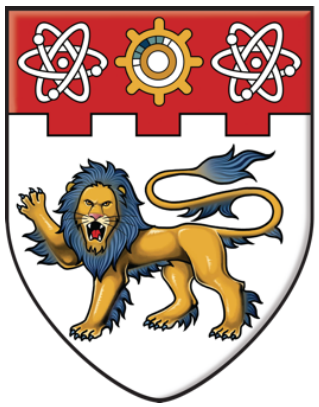


NANYANG TECHNOLOGICAL UNIVERSITY

SC4061/CE4003/CZ4003: Computer Vision

Lab 1 Report

Point Processing + Spatial Filtering + Frequency Filtering + Imaging Geometry

Contents

1. Introduction

1.1 Objectives

This laboratory aims to introduce image processing in MATLAB context. In this laboratory you will:

a. Become familiar with the MATLAB and Image Processing Toolbox software package.
b. Experiment with the point processing operations of contrast stretching and histogram equalization.
c. Evaluate how different Gaussian and median filters are suitable for noise removal.
d. Become familiar with the frequency domain operations e. Understand imaging geometry.

2. Experiments

2.1 Contrast Stretching

```
%% 2.1 Contrast Stretching
%a
Pc = imread('mrt-train.jpg');
whos Pc;
P = rgb2gray(Pc);

%b
figure;
imshow(P);
title('Original Image');

%c
minIntensity = double(min(P(:)));
maxIntensity = double(max(P(:)));
fprintf('Minimum intensity is: %d\n', minIntensity);
fprintf('Maximum intensity is: %d\n', maxIntensity);
% Define the desired minimum and maximum intensities (0 and 255)
desiredMin = 0;
desiredMax = 255;

%d
% Apply contrast stretching formula
P2 = (double(P) - minIntensity)*255 / (maxIntensity - minIntensity);
P2 = uint8(P2);
minIntensity2 = min(P2(:));
maxIntensity2 = max(P2(:));
fprintf('Minimum intensity in stretched image is: %d\n', minIntensity2);
fprintf('Maximum intensity in stretched image is: %d\n', maxIntensity2);

%e
figure;
imshow(P2);
title('Contrast-Stretched Image');
```

The MATLAB code above provides step-by-step answers to all the tasks mentioned in Contrast Stretching. The code essentially implements the formula of contrast stretching which is given by:

$$s = \frac{255\left(r - r_{\min}\right)}{r_{\max} - r_{\min}}$$

The image has a size of 320x443x3 and so it needs to be converted to grayscale before we proceed. The image is initially in **uint8** datatype and needs to be changed to **double** for further operations like addition and subtraction.

Fig 1: Original Image

Fig 2: Contrast Stretched Image

The Fig 2 shows the output image after applying contrast stretching on Fig 1. Also notice how the minimum and maximum intensities of the image changes through the Command Window output shown below:

```
Minimum intensity is: 13
Maximum intensity is: 204
Minimum intensity in stretched image is: 0
Maximum intensity in stretched image is: 255
```

2.2 Histogram Equalization

The image intensity histograms of P with 10 bins and 256 bins have been provided below. Bins show the gray levels which are to be represented in a histogram. In Fig 3, the total number of pixels given by the entire picture which is 320 x 443 will be spread across these 10 bins. Similarly, the histogram with 256 bins (Fig 4) has 256 different available gray levels for each pixel of the picture. The difference is that when we have only 10 bins, the frequency counts for pixel values can go to the order of 10^4 but with 256 bins, the maximum frequency is only up to 2500. So the histogram with 10 bins feels more equalized than the histogram with 256 bins.

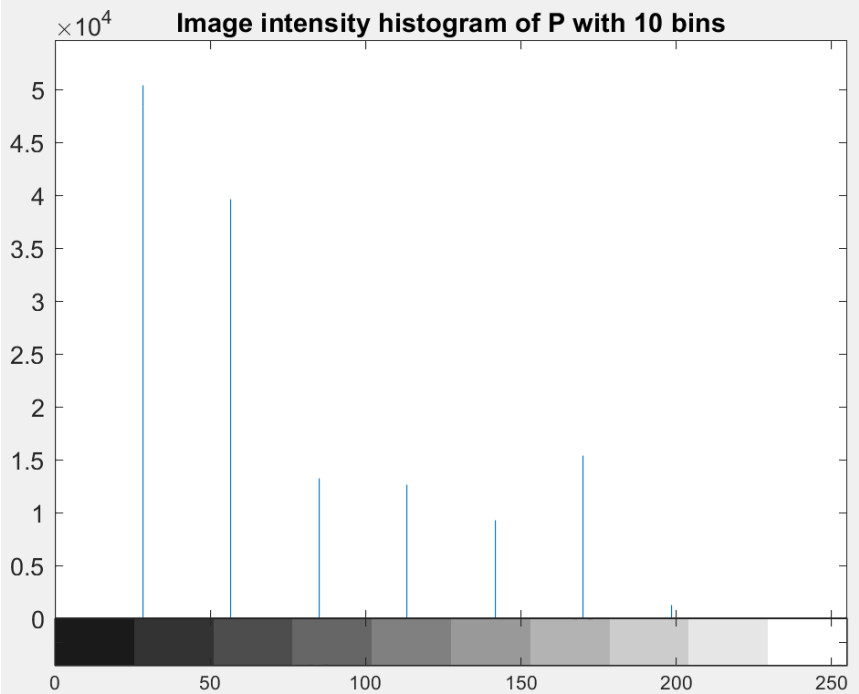


Fig 3: Image Intensity Histogram of P with 10 bins

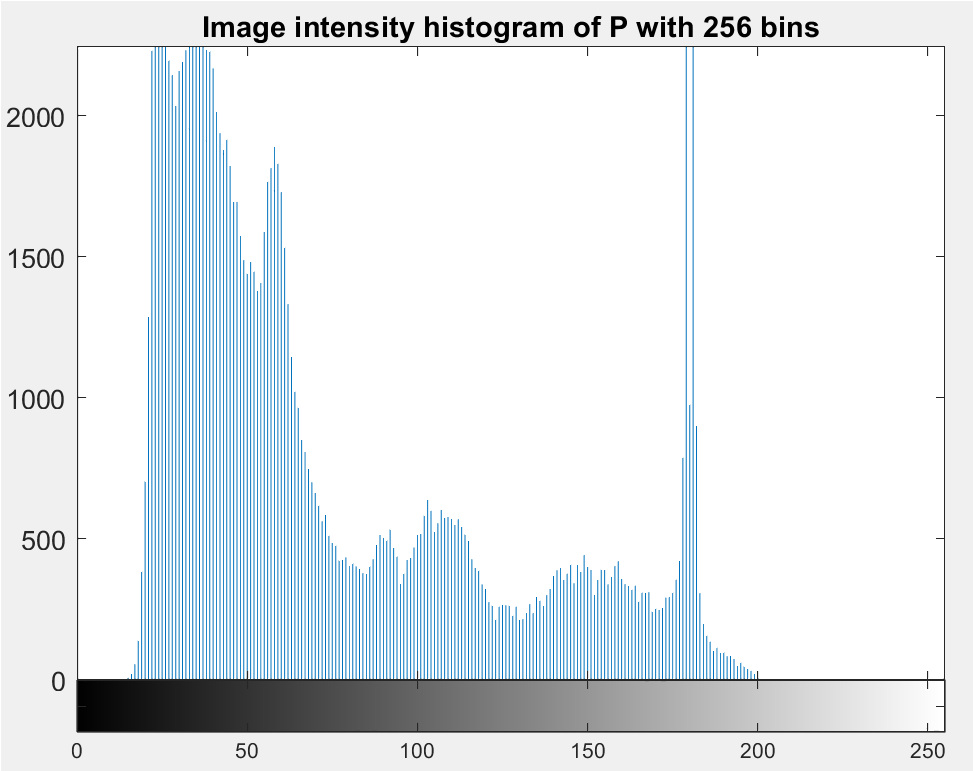


Fig 4: Image Intensity Histogram of P with 256 bins

Now, we carry out histogram equalization as follows: >> P3 = histeq(P,255);

And then generate the intensity histogram of P3 with 10 bins and 256 bins. The outputs are shown as Fig 5 and Fig 6 respectively.

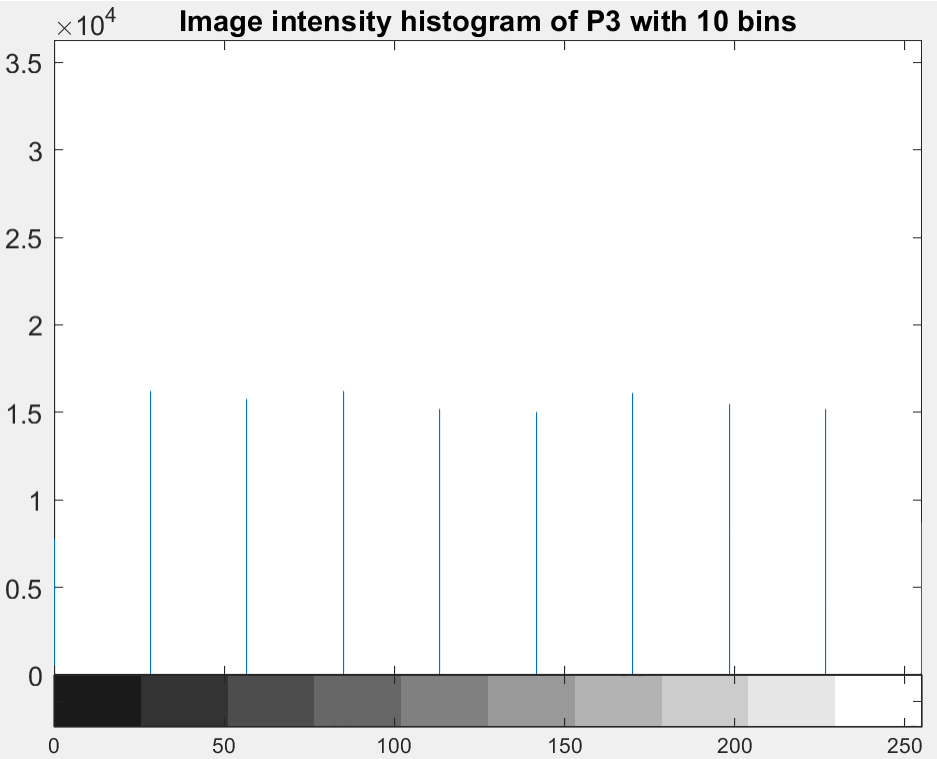


Fig 5: Image Intensity Histogram of P3 with 10 bins

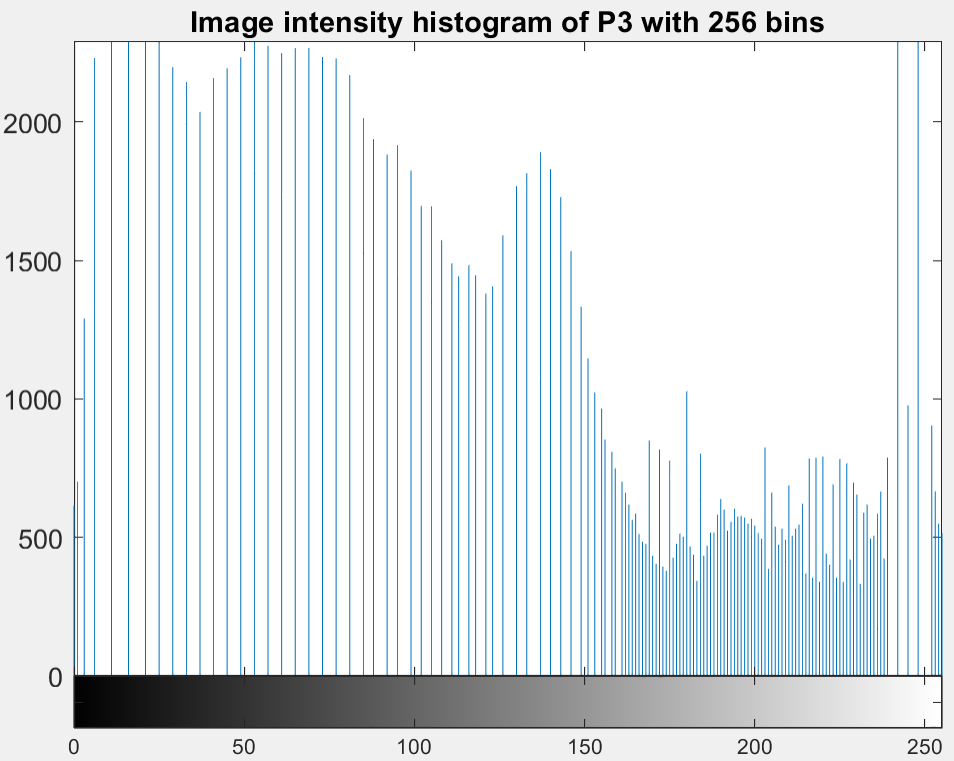


Fig 6: Image Intensity Histogram of P3 with 256 bins

Histogram equalization is done with the following code shown below. With this, the similarities are that the total number of pixels are spread across the 10 and 256 bins. But the difference is that the 10 bins are more equalized than the 256 bins.

Rerunning the histogram equalization on P3 (assigning the new value as P4) does not change the output. Therefore, the histogram does not become more uniform as shown by Fig 7 and Fig 8. Due to it having the same total number of pixels as well as the same number of bins spread across the probabilities, it will just follow the formula and place the pixels in each bin similar to the previous calculated output bin.

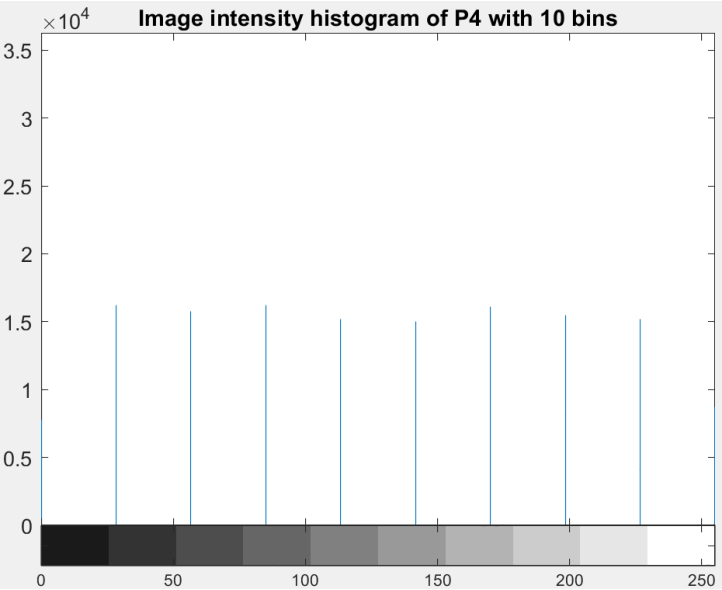


Fig 7: Image Intensity Histogram of P4 with 10 bins

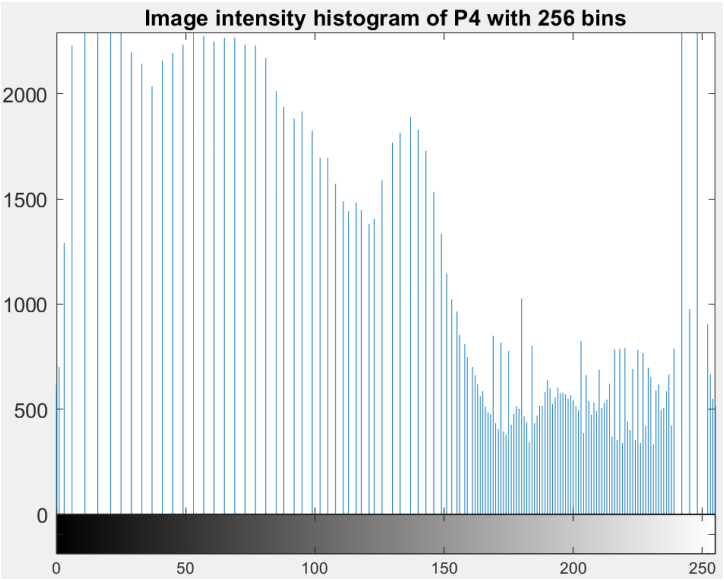


Fig 8: Image Intensity Histogram of P4 with 256 bins

2.3 Linear Spatial Filtering

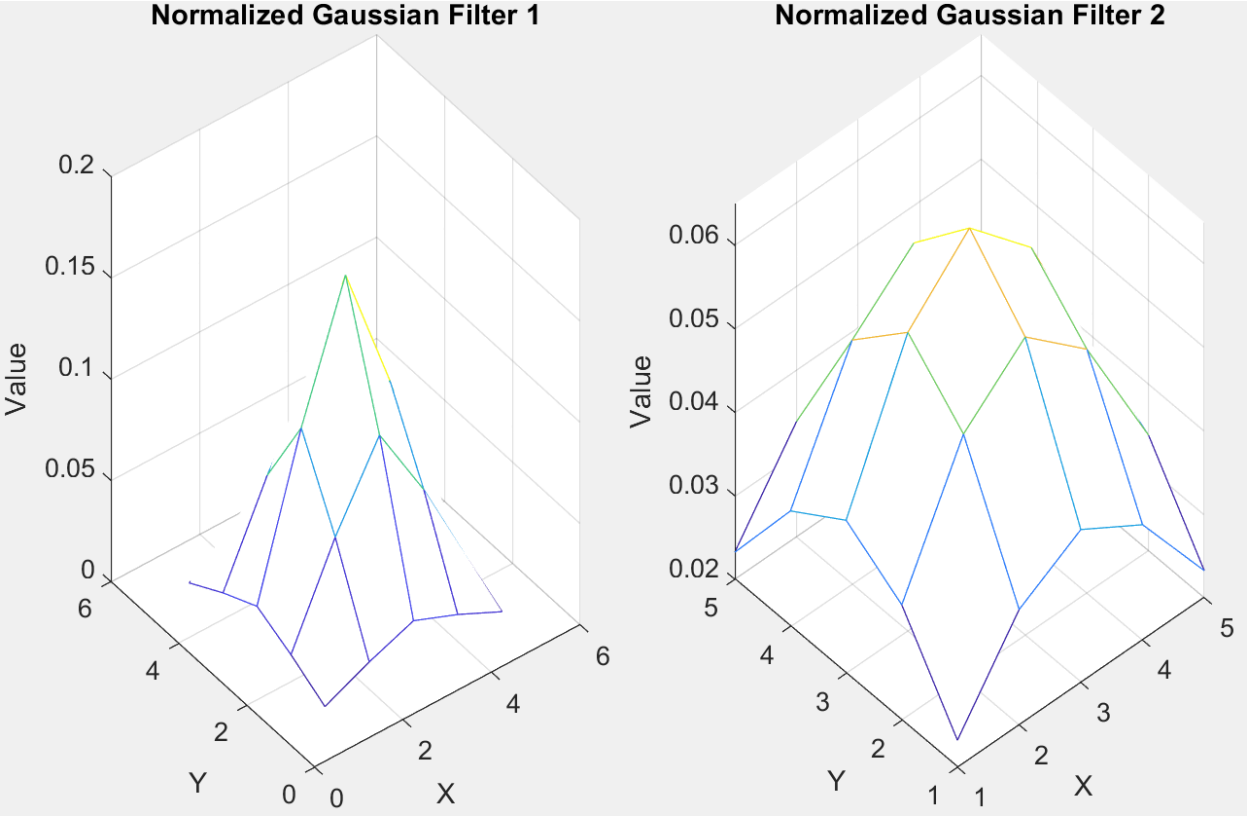
Firstly, the gaussian filters which are mathematically represented as,

$$h(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{x^2+y^2}{2\sigma^2}}$$

are represented in MATLAB through the code below:

```
sigma1 = 1;  % Standard deviation for the first Gaussian filter
sigma2 = 2;  % Standard deviation for the second Gaussian filter
filter_size = 5;  % Size of the filter kernel (e.g., 5x5)

% Create Gaussian filter kernels
h1 = 1 / (2 * pi * sigma1^2) * fspecial('gaussian', filter_size, sigma1);
h2 = 1 / (2 * pi * sigma2^2) * fspecial('gaussian', filter_size, sigma2);

% Normalize the filters
h1_normalized = h1 / sum(h1(:));
h2_normalized = h2 / sum(h2(:));
```

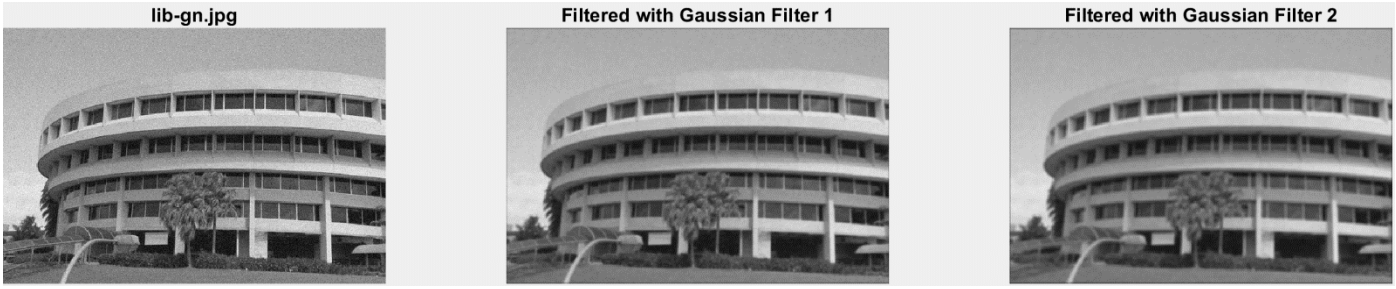
Next, we view the 3D graphs of these filters using the **mesh** function. The graphs for both filters are shown below:



Next, we download the 'lib-gn.jpg' image. It has additive Gaussian noise. The below picture summarizes the original image and the output when both filters are separately applied to the original image using the **conv2** function as shown in the code snippet below.

```
% Filter the image using the first normalized Gaussian filter
filtered_img1 = conv2(double(Pc2), h1_normalized, 'same');

% Filter the image using the second normalized Gaussian
filter
filtered_img2 = conv2(double(Pc2), h2_normalized, 'same');
```

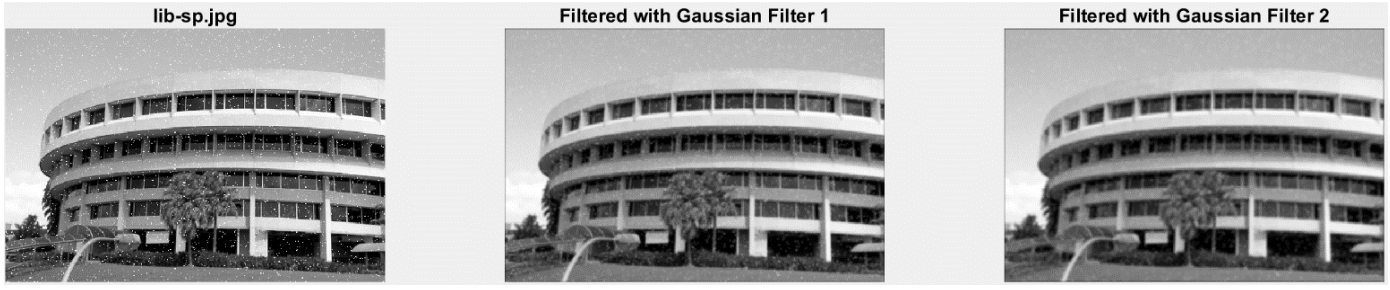
Next, we proceed to display the images as shown below.



The filters are quite effective in removing the noise. The noise is even more less obvious when the value of sigma increases.

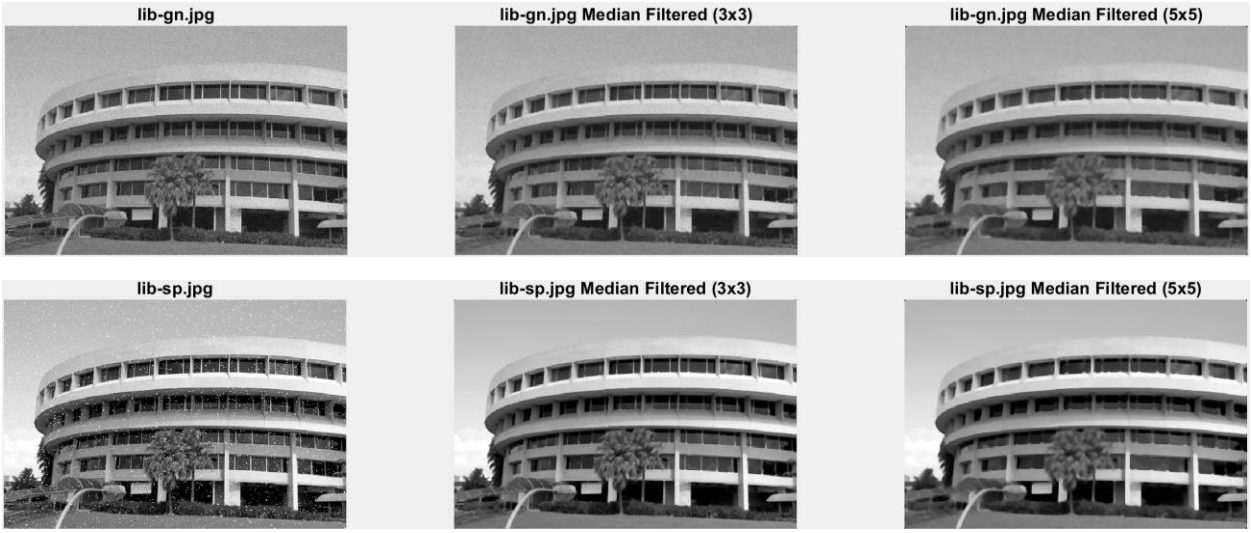
The trade-offs include the loss of details with increasing of sigma. The higher the sigma, the higher is the blur filter. The original image is much sharper and has more details.

The same process is repeated for 'lib-sp.jpg' image. This image has additional speckle noise. Next, we apply both the filters using **conv2** function. On close inspection, we realize that the speckles are not entirely gone. Therefore, we can conclude that the gaussian filter is not good at removing speckle noise. The outputs for 'lib-sp.jpg' image are shown in the picture below:



2.4 Median Filtering

Now, we perform median filtering on both 'lib-gn.jpg' and 'lib-sp.jpg'. Median filtering is a special case of order-statistic filtering. It involves replacing the target pixel with the median pixel intensity. It can be easily done by using **medfilt2** instead of **conv2**. We need to perform median filtering using 2 filter sizes: [3x3] and [5x5].



The speckles are entirely removed from the image which suggests that median filter is better than Gaussian averaging filters are removing speckles.

Comparison between Gaussian filters and Median filters:

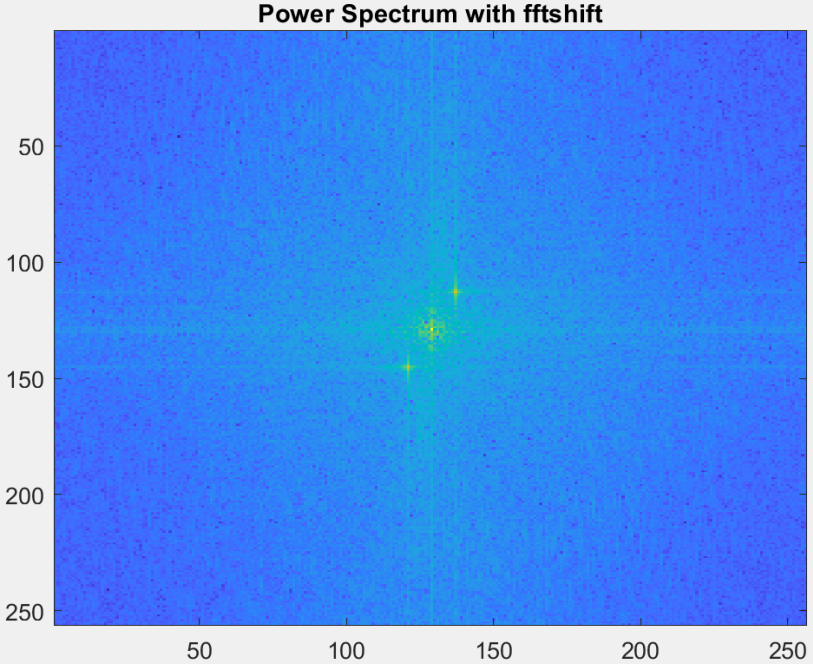
Closely observing the image outputs, we notice that median filters perform very well in removing 'salt-and-pepper' noise while Gaussian filters just blurs the image to reduce noise. Due to this blurring effect by Gaussian filters, the edges lose their sharpness and become harder to detect as the sigma value increases. On the other hand, median filters preserve the edge detail better than Gaussian filters.

2.5 Suppressing Noise Interference Patterns

In this experiment, we will be working on removing periodic noise from an image. The original image 'pck-int.jpg' looks like the image shown below:

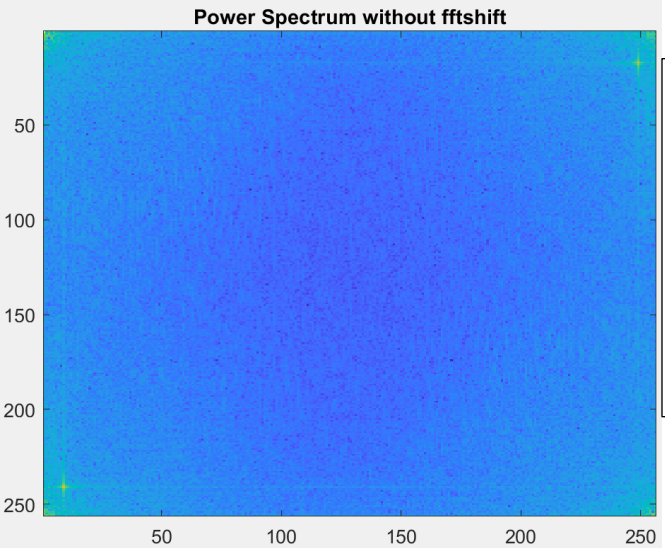

```
%% 2.5 Suppressing Noise Interference
Patterns
%a
Pc4 = imread('pck-int.jpg');
whos Pc4;
figure;
imshow(Pc4);
title('pck-int.jpg');
```

Periodic noise is basically repeated patterns that seems to have been added on top of the original image. First we compute the Fast Fourier Transform of the image using **fft2** and further generate the power spectrum using **imagesc** and **fftshift.** The image of power spectrum and the code snippet are provided below:




```
%b
% Compute the Fourier transform of the image
F = fft2(double(Pc4));
S = abs(F);
figure;
imagesc(fftshift(S.^0.1));
title('Power Spectrum with fftshift');
colormap('default');
```

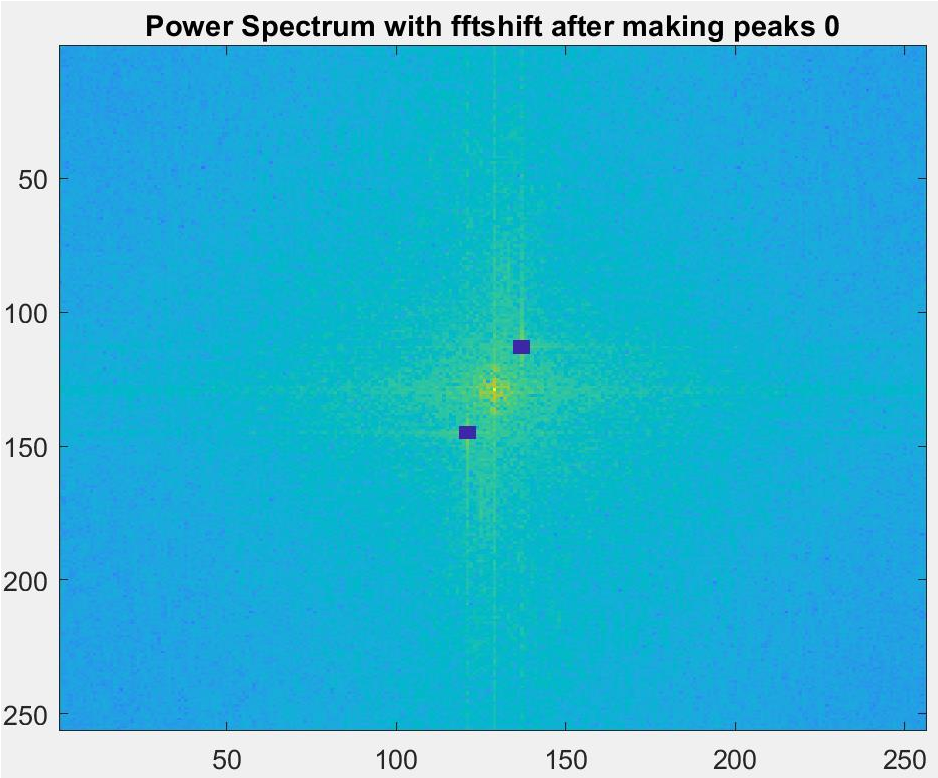
Next, let's try to view the power spectrum without using **fftshift.**




```
%c
figure;
imagesc(S.^0.1);
title('Power Spectrum without fftshift');
colormap('default');

%coordinates of peaks
x1 = 249;
y1 = 17;
x2 = 9;
y2 = 241;
```

Next, we try to remove the peaks using the coordinates provided in the code snipped above. We take a margin of ±2 from each coordinate and assign it the value 0. This basically makes a 5x5 neighborhood which gets assigned to 0. The code snippet as well as the power spectrum after making the peaks 0 is shown below:

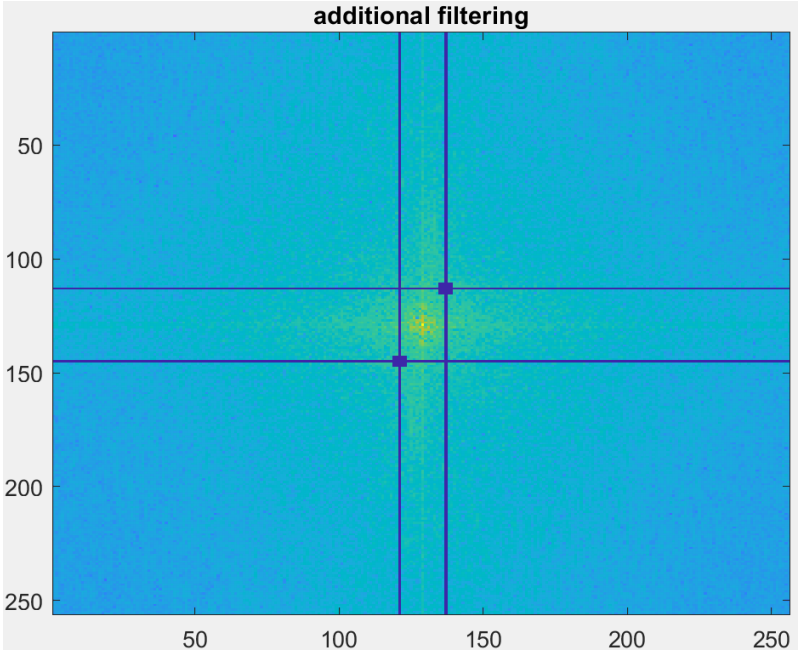
```
%d
F(y1-2 : y1+2, x1-2 : x1+2) = 0;
F(y2-2 : y2+2, x2-2 : x2+2) = 0;
S = abs(F);
figure;
imagesc(fftshift(S.^0.1));
title('Power Spectrum with fftshift after making peaks 0');
colormap('default');
```



After the peaks have been assigned to 0, we can compute the Inverse Fast Fourier Transform of the image using **ifft2**. The output image and code snippet are shown below.

```
%e
img = uint8(ifft2(F));
figure;
imshow(img)
title('Inverse Fast Fourier Transform (IFFT)');
```

Now, we need to try to further improve the picture and make it clearer. We can try that by assigning the whole rows and columns i.e., the x-axis and y-axis of the coordinates we used above to 0. The code implementation, the associated power spectrum and the output image are provided below:

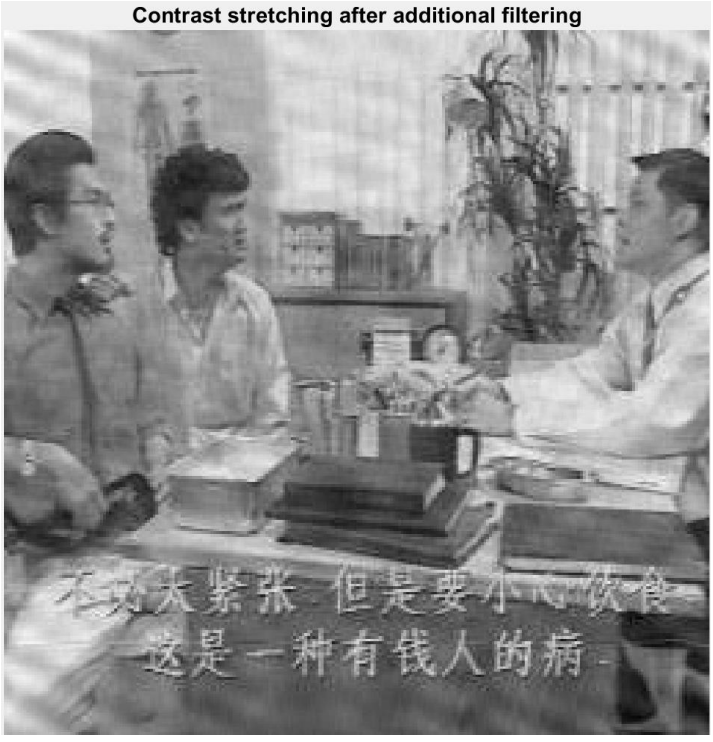



```
% Additional Filtering
F(y1, :) = 0;
F(y2, :) = 0;
F(:, x1) = 0;
F(:, x2) = 0;
S = abs(F);
figure;
imagesc(fftshift(S.^0.1));
title('additional filtering');
img = uint8(ifft2(F));
figure;
imshow(img);
title('IFFT after additional filtering');
```

IFFT after additional filtering

While the periodic noise has mostly been removed, the image can further be enhanced with contrast stretching as shown below:

Contrast stretching after additional filtering

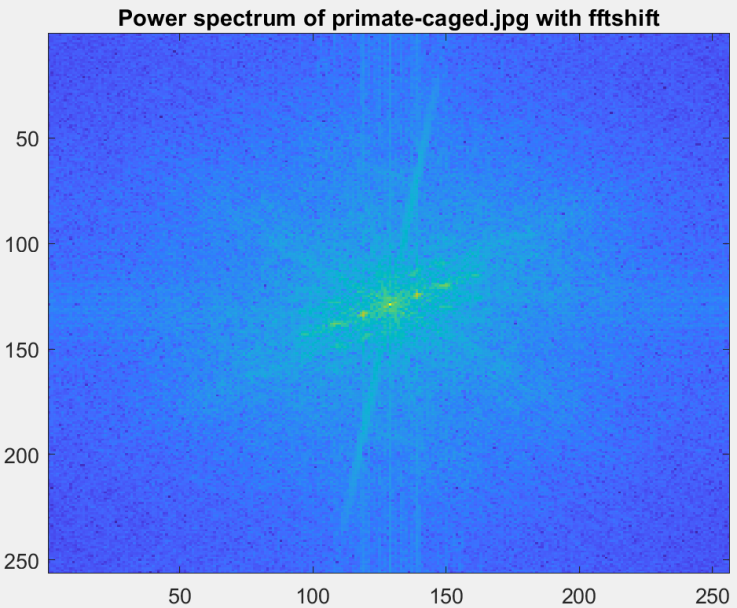


```
% Contrast Stretching
r_min = double(min(img(:)));
r_max = double(max(img(:)));
img = uint8(255 * (double(img) - r_min) / (r_max - r_min));
figure;
imshow(img);
title('Contrast stretching after additional filtering');
```

Now we need to work on 'primate-caged.jpg' which shows a primate behind a vague. We will attempt to use image processing to remove the cage. The original picture is shown below.

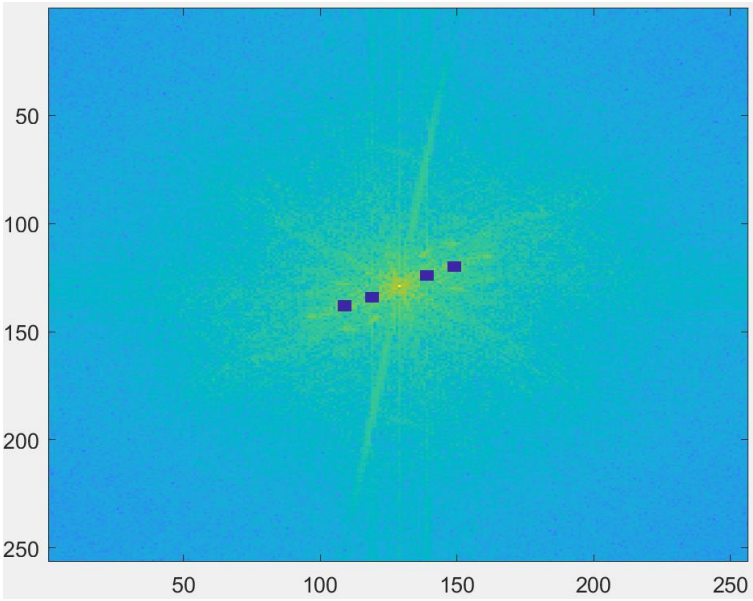

```
%f
Pc5 = imread('primate-
caged.jpg');
whos Pc5;
Pc5 = rgb2gray(Pc5);
figure;
imshow(Pc5);
title('primate-caged.jpg');
```

First, we convert the image into frequency domain, perform **fftshift** and then generate the power spectrum to see where the peaks are located. The power spectrums with fft2 and without fft2 have been shown below:




```
F = fft2(Pc5);
S = abs(F);
figure;
imagesc(fftshift(S.^0.1));
title('Power spectrum of primate-caged.jpg with fftshift');
```

Next, we try to remove the peaks by assigning them to zero. Following the same approach used for 'pck-int.jpg', the output and code snippet are. shown below.




```
x1 = 11;
y1 = 252;
x2 = 247;
y2 = 6;
x3 = 21;
y3 = 248;
x4 = 237;
y4 = 10;
F(y1-2 : y1+2, x1-2 : x1+2) = 0;
F(y2-2 : y2+2, x2-2 : x2+2) = 0;
F(y3-2 : y3+2, x3-2 : x3+2) = 0;
F(y4-2 : y4+2, x4-2 : x4+2) = 0;
S = abs(F);
figure;
imagesc(fftshift(S.^0.1));
```

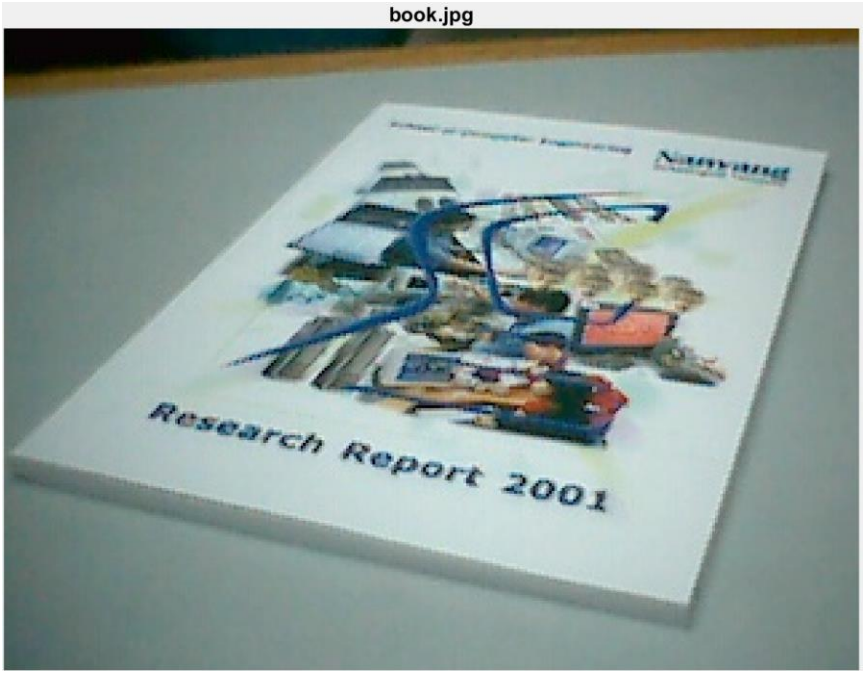
After removing the peaks, we can g using **ifft2**. The output and code are shown below. From the final result, we can see that although the fence is removed. It is hard to achieve and clear result as the lines from the cage are not straight and parallel.


```
% Display new image
img = uint8(ifft2(F));
figure;
imshow(img)
title('Final Result');
```

2.6 Undoing Perspective Distortion of Planar Surface

In this experiment, the goal is to take an original slanted view of a book, and compute the projective transformation to warp the image to a frontal view. The final warped image should have the scale 1 pixel (image) = 1 mm (book dimension).

The image and the code to read the image is shown below:




```
%a
Pc6 = imread('book.jpg');
whos Pc6;
figure;
imshow(Pc6);
title('book.jpg');
```

Next, we try to retrieve the four corners of the book. It can be done through the **ginput** function like this: **>> [X Y] = ginput(4).** For maintaining homogeneity in experiment result, we have hardcoded the values of **X** and **Y** from one of the iterations of this experiment as shown by the code snippet below.

```
% [X Y] = ginput(4);
% For replicating experiment
X = [143.0000 309.0000 7.0000 255.0000];
Y = [28.0000 48.0000 160.0000 213.0000];
```

Next, we define image vectors. The inputs are in this format for these vectors so that they follow the dimensions of A4-size paper and factor in the manner in which ginput was taken.

Top Left Corner: (0,0),
Top Right Corner: (210,0),
Bottom Left Corner: (0, 297),
Bottom Right Corner: (210, 297).

```
%b
imageX = [0 210 0 210];
imageY = [0 0 297 297];
```

Next, we define the matrices needed to perform projective transformation based on the equation:

```
>> u = A \ v;
```

The above computes $u = A^{-1}v$, and we can convert the projective transformation parameters to the normal matrix form by:

```
>> U = reshape([u;1], 3, 3);
```

Next, we need to verify that this is correct by transforming the original coordinates by:

```
>> w = U*[X; Y; ones(1,4)];
```

```
>> w = w ./ (ones(3,1) * w(3,:))
```

The code implementation for the above procedures is provided below:

```
%c
A = [[X(1), Y(1), 1, 0, 0, 0, -imageX(1)*X(1), -imageX(1)*Y(1)];
     [0, 0, 0, X(1), Y(1), 1, -imageY(1)*X(1), -imageY(1)*Y(1)];
     [X(2), Y(2), 1, 0, 0, 0, -imageX(2)*X(2), -imageX(2)*Y(2)];
     [0, 0, 0, X(2), Y(2), 1, -imageY(2)*X(2), -imageY(2)*Y(2)];
     [X(3), Y(3), 1, 0, 0, 0, -imageX(3)*X(3), -imageX(3)*Y(3)];
     [0, 0, 0, X(3), Y(3), 1, -imageY(3)*X(3), -imageY(3)*Y(3)];
     [X(4), Y(4), 1, 0, 0, 0, -imageX(4)*X(4), -imageX(4)*Y(4)];
     [0, 0, 0, X(4), Y(4), 1, -imageY(4)*X(4), -imageY(4)*Y(4)];];

v = [imageX(1); imageY(1); imageX(2); imageY(2); imageX(3); imageY(3); imageX(4); imageY(4)];

u = A \ v;
U = reshape([u;1], 3, 3);
w = U*[X; Y; ones(1,4)];
w = w ./ (ones(3,1) * w(3,:));
```

The transformation does give back the 4 corners of the desired image. The values are like that of those that have been declared in vectors ImageX and imageY in part b.

After this, we need to warp the image and display the result as shown in the code snippet below:

```
%d
T = maketform('projective', U);
P2 = imtransform(Pc6, T, 'XData', [0 210], 'YData', [0 297]);

%e
figure;
imshow(P2);
```

The image output is great and beyond expectations. The quality is also satisfactory considering the quality of input image. The top of the image is however blurred but it is reasonable because the top section of the input image is unreadable too. This might be possible because of the resolution of the input image but overall, the performance is decent.

Now, we need to identify the large rectangular red area which looks like a computer screen. For this, we can create a binary mask using the code below:

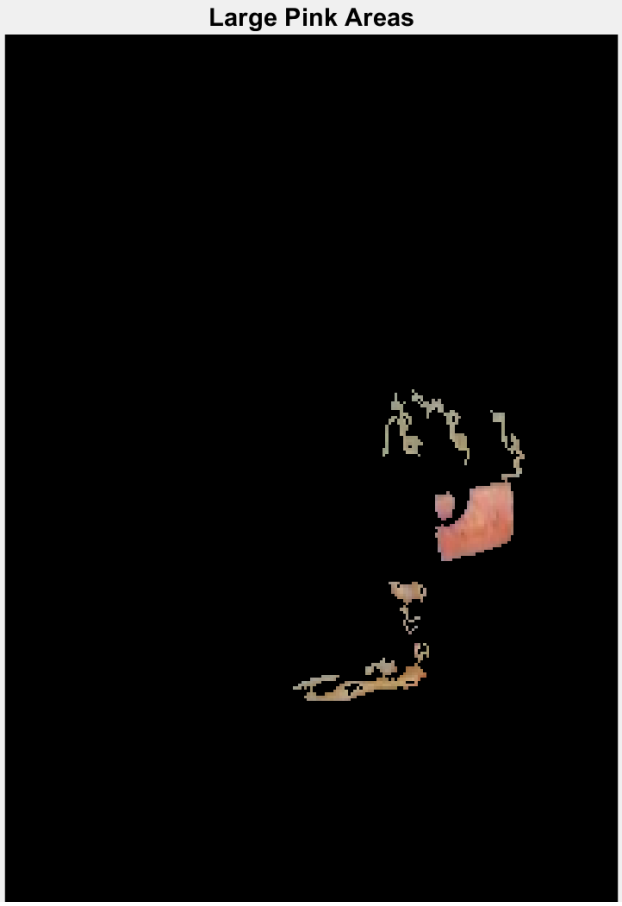
```
%f
whos P2;
% Define the target color range for pink
pinkLowerBound = [151, 90, 62];
pinkUpperBound = [255, 170, 154];

% Create a binary mask
binaryMask = (P2(:,:,1) >= pinkLowerBound(1) & P2(:,:,1) <= pinkUpperBound(1)) ...
             & (P2(:,:,2) >= pinkLowerBound(2) & P2(:,:,2) <= pinkUpperBound(2)) ...
             & (P2(:,:,3) >= pinkLowerBound(3) & P2(:,:,3) <= pinkUpperBound(3));

% Visualize the identified pink area
maskedImage = P2;
maskedImage(repmat(~binaryMask, [1 1 3])) = 0;
% Set non-pink pixels to black

figure;
imshow(maskedImage);
title('Identified Pink Area');
```

Variables pinkLowerBound, pinkUpperBound help us define the range of RGB values we will be using to create our mask. The variable binaryMask stores the condition about which pixels satisfy the criteria. Then with the maskedImage variable, we are able to generate the output of the binary mask as shown below:

While the rectangular pink shape is clearly visible, there are more pixels which satify the condition of binary mask. We will try to remove them using **regionprops** function. In this, we identify pink areas and try to eliminate all pink areas which are smaller than the minimum pink area threshold. With trial and error, we came up with a value of 50. After this, we proceed to display the new image. The implementation for this procedure is shown in the code below:

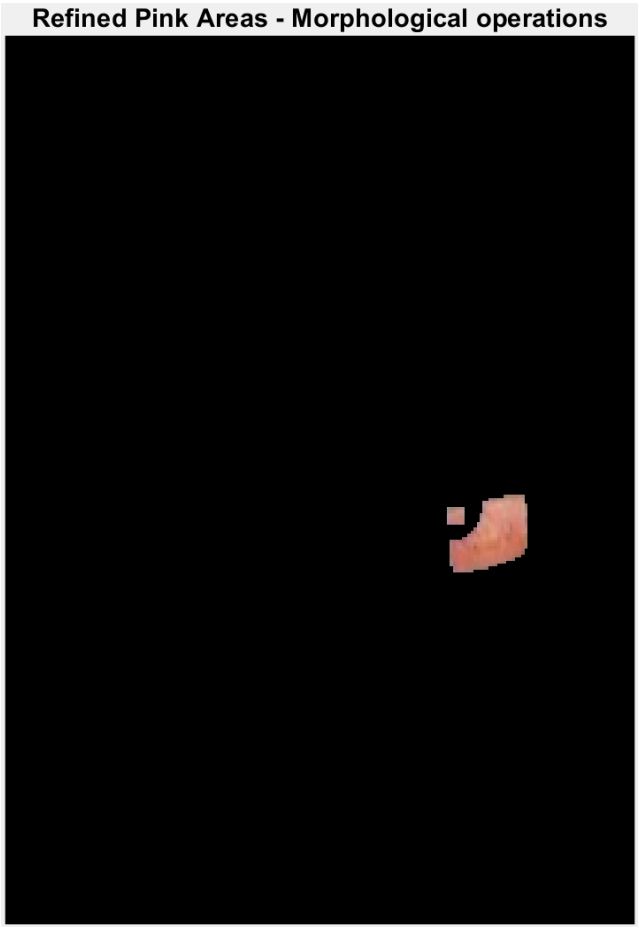
```
% Label connected components in the binary mask
pinkLabelMatrix = bwlabel(binaryMask);

% Analyze the properties of connected components
pinkStats = regionprops(pinkLabelMatrix, 'Area');

% Define a threshold for the minimum area (adjust as needed)
minPinkAreaThreshold = 50;

% Create a binary mask to keep only large pink regions
largePinkBinaryMask = ismember(pinkLabelMatrix, find([pinkStats.Area] >= minPinkAreaThreshold));

% Filter the maskedImage to keep only large pink regions
largePinkImage = maskedImage;
largePinkImage(repmat(~largePinkBinaryMask, [1 1 3])) = 0; % Set non-large pink pixels to black

% Display the result
figure;
imshow(largePinkImage);
title('Large Pink Areas');
```

The output for the above code is:



The new image removes a lot of the irrelevant pink areas from the previous image but further improvements could be done to eliminate the remaining red areas which don't form a part of the rectangle. This can further be fixed by using morphological operations.

The morphological operations include **imerode** (helps to remove small artifacts based on their shape) and **imdilate** (helps to dilate/expand the foreground objects in an image). We will also need to define a structuring element which uses the shape of a rectangle. The dimensions of this rectangular structuring element have been found by trial and error to give the best possible result i.e., pink areas only from the computer screen. The code and output image for this procedure is provided below:



Now, we can safely conclude that we have identified the large pink rectangular area.

```
% Perform morphological operations
% Define a rectangular structuring element (adjust the size as needed)
se = strel('rectangle', [6, 6]);

% Erosion to remove small artifacts (optional)
erodedMask = imerode(binaryMask, se);

% Dilation to expand the pink region back to the original size
dilatedMask = imdilate(erodedMask, se);

% Use the dilated mask to keep only the pink regions
resultImage = P2;
resultImage(repmat(~dilatedMask, [1 1 3])) = 0; % Set non-pink pixels to black

% Display the result
figure;
imshow(resultImage);
title('Refined Pink Areas - Morphological operations');
```