works better.

Conclusion

The SSD matching has its limitation and practical uses are very subjective since it is unstable and unreliable. But it is important to understand the fundamental of 3D Stereo. This help ensures student can learn and continue to stay curious and learn beyond what they are being told to do so.

--end of the report--