LAB 6:

Name: Swarna Prabhu

UIN: 932003046

CANNY EDGE DETECTOR ALGORITHM IMPLEMENTATION IN MATLAB:

The lab report has been divided into 3 sections.

A. Steps showing the implementation of the Canny Edge Detector
B. Output for different images
C. Inference on change in the parameters (change in sigma, low threshold, high threshold)
D. Appendix: Code

SECTION 1: Implementation steps:

1>**Original gray scale image:**

Fig 1 represents the original image which is the grayscale image of Lena size 512*512

2> **Noise removal -Gaussian Smoothening**

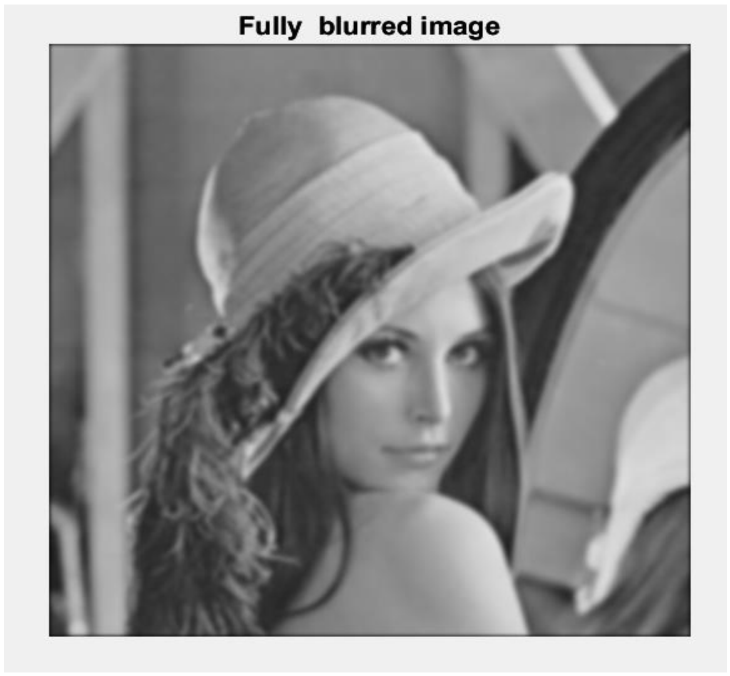


Fig 2

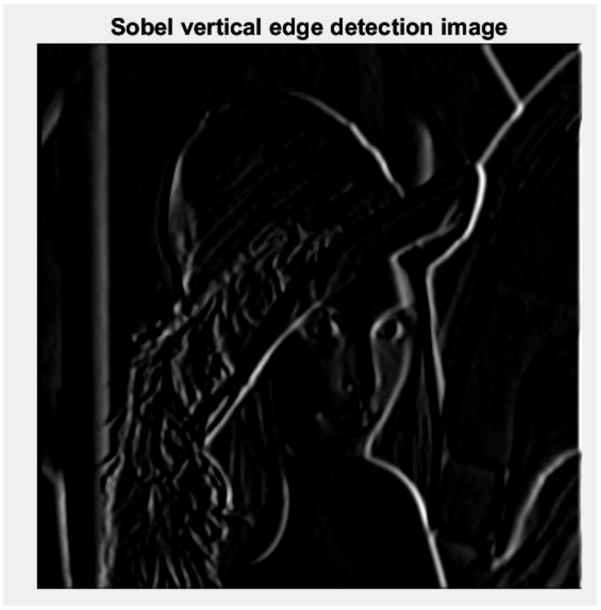
In the 2nd step, the gray scale image is smoothened using the Gaussian convolution process. Since convolution with Gaussian kernels will remove the noise and blur the image, a kernel of size 5 *5 is chosen with standard deviation of the Gaussian spread of sigma = 3. This ensures that unwanted noise present in the images won't be included as edges in the Canny detection. Since Gaussian kernels are separable in x and y components, initially Gaussian weights in x direction is calculated using the below formula

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

The convolution with the filter results in Gaussian smoothening in x direction. The transpose of the Gaussian weights along x direction is used to perform the vertical convolution. The final result of the smoothened image is shown in Fig2. Here sigma is given as a user parameter that can be varied depending on the image to get accurate Canny result. Increase in value of sigma will increase the spread of the variance of the Gaussian distribution. Hence a very large value of sigma will cause the smoothening to be very high which can result to the loss of valid edges. On the other hand smaller values of sigma will result in less blurring hence less noise is removed. Therefore the noise can also appear as extra edges in the image.

3> **Sobel – horizontal and vertical edge detection**

Sobel vertical edge detection using horizontal filter   Sobel Horizontal edge detection using vertical filter.



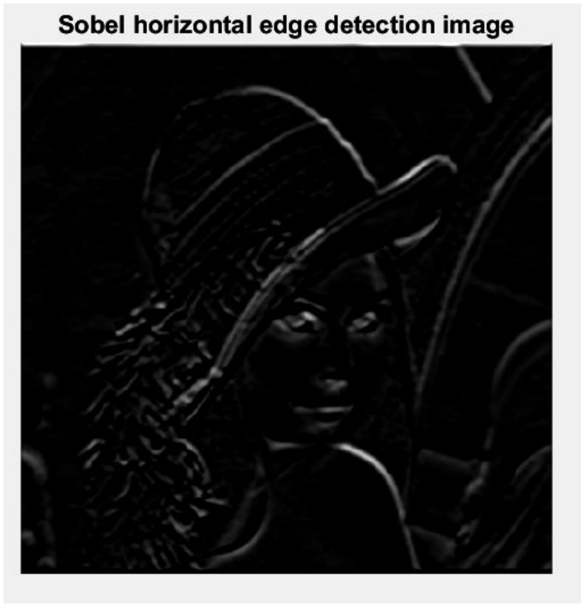


Fig 3

In the third step of the implementation, Sobel operator is used to perform edge detection on the smoothened version of the image. Here we use 2 Sobel filters to detect the edges in each direction. Horizontal Sobel operator [1 0 -1; 2 0 -2; 1 0 -1]

Vertical Sobel Operator [1 2 1; 0 0 0; -1 -2 -1]

The horizontal sobel operator will start scanning the edges from left to right direction due to which the vertical edges are detected as shown in Fig 3 left image. Similarly, the Vertical Sobel filter processes the edges from top to bottom, hence edges that are horizontal in shape are detected. In this we get edges detected that are non-uniform including the fat edges. This is taken care in the next steps.

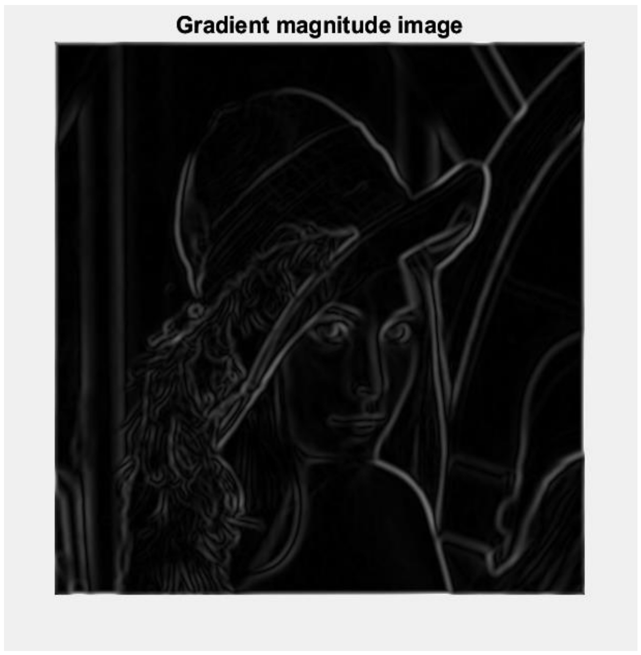
4>**Gradient Magnitude and Angle calculations using the above Sobel Hx and Hy images**.

After edge detection using Sobel operators, images are segregated into Gradient magnitude and Gradient angles. Let gradx , grady denote the images obtained in above section after applying edge detection in horizontal and vertical directions. Then Gradient magnitude is computed using the formula:

Gradient Magnitude = sqrt (gradx.^2 + grady.^2)

Gradient Angle = tan -1(grady/ gradx)

The result of the above operations is given below. The gradient magnitude image which is normalized to range [0,1] gives the overall edges detected which are non-uniform and appear to be faint(Fig 4 left). Since angle always preserves maximum information ie orientation of the image the same appears on the Gradient angle image as shown in Fig 4 below(right).



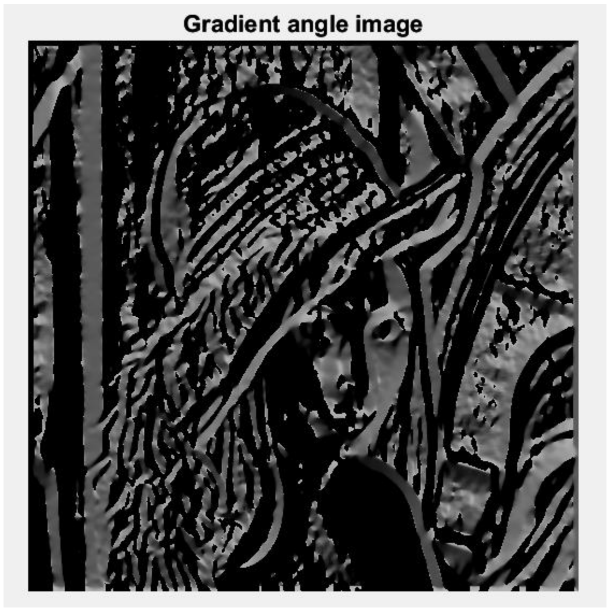
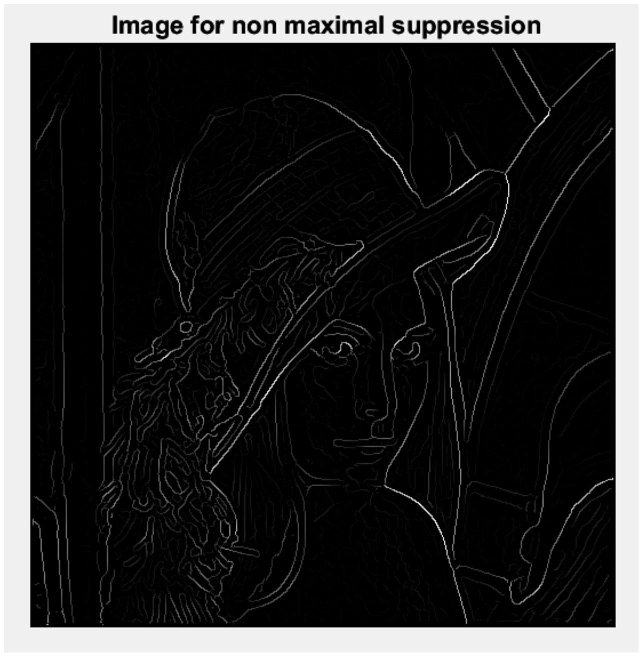


Fig 4

5>**Non maximal Suppression Result**:

From the gradient magnitude we can see clearly that the edges have extra ridges that appear thick and must be further processed to make it look thinner. This is done in this step using a technique referred to as Non maximal suppression. In this, we quantize the angle orientation (in degree) of every pixel into 4 range of values namely 0 ,90 , -45 and +45 and categorize every angle to fall into one of the ranges. Before this the negative angles are converted to positive by adding + 180 degree. Then we evaluate the top and lower neighboring pixels for a given pixel along a particular direction based on the quantized angle category they fall into. If the value of the current pixel is greater than both the neighbors that pixel edge is valid and is retained. If not, then we consider that pixel as a ridge and is removed. In this way thinning of the edges is obtained. The result of this step is shown in Fig 5 where we see that the extra ridges on the edge have been removed and the edges are now thinner.



The quantization of the angles is down as shown in the below:

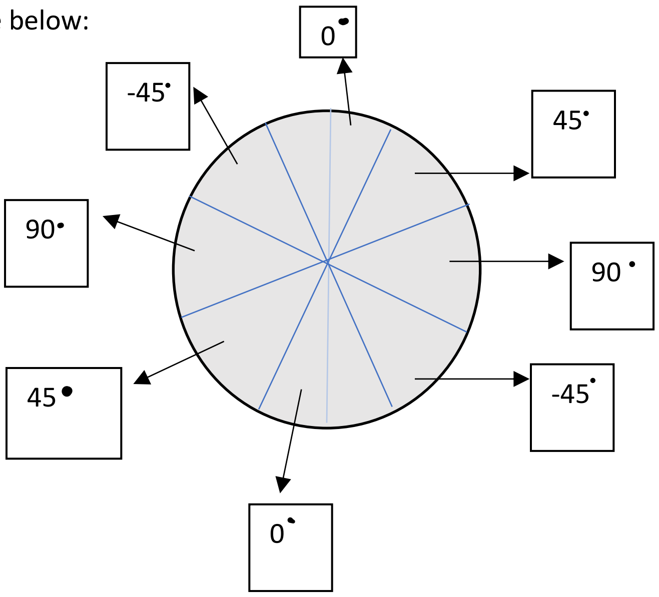


Fig 5

6>Hysteresis thresholding (Double thresholding) and Edge Connectivity

In order to make sure that we extract the valid or correct edges in Canny's algorithm, we use pair of threshold values – low threshold and high threshold values to extract the valid edges in that given range. By tuning the values of low and high threshold which is usually in the ratio of 1:2 or 1:3 we divide the image to 2 categories by applying low threshold and high threshold separately. This method is known as hysteresis thresholding. The low threshold image is obtained by considering all the valid pixels above the low threshold value as 1 and high threshold image will have all the valid pixels marked as 1 that have value greater than high threshold. In this step we get two binary images. Since all the low threshold image will also have the high threshold valid pixels too these need to be removed. Hence the final low threshold image is obtained by removing the non-zero pixels of high threshold image.

The result of the above process is shown in Fig 6. It is evident that low threshold image will have relatively large number of valid pixels marked 1 and hence we see more extra edges in the low threshold image. Whereas the high threshold will have very less valid edges.

Connectivity: In this step we connect the pixels we mark as valid edge pixels in the low and high threshold images based on 8 – connectivity. Corresponding to all the edge pixels in the high threshold the 8 - connected neighbors are marked as valid in the low threshold image. This process is repeated until all valid edge pixels of high threshold are visited. And then we perform logical AND to remove the invalid pixels in the newly derived low threshold.

In the last step, to the high threshold image we add all the valid pixels from the low threshold and the final implementation is shown in Fig 7 left

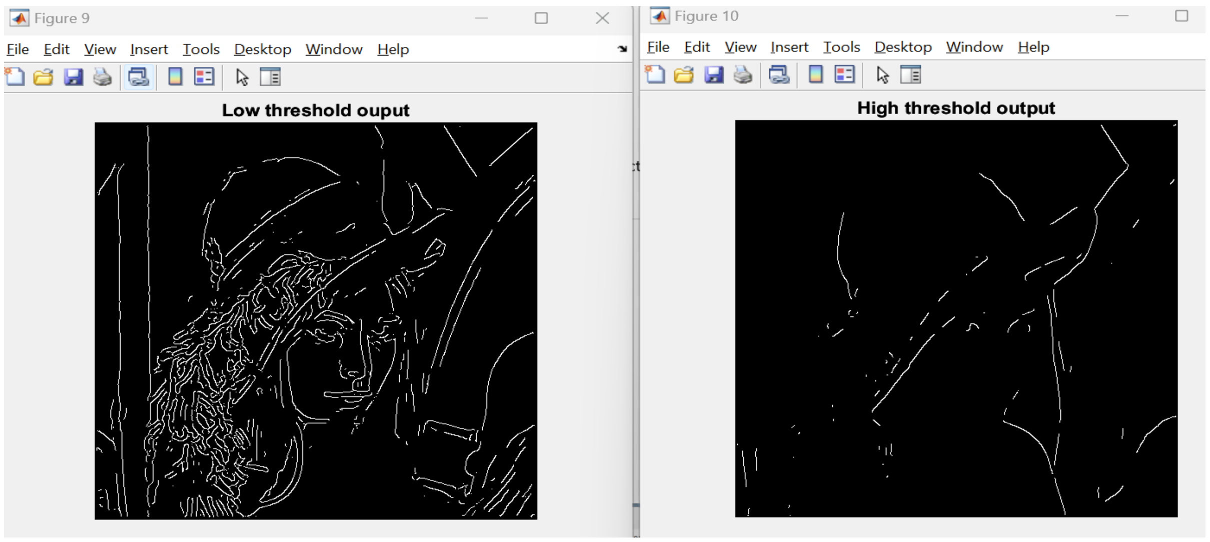


Figure 9
File Edit View Insert Tools Desktop Window Help
Low threshold ouput
Figure 10
File Edit View Insert Tools Desktop Window Help
High threshold output

Fig 6

7>Final Result from my implementation of Canny Edge detection algorithm and Matlab's inbuilt Canny function

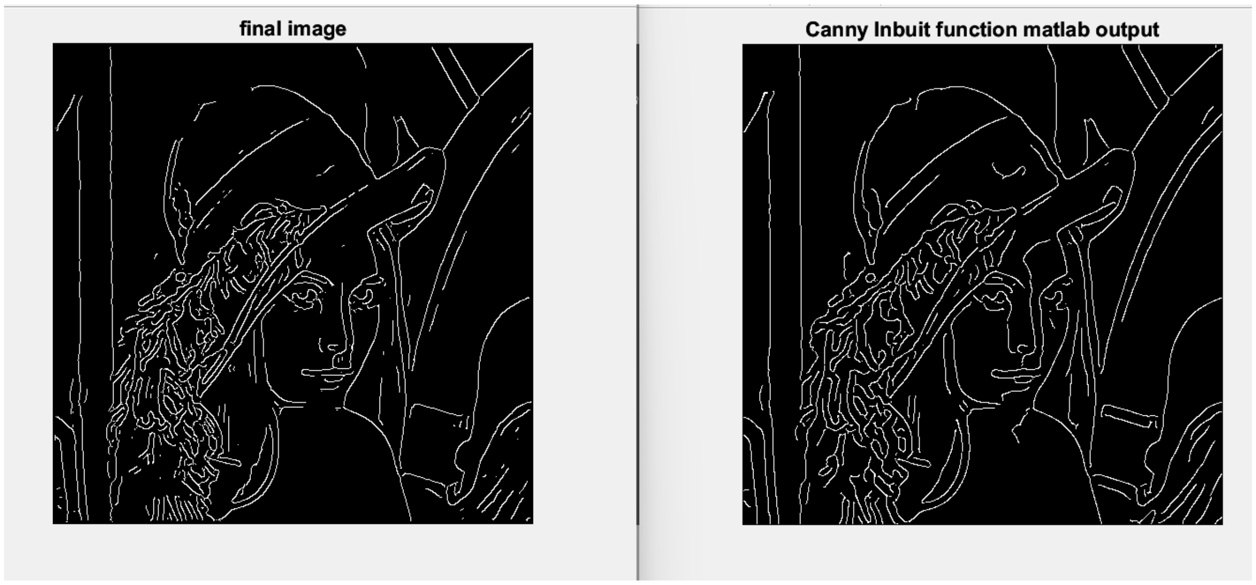


Fig 7:

Conclusion and Analysis:

The final implementation of Canny Edge detector algorithm has been displayed in Fig 7 -left and can be compared to the inbuilt edge function in MATLAB with Canny as the option. We see that 100 percent perfect reconstruction of matlab's inbuilt is not possible since Matlab can have its own algorithm or implementation for edge tracking and connectivity since in my implementation after double thresholding only connectivity is performed. But the result closely matches MATLAB's Canny as all the important edges has been identified.

SECTION 2: TESTING WITH DIFFERENT IMAGES

Testing for the accuracy and correctness of the implementation with different grayscale test images and the final results along with the best parameters, threshold for each is mentioned below:

Results:

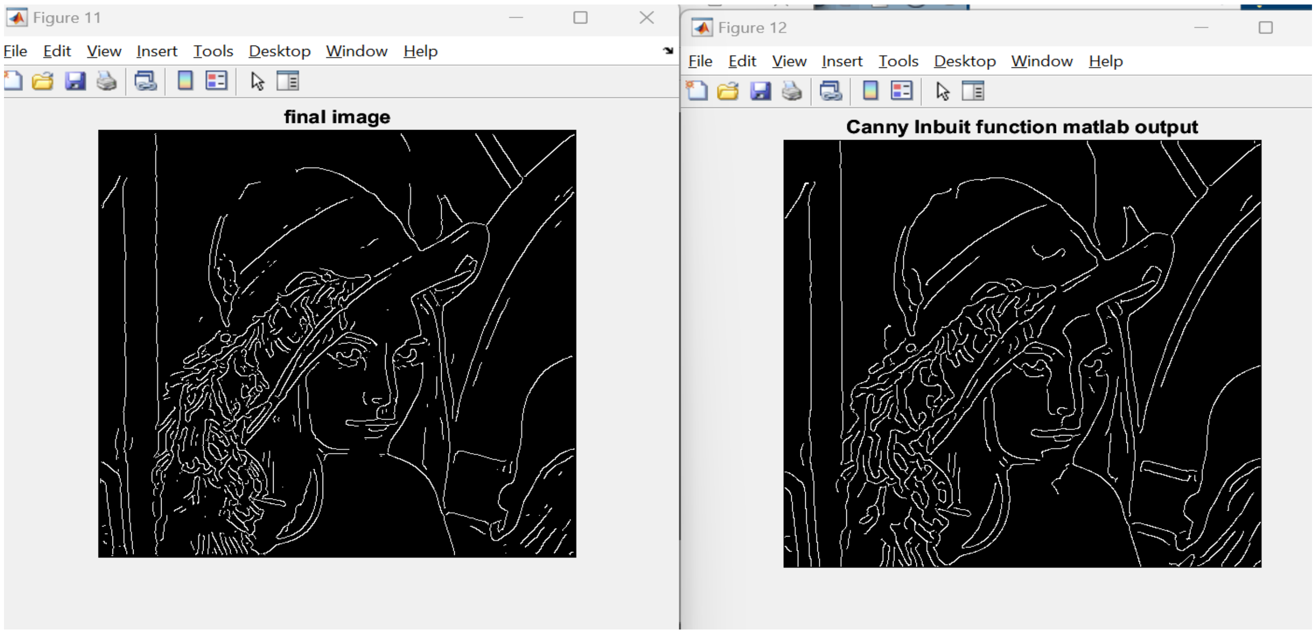


Figure 11
File Edit View Insert Tools Desktop Window Help
final image
Figure 12
File Edit View Insert Tools Desktop Window Help
Canny Inbuit function matlab output

```
Image: Lena sigma = 3; kernel size M = 5; low = 0.20; high = 0.60;
```

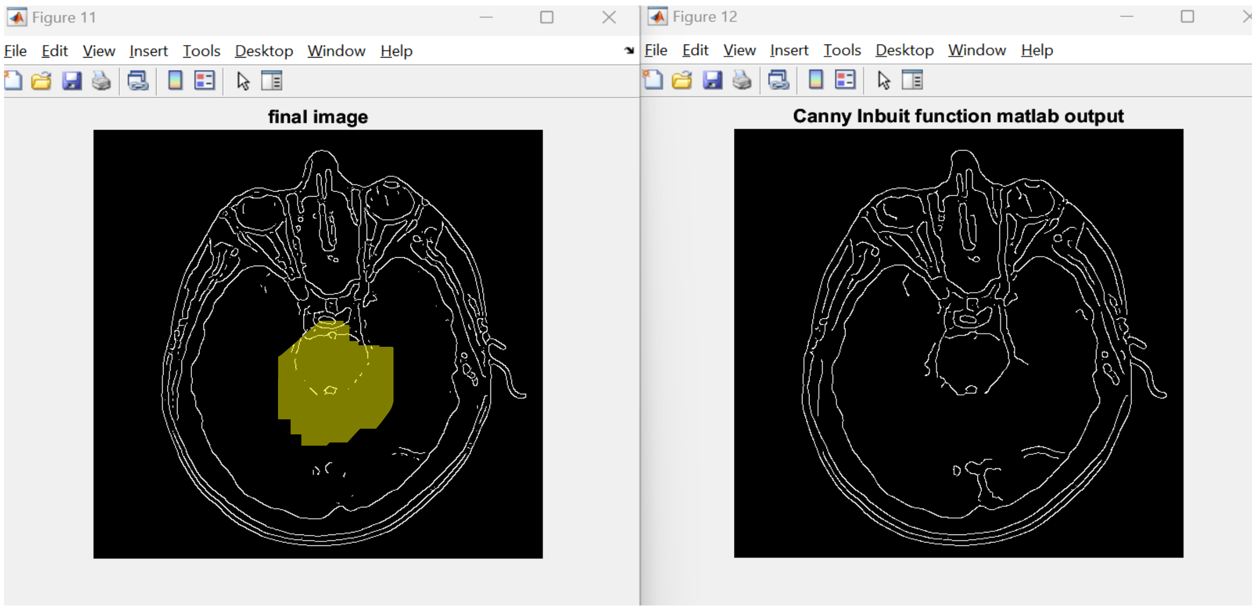


Figure 11
File Edit View Insert Tools Desktop Window Help
final image
Figure 12
File Edit View Insert Tools Desktop Window Help
Canny Inbuit function matlab output

Image: HEAD CT High threshold = 0.5, low threshold = 0.2 sigma = 3 MSE =0.026

The posterior part of the head CT marked in Yellow is retained in my implementation as expected from Canny's algorithm (from MATLAB result) for the above parameter values. This shows the correctness of the implementation.

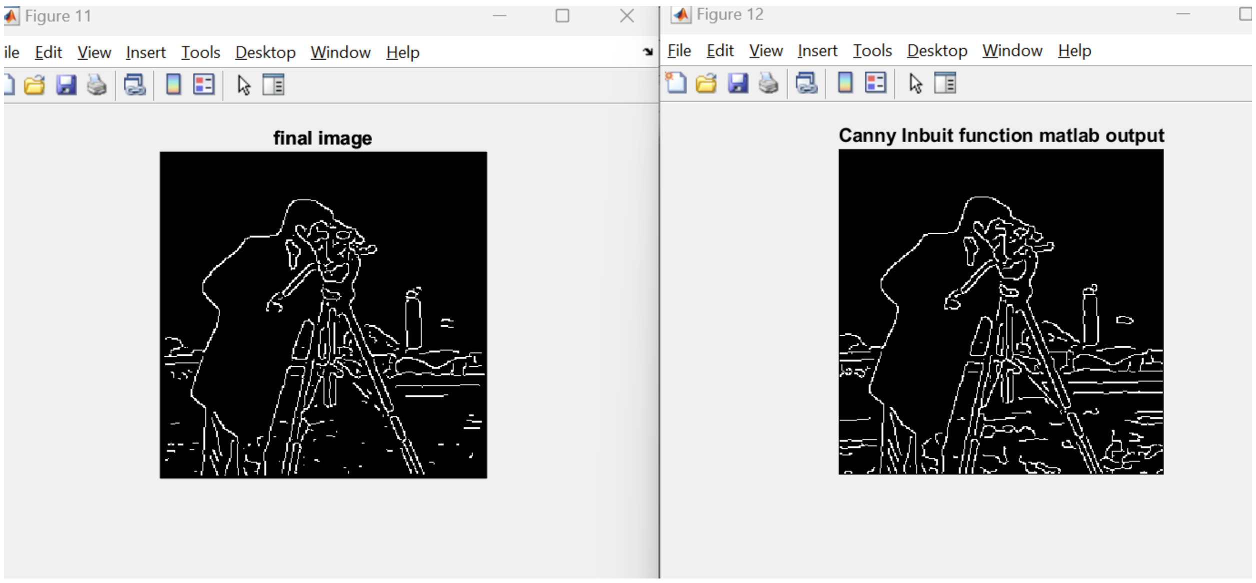


Figure 11
File Edit View Insert Tools Desktop Window Help
final image
Figure 12
File Edit View Insert Tools Desktop Window Help
Canny Inbuit function matlab output

Image: Camera man LOW = 0.1 high = 1.01   sigma = 2

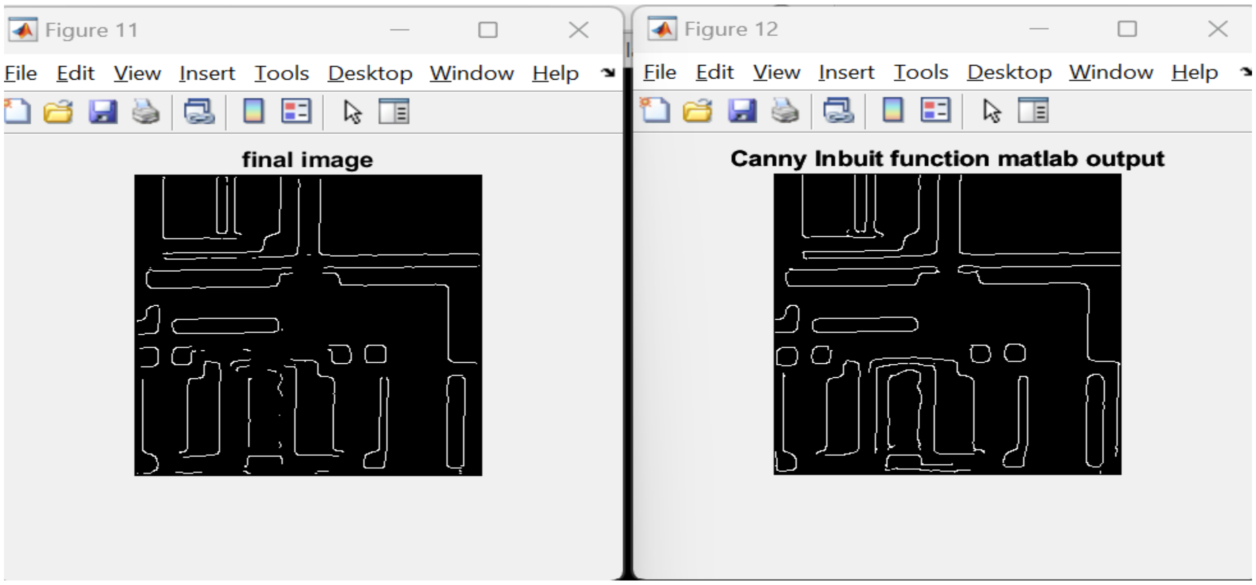


Figure 11
File Edit View Insert Tools Desktop Window Help
final image
Figure 12
File Edit View Insert Tools Desktop Window Help
Canny Inbuit function matlab output

```
Image circuit: sigma = 3; low = 0.40; high = 1.01
```

Different parameter for the above circuit image:

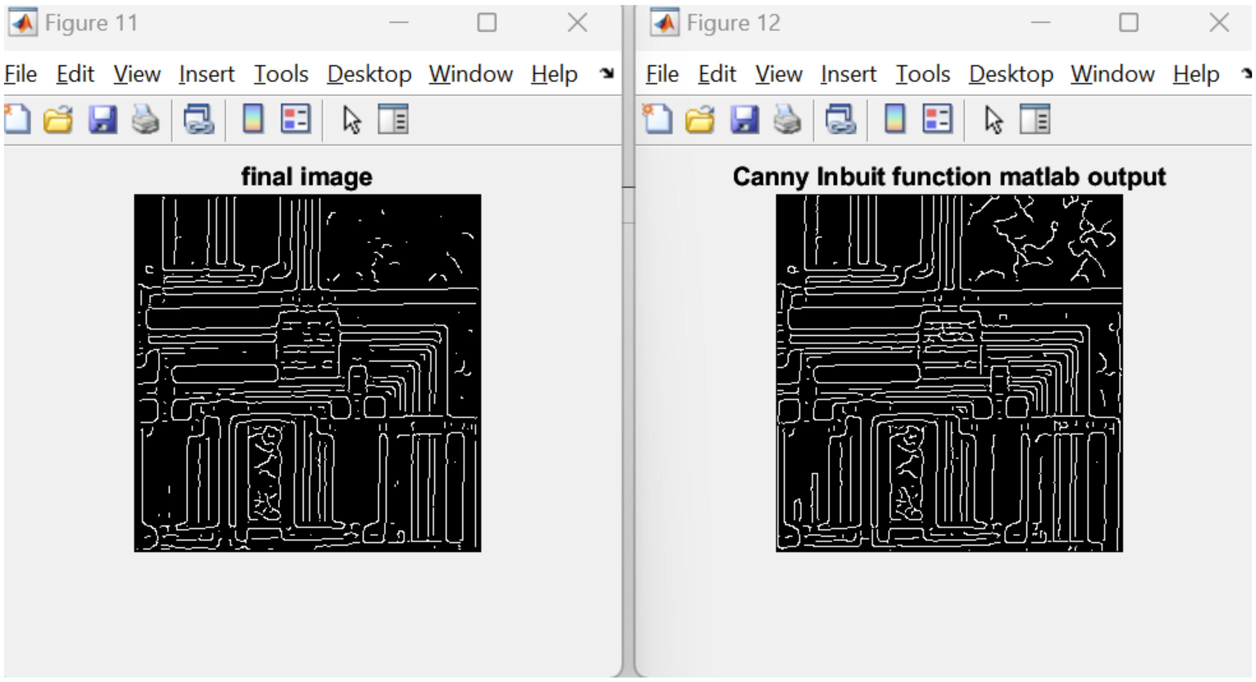


Figure 11
File Edit View Insert Tools Desktop Window Help
final image
Figure 12
File Edit View Insert Tools Desktop Window Help
Canny Inbuit function matlab output

// Result for the same image but different parameters `sigma = 2;low = 0.20;high = 0.60;`

<u>SECTION 3: INFERENCE ON CHANGE IN PARAMETERS</u>

Effects of change in parameters (sigma and threshold) for same image and conclusion for the same:

3.1> Effect of Change in threshold values: 1> Keeping the low threshold same and varying only high threshold

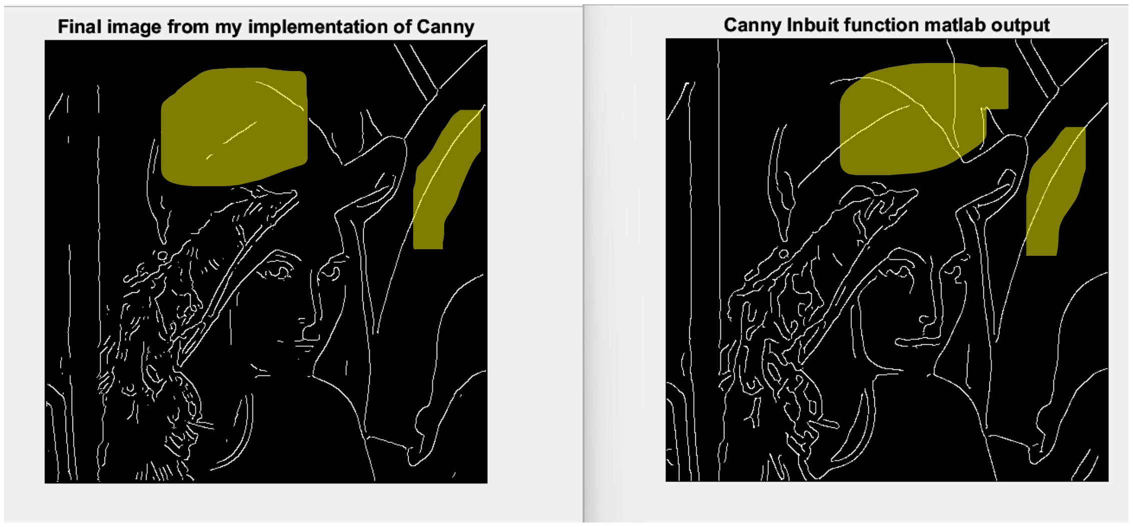

Final image from my implementation of Canny
Canny Inbuit function matlab output

Fig 3.1

High =1.1 Low 0.2 Sigma = 3

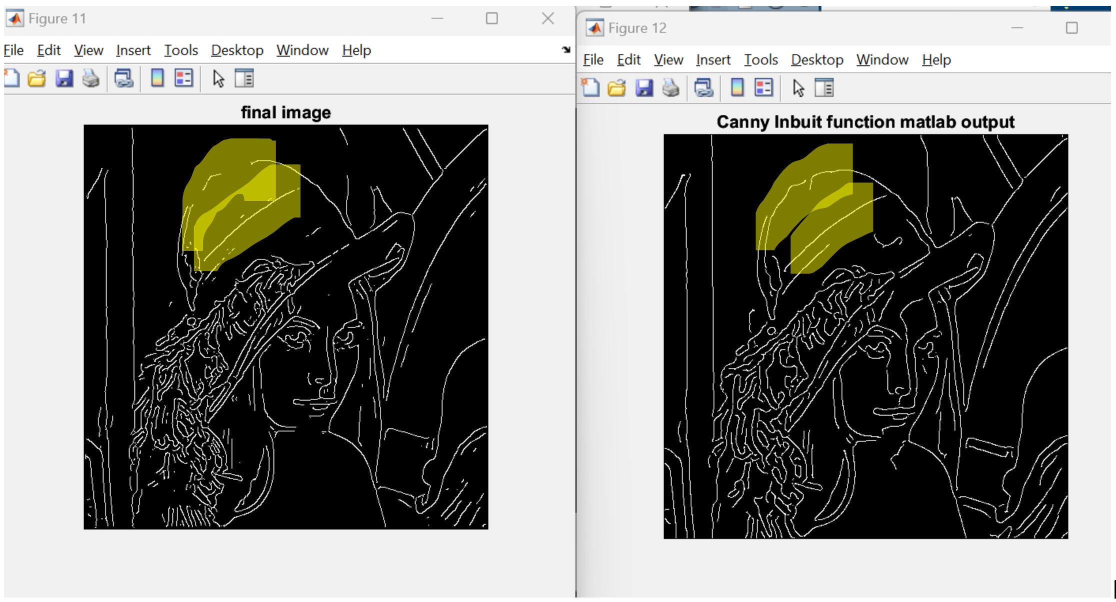


Figure 11
File Edit View Insert Tools Desktop Window Help
final image
Figure 12
File Edit View Insert Tools Desktop Window Help
Canny Inbuit function matlab output

Fig 3.2

High = 0.6 Low = 0.2 Sigma = 3

We see that when high threshold is kept 1.1 higher value very few values will be marked as valid edges hence, we see less edges in the final output for high threshold 1.1 compared to one with 0.6 high.

2>Keeping high threshold same but varying or different low threshold values.

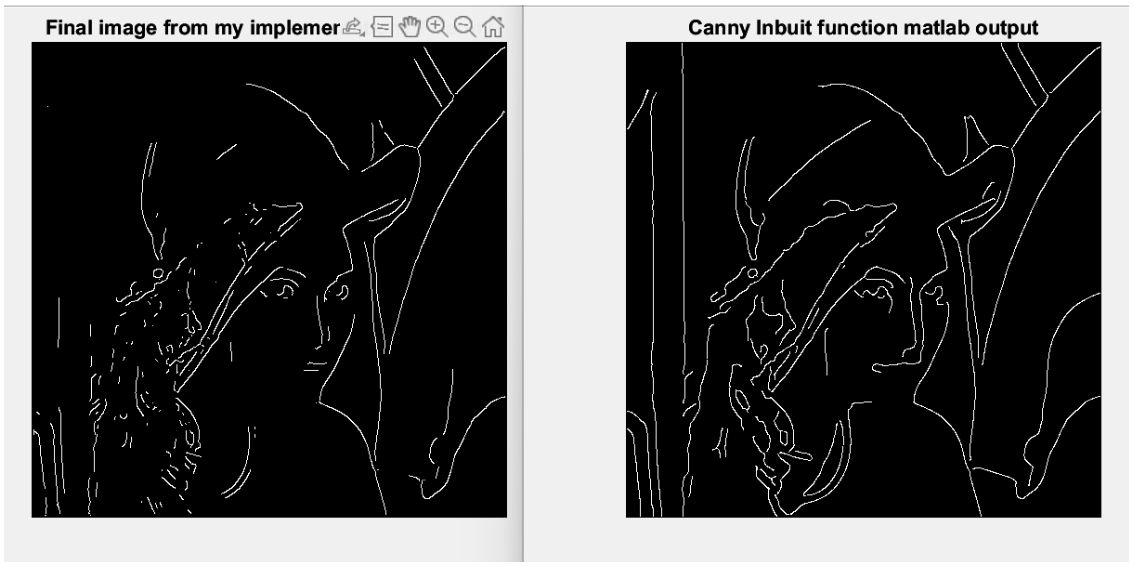

Final image from my implemer
Canny Inbuit function matlab output

Fig 3.3

High 0.6 ,Low 0.5 ,Sigma = 3

Here we keep the high threshold and sigma same as in Fig 3.2 and compare by varying thelow threshold only. Since low threshold marks all the weak edges they are not detected very well when we select a high value for low threshold. Hence in Fig 3.3 the low = 0.5 doesn't detect the weak points very well and we get insufficient edges in the final output. Whereas in Fig 3.2 with low being 0.2 a good amount of edges are detected.

<u>Effect of Sigma parameter</u>

3.2> Keeping the same value of low and high threshold, changed the value of sigma to 1.5 and kernel size to 4

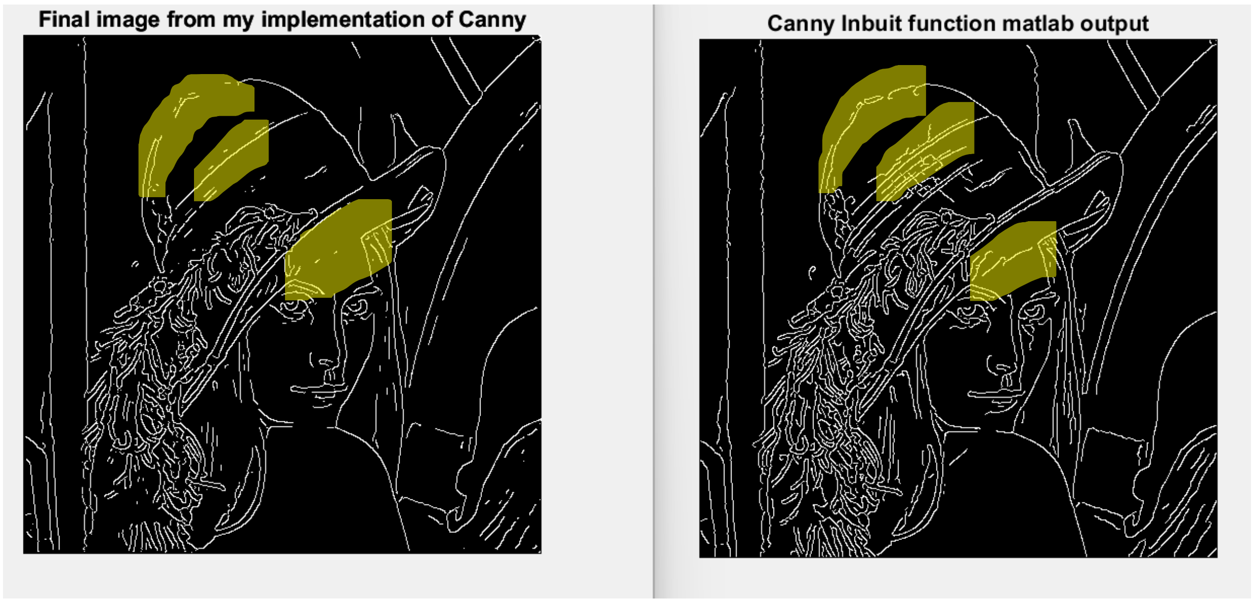

Final image from my implementation of Canny
Canny Inbuit function matlab output

Fig 3.4 High = 0.6 Low 0.2 Sigma 1.5 Kernel size = 4

Here we compare the results from Fig 3.2 High = 0.6 Low = 0.2 Sigma = 3 Kernel size = 5

And Fig 3.4, the sigma is chosen to be smaller value, which implies that the Gaussian smoothening wont be very effective since the variance of the distribution is less spread compared to in Fig 3.2 where sigma is 3. Due to this in Fig 3.4 less noise will be eliminated, and this noise can actually appear as extra edges around the actual detected edges marked in yellow in Fig 3.4. It can be compared with Fig 3.2 which looks relatively cleaner at the highlighted areas.

<u>Appendix</u>

<u>Code :</u>

```
function MSE = canny_detector(x, sigma , low, high)


%read the grayscale input lena image
% x=imread("lena_gray_512.tif");
% x = imread("livingroom.tif");
% x = imread("headCT.tif");
% x = imread("mandril_gray.tif");
% x = imread("pirate.tif");
%x = x/ max(x(:));

%here the input grayscale image is taken as a function parameter and
```

```
%displayed
x1 = x;
figure;
imshow(x1);
title('Original gray-scale image');

%computes the size of the input image and store it as row, columns
[row, columns] = size(x1);


%As a first step to the implementation , it is essential to  remove the noise in the
input image  by smoothening or blurring the gray scale image
%This is acheived by  performing a Gaussian convolution Kerenl

%create the gaussian kernel
% m - size of gaussain kernel and simplicity K=1
m = 5;
K= 1;

% sigma - this parameter is a user entered parameter passed as a function
% arguement
% sigma = 3;
%create a matrix g for gausian kernel of size m*m

g = zeros(m,m);
for i= 1:1:m
 for j = 1:1:m
 %compute the exponential part of the formula given in textbook eq 3-45
 %here only the component along x direction is taken since we are
 %implemneting Gaussian convolution seperatley - horizontally and
 %vertically
 %this is due to the reason that Gaussian kernels are seperable:
 exp_numpart = -[((i-((m-1)/2)-1).^2) ];
 exp_denpart = 2*sigma *sigma;
 g(i , j) = K*exp(exp_numpart/ exp_denpart);
 end
 end

%AFTER Computation of the weights normalize the values for range [0,1] using by
diving it with the sum
g_norm = g/ sum (g(:));

% g_norm = g;

%performs  horizontal convolution of the image using conv2 function with the
horizontal filter
%keeping the final size similar to the original -hence 'same' option
conv_output_horizontal = conv2(x1, g_norm, 'same');

%performs the vertical convolution from the above result, for which we take
%the transpose of the weights to get the values along vertical direction
%(uses seperability property for Gaussian kernels)
g_norm_transpose = transpose(g_norm);

conv_output_vertical = conv2(conv_output_horizontal, g_norm_transpose, 'same');
```

```matlab
figure;

imshow(uint8(conv_output_horizontal));
title(" Partially blurred image -horizontal comvolution");

%displays the final image for gaussian blurring to remove the noise in the
%image
figure;

imshow(uint8(conv_output_vertical));
title('Fully  blurred image');
%once we perform Gaussain smoothening to remove the noise , next we need to
%get the edge detection using derivatives along  horizontal and vertical
%direction

%this is done using the sobel filter

%this is the horizontal filter that will track the edges in horizontal
%direction , hence it will detect the vertically
Sobel_horizontal = [-1 0 1 ; -2 0 2; -1 0 1];

%Sobel vertical filter that will track from top - bottom so we get
%horizontal edges
Sobel_vertical = [ 1 2 1;0 0 0;-1 -2 -1];


%detects edge by using a horizontal filter and display the result

sobel_conv_h = conv2(conv_output_vertical, Sobel_horizontal, 'same');

figure;

imshow(uint8(sobel_conv_h));
title('Sobel vertical edge detection image');

%detects edge by using vertical filter and displays the result

sobel_conv_v = conv2(conv_output_vertical, Sobel_vertical, 'same');

figure;

imshow(uint8(sobel_conv_v));
title('Sobel horizontal edge detection image');


%computes the gradient of the horizontal and vertical edge detected image
%using sqrt function
%formula used is mag = sqrt ( ( horizontal_edge_image )^2 +(verticaledgeimage )^2)

magnitude = sqrt(sobel_conv_v.^2 + sobel_conv_h.^2);

normalized = magnitude / max(magnitude(:));

figure;
```

```
imshow((normalized));
title('Gradient magnitude image');

%calculate the angle theta - which denotes the slope of the gradient
%tan inverse ( y / x) gives the angle , mulitply by ( 180 / pi ) to get the
%angle in degree for further processing of the images.
theta = atan2(sobel_conv_h, sobel_conv_v ) * 180 /pi;

%displays the angle image
figure;
imshow(uint8(theta));
title('Gradient angle image');

%iterate through every pixel to convert the negative angle to posititve for further
processing and
%simplicity
for i = 1:row
    for j = 1:columns
        if (theta(i,j) < 0 )
            theta(i,j) = theta(i,j) + 180;
        end
    end
end
% theta = theta;
% normalized = normalized;
%
%%%%%%implenentation of maximal suppression to get rid of the thick edges
%%%%%%using the angle of the gradient image from edge detection

[row, columns] = size(normalized);
%we initialize a zero 2D matrix to store the result from non maximal
%suppression process
supress_arr = zeros(row, columns);

%the gradient magnitude has thick edges , we use suppression to thin out
%the edges and make it more uniform ,so the angle image is quantized into 4
%levels 0 , 90, -45 and +45 and every magnitude pixel that falls on one of
%the quantized angle group is compared with its adjacent pixel in the
%respective direction to check if the current pixel has maximum intensity
%value..if so retain else mark it as 0.This is implenented by iterating
%through every pixel of the gradient normalized image as show in the below
%nested for loops
for i = 2:row -1
   for j = 2:columns-1
        curr_pixel = normalized(i,j);   %%reads the current pixel

        if (theta(i,j) < 22.5 && theta(i,j) >= 0  ) || ( theta(i,j) <= 180 &&
theta(i,j) >= 157.5 )

         ang = 0;   %%here we quantize to 0 angle and  consider only horizontal
neighbours
        elseif ( theta(i,j) >=22.5 &&  theta(i,j) < 67.5 )
            ang = -45; %%here we quantize to -45 angle and consider diagonal right to
left as neighbours for comparison
```

```
        elseif (  theta(i,j) < 112.5 && theta(i,j) >=67.5 )
            ang = 90; %% here we quantze to 90 degree angle where we compare
vertically

        elseif (theta(i,j) >= 112.5 && theta(i,j) < 157.5)
            ang = 45; %%diagonal from right to left considered and angle is quantized
as 45

        end
     %%if angle is zero category then top and bottom neighbours since x
     %%axis is vertical and y axis is horizontal based on the
     %%implementation in textbook chapter 10
       if(ang == 0  && (normalized(i+1, j) <= curr_pixel && normalized(i-1, j) <=
curr_pixel))

         supress_arr(i,j) = normalized(i,j);

       %if angle is -45 degree then top right and bottom left neighbours
       %compared
       elseif (ang == -45 && (normalized(i-1, j+1) <= curr_pixel && normalized(i+1,
j-1) <= curr_pixel))

         supress_arr(i,j) = normalized(i,j);
       %%if angle is 90 degree then consider right and left pixels
       elseif (ang == 90  && (normalized(i, j-1) <= curr_pixel && normalized(i, j+1)
<= curr_pixel))

         supress_arr(i,j) = normalized(i,j);
       %%if angle is 45 we consider top left and bottom right diagonal
       %%neighbours
       elseif (ang == 45 && (normalized(i-1, j-1) <= curr_pixel && normalized(i+1,
j+1) <= curr_pixel))

         supress_arr(i,j) = normalized(i,j);

      else
         supress_arr(i,j) = 0;

       end



%         j = j+1;


    end
%     i = i+1;
end

%normalized the image from non maximal suppression to range [0,1]
supress_arr = supress_arr/ max(supress_arr(:));

%display the image for the above computation
figure;
imshow((supress_arr));
```

```
title(" Image for non maximal suppression");



%%%%%%%next we perform hysteris thresholding or double thresholding
%%%%%%%procedure to remvoe the edge points that are not true
%select threshold parameters
%here high and low are the parameters passed by the user in the script_call

high_thresh = high*max(max(supress_arr));
% 1.5 *max(max(supress_arr));

low_thresh = low * high_thresh;

%%next initialzie 2 arrays for storing the high and low threshold based
%%image
gnh = zeros(row, columns);
gnl = zeros(row, columns);
%in this thresholding process we binarize the image and iterate through
%every pixel and if any value is greater than low threshold the
%corresponding pixel is marked 1 (strong ) and stored in gnl
%%Same procedure for high threhsold and is stored in gnh
for i = 1:row
    for j =1:columns
      if(supress_arr(i,j) >= low_thresh)
        gnl(i,j) = 1;
      end
      if (supress_arr(i,j) >= high_thresh)
          gnh(i,j) = 1;
      end

    end
end

%%since the low threshold value will also have the high thresold values we
%%remove it by subtracting gnh from gnl
gnl = gnl - gnh;

%displays the result for low threshold and high threshold images
figure;
imshow((gnl));
title("Low threshold ouput");


figure;
imshow((gnh));
title("High threshold output");


%%%%implementation of connectivity for edges

%here we use 8 connectivity to check if any image pixel intensity is
%surrounded by a strong threshold pixel
%%here the non zero pixel intensity in high threshold image gnh is taken as
%%'strong' pixel and the non zero pixel in low threshold image is weak
```

```
%%pixel

%initalize a copy of the low threshold image
gnl_connect =gnl;
visited = zeros(row, columns);
%modifed of gnl
for i =2:row-1
    for j = 2:columns-1

    if( gnh(i,j)== 1)
         visited(i,j)=1;
%for evry strong pixel in high threshold imge we mark the 8 neighbour of
%for that pixel position in the low threshold image gnl_connect
         gnl_connect(i-1,j-1)= 1; gnl_connect(i-1, j) = 1; gnl_connect(i-1, j+1) =1;
          gnl_connect(i, j-1) = 1; gnl_connect(i, j+1) =1; gnl_connect(i+1, j-1) =1;
          gnl_connect(i+1, j) =1; gnl_connect(i+1, j+1) =1;
          gnl_connect(i,j)=1;
    end

   end
end
%%now we have 2 versions of valid pixels for low threshold image --gnl and
%%gnl_connect ..so we need to get the final low threshold image by
%%performing and operation (multiplication since images are binary )since
%%we have to set those pixels zero that are not valid and conisder only
%%common valid pixels between the orignal low threshold gnl and computed
%%one gnl_connect
gn_new = zeros(row, columns);
for i =2:row-1
    for j = 2:columns-1
%        if(visited(i,j)==0)
%            gnl(i,j)=0; // element wise gnl with subtraction
        gnl_connect(i,j)= gnl(i,j) * gnl_connect(i,j);
%        gnh(i,j) = gnh(i,j) | gn_new(i,j);
    end
end

%%%append to gnh all the valid pixels from modified gnl above


final_img = gnh | gnl_connect;

%displays the final result of the image


figure;
imshow((final_img));
title("Final image from my implementation of Canny")

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%comparison with the inbuit canny output
% canny_input = imread("walkbridge.tif");
% x=imread("lena_gray_512.tif");
canny_input  = x1;
```

```
% canny_input = imread("headCT.tif");
% canny_input = imread("mandril_gray.tif");

%x = imread("mandril_gray.tif");
%defining the input parameters for the inbuit function
% sigma = 3;
lower_threshold = low;   %0.1;
upper_threshold = high;    %1.5; %1.01;

thresh = lower_threshold*upper_threshold;
%using the inbuilt edge matlab function by passing canny as an option along
%with sigma and threshold values
canny_out = edge(canny_input, 'Canny', thresh, sigma);

%displays matlabs canny output
figure;
imshow(canny_out);
title('Canny Inbuit function matlab output');

%calcultes the mean square error between the 2 images for accruacy of the
%implementation
% diff = abs(final_img- canny_out ).^2;Mean_square_error =
sum(diff(:))/numel(final_img);
% MSE = (Mean_square_error);
MSE = immse(double(final_img), double(canny_out));


end


Script to call the above function :


clear all;

close all;



% x = imread("livingroom.tif");
%  v = imread("headCT.tif");
% x = imread("mandril_gray.tif");
% v = imread("cameraman.tif");
%%testing for circuit image
% v = imread("circuit.tif");


% sigma = 2;
%  testing for cameraman image
% % high = 1.01; %1.01;
% v = imread("cameraman.tif");
% low = 0.10;
% high = 1.01;
% error = canny_detector(v, sigma,low,high );
```

```
%
%%best parameters for lena test
v=imread("lena_gray_512.tif");
sigma = 3;
low = 0.2;
high = 0.6;
error = canny_detector(v, sigma,low,high );


%%parameter and testing for brain /head CT IMAGE
% v = imread("headCT.tif");
% sigma = 3;
% low = 0.20;
% high = 0.50;
% error = canny_detector(v, sigma,low,high );


%%testing parameters for circuit image


% v = imread("circuit.tif");
% sigma = 3;
% low = 0.20;
% high = 0.6;
% error = canny_detector(v, sigma,low,high );
```

References: The logic and idea for implementing Suppression , Thresholding , connectivity has been referenced from DIP -4TH Edition Gonanzelez and Woods