2.1 Contrast Stretching

Input the image into a MATLAB matrix variable by executing:

```
Pc = imread('Images/MRT Train.jpg');
whos Pc
```

In this case my MRT Train image is placed in a folder called *Images*, hence there is an additional *Images/* in the filepath.

This returns:

```
Name         Size                 Bytes  Class     Attributes

Pc         320x443x3             425280  uint8
```

The *whos* command will show whether the image is read as an RGB or gray-scale image. In this case, *Pc*'s size is three-dimensional, which means that *Pc* is an RGB image, and we need to convert it to a gray-scale image by executing:

```
P = rgb2gray(Pc);
```

We will view the image with *imshow*:

```
imshow(P)
```

This returns:

The image has poor contrast, and hence we will need to improve the image appearance using point processing.

First, we check the minimum and maximum intensities present in the image:

```
min(P(:)), max(P(:))
```

This returns:

```
ans =

  uint8

    13
```

```
ans =

  uint8

   204
```

The minimum intensity is 13 and the maximum intensity is 204.

We will then do contrast stretching, which involves linearly scaling the gray levels such the smallest intensity present in the image maps to 0, and the largest intensity maps to 255:

```
P2 = (255-0)/(204-13)*imsubtract(P,13);
```

We take 255 multiplied by the (image - minimum intensity) divided by the (maximum intensity - minimum intensity).

Do note that the usage of operators such as *imadd* and *imsubtract* requires the installation of Image Processing Toolbox on MATLAB.

We can also convert to a double-valued matrix first, and convert it back to *uint8* prior to using *imshow*:

```
P2_double = (255-0)/(204-13)*(double(P)-13);
P2_double = uint8(P2_double);
```

We then check to see if our final image has the correct minimum and maximum intensities of 0 and 255:

```
min(P2(:)), max(P2(:))
```

```
min(P2_double(:)), max(P2_double(:))
```

Both return the same result:

```
ans =

  uint8

    0
```

```
ans =

  uint8

   255
```

This means that our final image has the correct minimum and maximum intensities of 0 and 255.

We redisplay the image by executing:

```
imshow(P2)
```

```
imshow(P2_double)
```

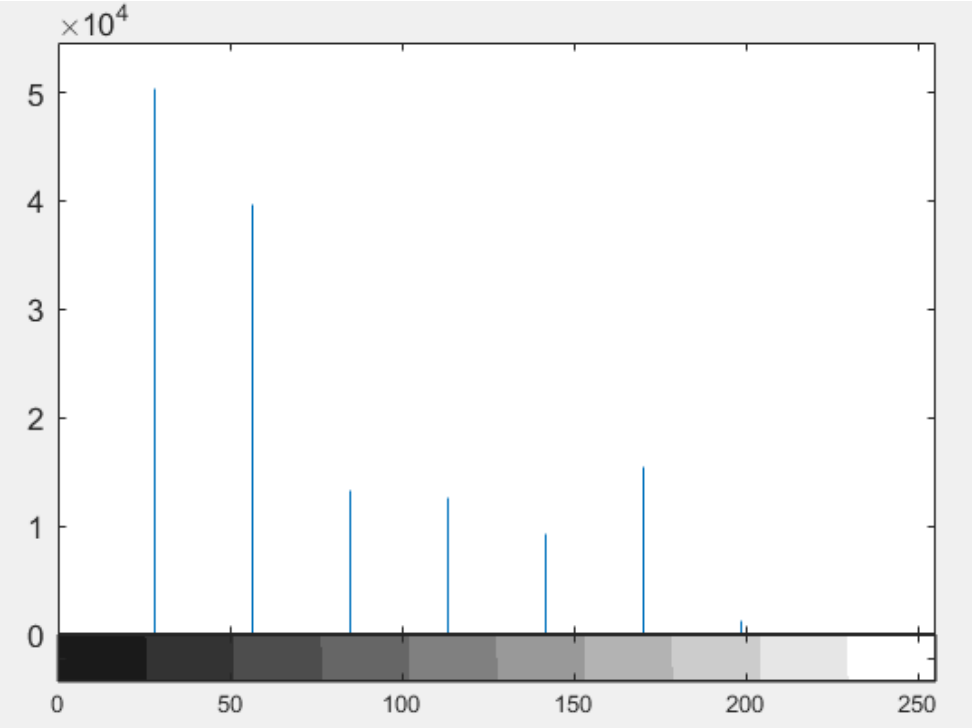
Both return the same result:

2.2 Histogram Equalization

We first display the image intensity histogram of P using 10 bins:

```
imhist(P,10);
```

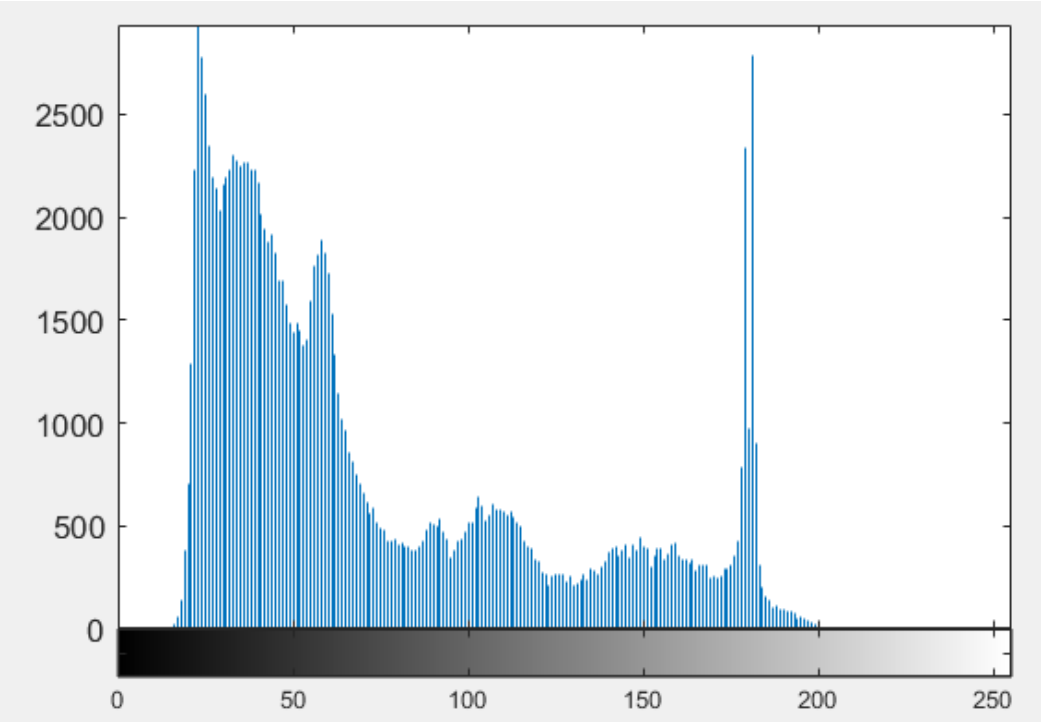
This returns:



We then display a histogram with 256 bins:

```
imhist(P,256);
```

This returns:



The difference between the two histograms is the distribution of the gray levels. The histogram with 256 bins has more details about the distribution than the histogram with 10 bins. In the histogram with 256 bins, we can see the intensity of every gray level as there are 256 gray levels (0-255) in a gray-scale picture. When we have more bins in a histogram, the average number of pixels in each bin becomes lesser and more detailed information about the image can be obtained. For example, there is an exceptionally high number of pixels in the 170-180 bins for the histogram with 256 bins, but this observation is not noticeable in the histogram with 10 bins.

Next, we carry out histogram equalization:

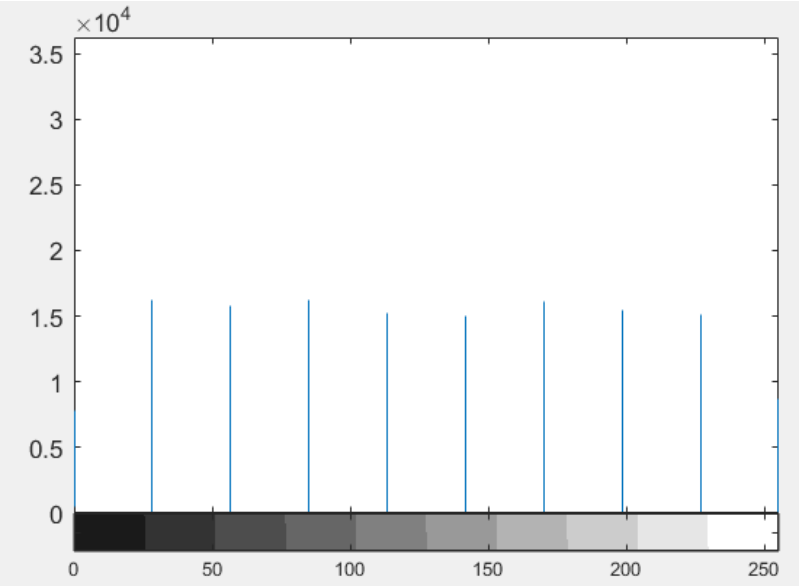
```
P3 = histeq(P,255);
```

We then display a histogram of P3 with 10 bins:

```
imhist(P3,10);
```

This involves nonlinearly mapping the gray levels to 255 discrete gray levels, in order to create a resultant histogram which is approximately uniform.

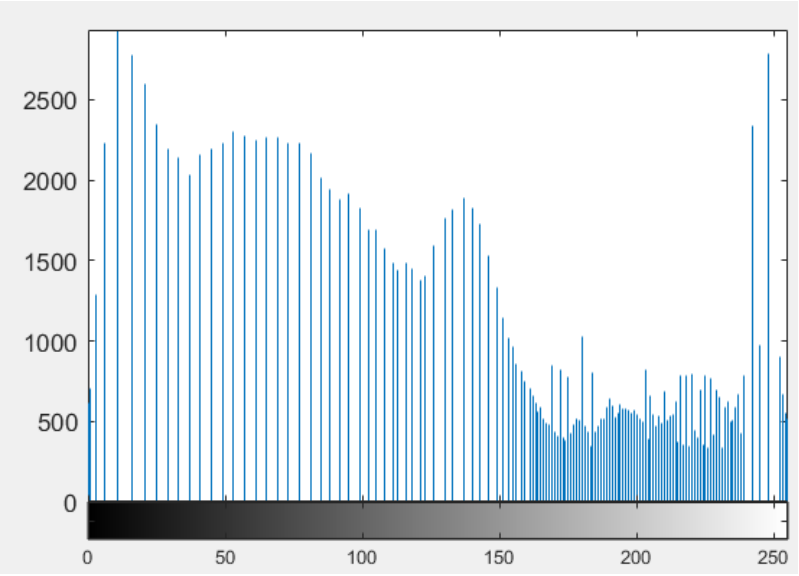
This returns:



We also display a histogram of P3 with 256 bins:

```
imhist(P3,256);
```

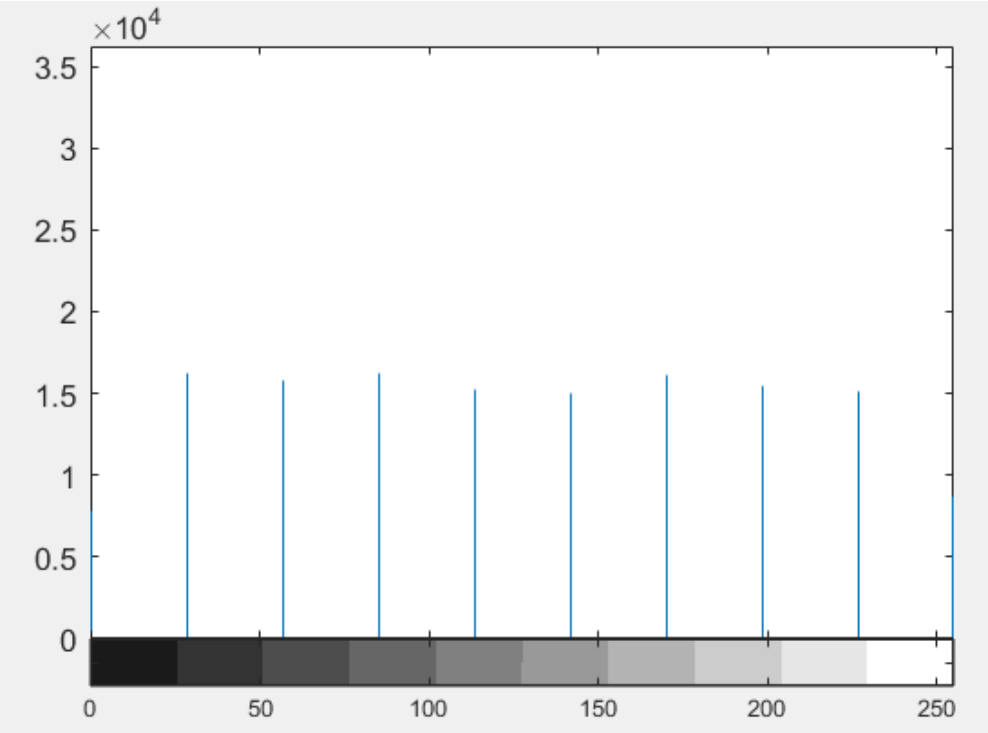
This returns:



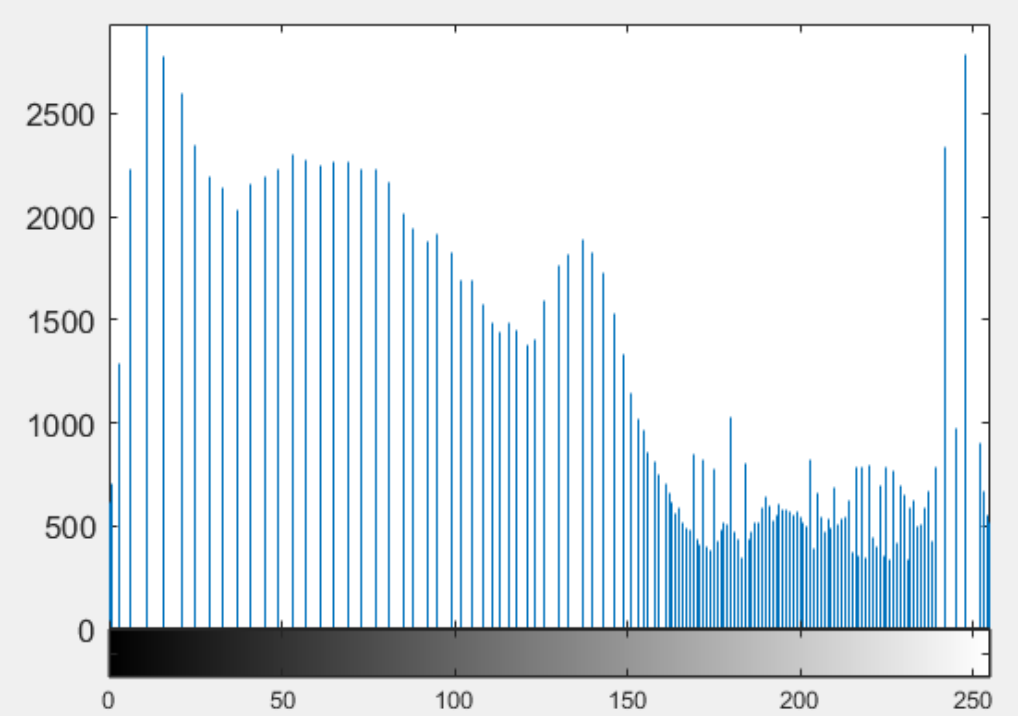
Both histograms are more uniform/flatter. The histogram with 256 bins is sparser as some bins are combined. Histogram equalization works by calculating the average number of pixels in each bin, followed by combining bins with the aim of all bins having number of pixels that are close to the average. It works in a way such that it can only entirely combine one bin with another (and not split one bin to different bins as it has no way of deciding which pixels in a same gray level should be brighter and which should be darker). As a result, the histogram with 256 bins looks more staggered as it is hard to perform perfect histogram equalization with the rule mentioned above. The histogram with 10 bins looks more uniform as each bin is looking at a range of gray levels instead of just one gray level, and the number of pixels should be more similar.

We then rerun the histogram equalization on P3 using the same commands.

The histogram with 10 bins looks like:



The histogram with 256 bins looks like:



There is no difference in the histograms. After running the first equalization, the pixels are already mapped to their new gray levels based on the average number of pixels. Hence, applying equalization again will not cause any change as there is no more possible combinations of bins.

2.3 Linear Spatial Filtering

We create symbolic variables *x* and *y*:

```
syms x y % x = sym('x'); y = sym('y');
```

Note that *syms* requires the Symbolic Math Toolbox.

We then create function *h(x,y)* with different values of sigma (h1 corresponds to (i) and h2 corresponds to (ii)):

```
h1(x,y) = 1/(2*pi*(1.0^2))*exp(-(x^2+y^2)/(2*(1.0^2))); %(i) sigma = 1.0
h2(x,y) = 1/(2*pi*(2.0^2))*exp(-(x^2+y^2)/(2*(2.0^2))); %(ii) sigma = 2.0
```

We find the maximum *x* and *y* distances (based on *x* and *y* dimensions of 5):

```
xMax = floor(5/2);
yMax = floor(5/2);
```

We then make h1 and h2 into matrices and normalize the filters:

```
[x,y] = meshgrid(-xMax:1:xMax, -yMax:1:yMax);
m1 = h1(x,y);
m1 = m1/sum(m1(:)); %normalise
sum1 = double(sum(m1(:)))
m1 = double(m1)
m2 = h2(x,y);
m2 = m2/sum(m2(:)); %normalise
sum2 = double(sum(m2(:)))
m2 = double(m2)
```

We should normalize such that the sum of all elements equals to 1, which we can check using *sum1* and *sum2*:

```
sum1 =

     1
```

```
sum2 =

     1
```

The filters *m1* and *m2* are:

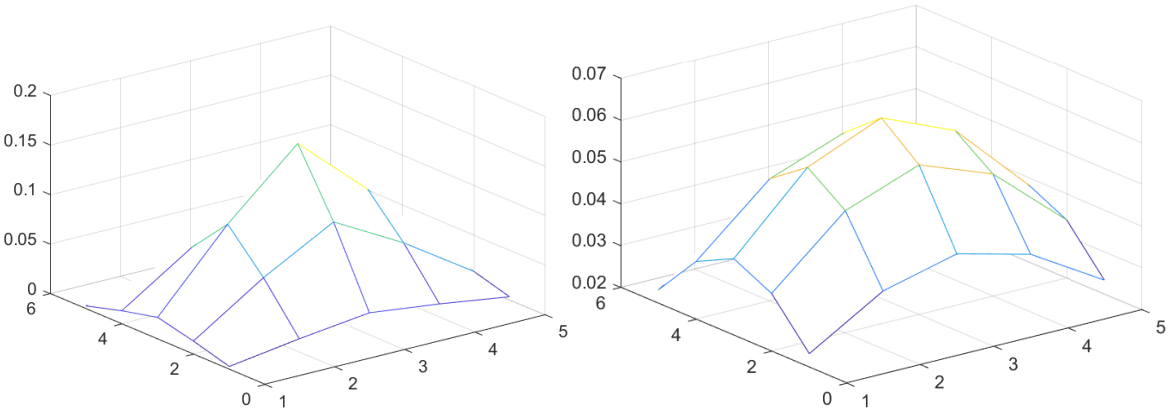
```
m1 =

    0.0030    0.0133    0.0219    0.0133    0.0030
    0.0133    0.0596    0.0983    0.0596    0.0133
    0.0219    0.0983    0.1621    0.0983    0.0219
    0.0133    0.0596    0.0983    0.0596    0.0133
    0.0030    0.0133    0.0219    0.0133    0.0030
```

```
m2 =

    0.0232    0.0338    0.0383    0.0338    0.0232
    0.0338    0.0492    0.0558    0.0492    0.0338
    0.0383    0.0558    0.0632    0.0558    0.0383
    0.0338    0.0492    0.0558    0.0492    0.0338
    0.0232    0.0338    0.0383    0.0338    0.0232
```

We then view the filters as 3D graphs:

```
mesh(m1)
mesh(m2)
```

m1 (left) and m2 (right):



We download an image and view it:

```
%Assume ntu_gn is 'Library with Gaussian Noise'
ntu_gn = imread('Images/Library with Gaussian Noise.jpg');
imshow(ntu_gn);
```

Filter the image using linear filters created above:

```
%m1
ntu_gn_m1 = conv2(m1, double(ntu_gn));
imshow(uint8(ntu_gn_m1));
```

```
%m2
ntu_gn_m2 = conv2(m2, double(ntu_gn));
imshow(uint8(ntu_gn_m2));
```

The filters are effective in removing noise but make the image more blurred (some details of the image are removed). m2 is more effective but the image is also more blurred. Gaussian averaging filters may work well for images with less edges, as there will be less of the smoothening effect that makes it look blurred. In this case, not filtering the image might be better.

Download another image and view it:

```
%Assume ntu-sp is 'Library with Speckle Noise'
ntu_sp = imread('Images/Library with Speckle Noise.jpg');
imshow(ntu_sp);
```

Repeat above steps with same commands for ntu_sp instead (left: m1, right: m2):


Both images were also blurred, but the speckle noise can still be seen on both images. m1 and m2 are Gaussian averaging filters and are better at handling at Gaussian noise.

2.4 Median Filtering

Using the same images we read in previously as *ntu_gn* and *ntu_sp*,

For the image with gaussian noise,

We now apply a 3x3 median filter:

```
ntu_gn_med33 = medfilt2(double(ntu_gn), [3 3]);
imshow(uint8(ntu_gn_med33));
```

Applying a 5x5 median filter:

```
ntu_gn_med55 = medfilt2(double(ntu_gn), [5 5]);
imshow(uint8(ntu_gn_med55));
```

For image with speckle noise (*ntu_sp*),

Applying 3x3 median filter:

```
ntu_sp_med33 = medfilt2(double(ntu_sp), [3 3]);
imshow(uint8(ntu_sp_med33));
```

Applying 5x5 median filter:

```
ntu_sp_med55 = medfilt2(double(ntu_sp), [5 5]);
imshow(uint8(ntu_sp_med33));
```

Median filter is able to reduce the Gaussian noise slightly, but also blurring the image. Gaussian averaging filters are significantly more effective in removing Gaussian noise but not so much in removing speckle noise. This is because Gaussian filters smoothen images, which will somewhat reduce spikes in gray levels (speckle noise) but not remove them completely. Median filters are however significantly more effective in removing speckle noise. This is because spikes (speckle noise) are removed completely. A median filter with a larger neighbourhood size smoothens the image more, making it more blurred.

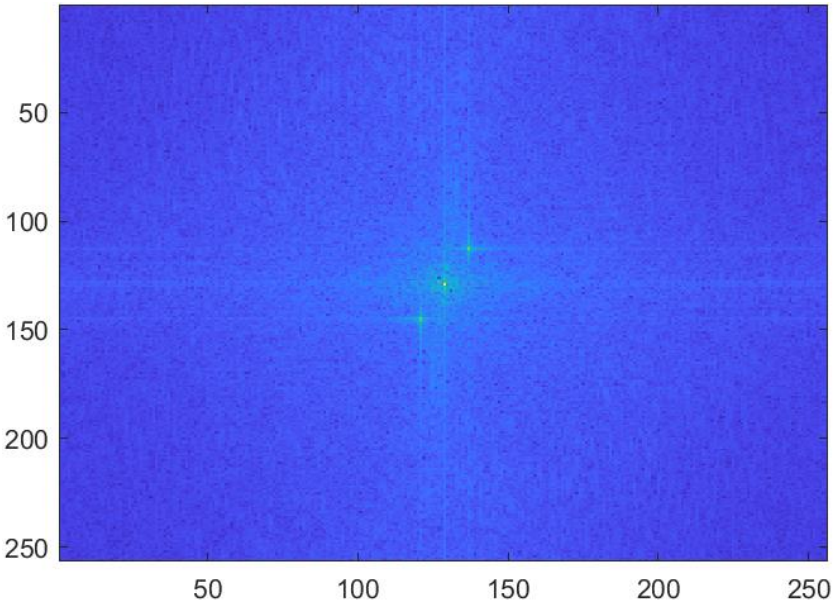
2.5 Suppressing Noise Interference Patterns

Downloading the image and displaying it:

```
pck = imread('Images/PCK with Channel Interference.jpg');
imshow(pck);
```

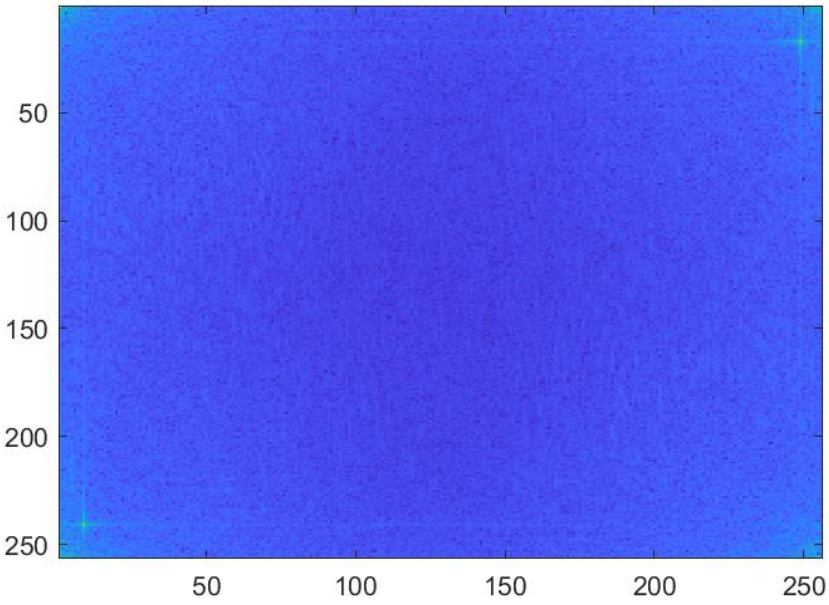
Obtain Fourier transform F and power spectrum S, and display S:

```
F = fft2(pck);
S = real(F).^2 + imag(F).^2;
imagesc(fftshift(S.^0.1));
colormap('default');
```



Redisplay power spectrum S without *fftshift*:

```
imagesc(S.^0.1);
```



Measure actual locations of peaks:

```
[x,y] = ginput
```

```
x =

    8.6814
  248.8464
```

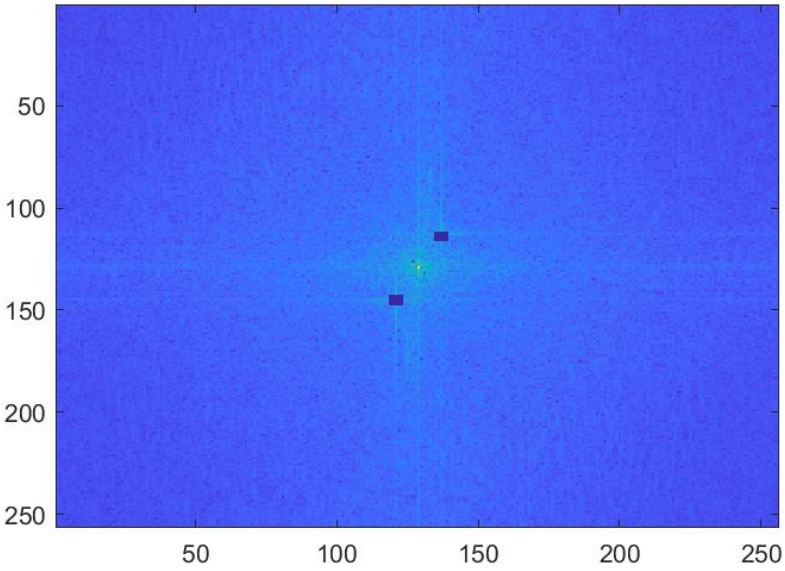
```
y =

  242.3584
   16.8173
```

Set to zero the 5x5 neighbourhood elements at locations corresponding to the peaks above:

```
%[x,y] = ginput
x1 = round(8.6814);
x2 = round(248.8464);
y1 = round(242.3584);
y2 = round(16.8173);
F(y2-2:y2+2, x2-2:x2+2) = 0;
F(y1-2:y1+2, x1-2:x1+2) = 0;
```

Recompute power spectrum and display:

```
S = real(F).^2 + imag(F).^2;
imagesc(fftshift(S.^0.1));
colormap('default');
```



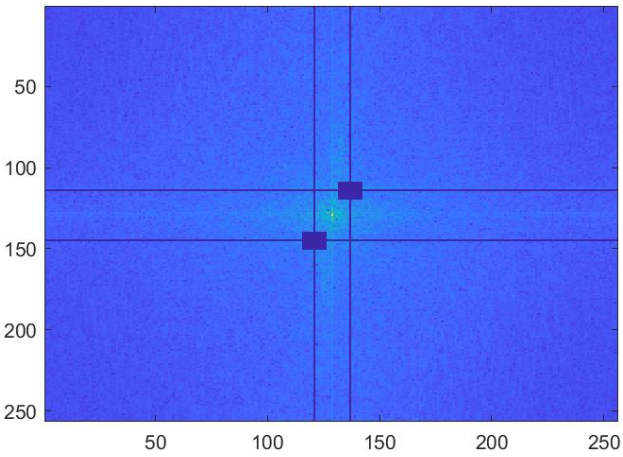
Compute inverse Fourier transform and display:

```
F_in = uint8(ifft2(F));
imshow(F_in);
```

The diagonal lines are reduced. The two white dots in the power spectrum of the original image tells us that the image contains some frequencies that are causing the interference. Hence, when we set zero the 5x5 neighbourhood elements at locations corresponding to those white dots, some interference is removed. However, the interference may be present as the area to be set zero is too small.

To improve the image, we can increase the area to be set zero to 10x10 instead of 5x5. Also, we can see from the power spectrum of the original image that the (white) lines that extend horizontally and vertically are causing the interference too, hence we remove those lines too:

```
F_in = uint8(ifft2(F));
imshow(F_in);
F_improve = fft2(pck);
F_improve(y2-5:y2+5, x2-5:x2+5) = 0;
F_improve(y1-5:y1+5, x1-5:x1+5) = 0;
F_improve(y1, :) = 0;
F_improve(:, x1) = 0;
F_improve(y2, :) = 0;
F_improve(:, x2) = 0;
S = real(F_improve).^2 + imag(F_improve).^2;
imagesc(fftshift(S.^0.1));
colormap('default');
```



Displaying the image:

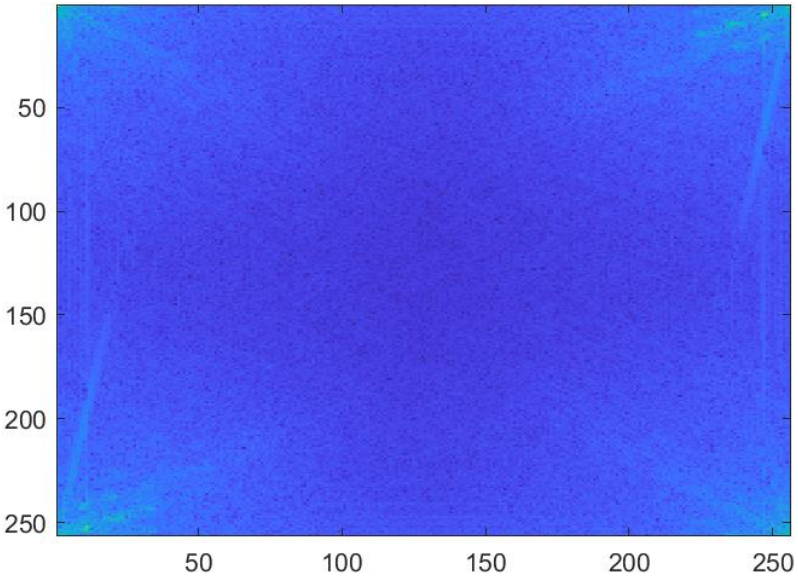
```
F_improve_in = uint8(ifft2(F_improve));
imshow(F_improve_in);
```

Download primate image:

```
primate = imread('Images/Caged Primate.jpg');
imshow(primate);
```

We get the power spectrum in the same method as previously (but without the fftshift):

```
primate = rgb2gray(primate);
F_primate = fft2(primate);
S = real(F_primate).^2 + imag(F_primate).^2;
imagesc((S.^0.1));
colormap('default');
```



Similarly, we set those white lines to zero.

We obtain the coordinates first:

```
%similarly, coordinates gotten from ginput and rounded
p1x = 11;
p1y = 252;
p2x = 22;
p2y = 248;
p3x = 248;
p3y = 4;
p4x = 238;
p4y = 11;
```

Then, similarly, we remove those lines by setting them to zero:

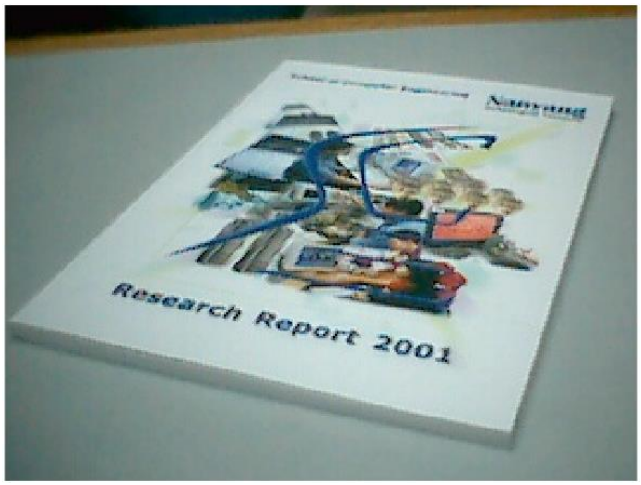
```
F_primate(p1y-3:p1y+3, p1x-3:p1x+3) = 0;
F_primate(p2y-3:p2y+3, p2x-3:p2x+3) = 0;
F_primate(p3y-3:p3y+3, p3x-3:p3x+3) = 0;
F_primate(p4y-3:p4y+3, p4x-3:p4x+3) = 0;
F_primate(p1y, :) = 0;
F_primate(:, p1x) = 0;
F_primate(p2y, :) = 0;
F_primate(:, p2x) = 0;
F_primate(p3y, :) = 0;
F_primate(:, p3x) = 0;
F_primate(p4y, :) = 0;
F_primate(:, p4x) = 0;
F_primate_in = ifft2(F_primate);
imshow(uint8(F_primate_in));
```

The fence is now less obvious.

2.6 Undoing Perspective Distortion of Planar Surface

Downloading and displaying the image:

```
book = imread('Images/Slanted View of Book.jpg');
imshow(book);
```



Obtaining the coordinates like above using *ginput* and specifying vectors to indicate four corners:

```
x =

    2.0864
  143.5028
  307.3555
  255.6841
```

```
y =

  161.2932
   28.0354
   47.0722
  215.0042
```

```
%[x,y] = ginput(4)
x = round([2.0864;143.5028;307.3555;255.6841]);
y = round([161.2932;28.0354;47.0722;215.0042]);
xd = [0;0;210;210];
yd = [297;0;0;297];
```

Set up matrices required:

```
A = [
    x(1),y(1),1,0,0,0,-xd(1)*x(1),-xd(1)*y(1);
    0,0,0,x(1),y(1),1,-yd(1)*x(1),-yd(1)*y(1);
    x(2),y(2),1,0,0,0,-xd(2)*x(2),-xd(2)*y(2);
    0,0,0,x(2),y(2),1,-yd(2)*x(2),-yd(2)*y(2);
    x(3),y(3),1,0,0,0,-xd(3)*x(3),-xd(3)*y(3);
    0,0,0,x(3),y(3),1,-yd(3)*x(3),-yd(3)*y(3);
    x(4),y(4),1,0,0,0,-xd(4)*x(4),-xd(4)*y(4);
    0,0,0,x(4),y(4),1,-yd(4)*x(4),-yd(4)*y(4);
    ];
v = [xd(1);yd(1);xd(2);yd(2);xd(3);yd(3);xd(4);yd(4)];
```

Compute matrix, convert to normal matrix form and verify correctness:

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

```
u =

    1.4536
    1.5519
 -252.7699
   -0.4293
    3.6833
  -41.3072
    0.0001
    0.0053
```

```
U =

    1.4536    1.5519 -252.7699
   -0.4293    3.6833  -41.3072
    0.0001    0.0053    1.0000
```

```
w =

         0         0  210.0000  210.0000
  297.0000    0.0000   -0.0000  297.0000
    1.0000    1.0000    1.0000    1.0000
```

The transformation gives me the 4 corners of the desired image.

Warp and display the image:

```
T = maketform('projective', U');
book_warp = imtransform(book, T, 'XData', [0 210], 'YData', [0 297]);
imshow(book_warp)
```

Yes, this is what is expected. We stretched the image to transform the original image by mapping some individual original pixels to a few pixels (which means they will have the same pixel) for the new image, hence causing the blur effect.