# Image Processing – HW4

**Problem 1**

Section a:

To answer this question, it's important to also understand what the desired outcome is. Geometric operations in general are much easier to understand and they are also much more visual compared to FFT. So, to fine tune the image it's easier to handle.

Using FFT on the other hand does still have a loss in data, but it's on the high frequencies which human eye rarely notices. It also has an advantage when specific patters are repeated in the image, which will be noticeable on the frequency domain. If we verify our sampling is above Nyquist and normalize the image by 1/sqrt(N) the resulting image should appear better compared to geometric operations.

So if you are looking for an immediate fix/change, and not planning to place the image in a gallery. The amount of tools and libraries for geometric operations offers much more options. If you need to be more precise, then FFT will offer better results.

Section b:

There are a few properties which explain why the fourier transform is unique for each image.

First and foremost, it has an inverse function. By rule, the inverse cannot have more than one distinct function. Otherwise without doing any change besides taking the inverse and going back to the original image would result in a different outcome.

Compared to the spatial domain, the frequency domain does not give us any information on the location of each pixel. But that does not mean a shifted image will have the same fourier transform, since the phase will be different. An example for this property was reviewed in slide 58 of the lecture.

**Problem 2**

Section a

Plotting the original image and fourier transform. We used the methods given in the instructions

```
image_fft = np.fft.fft2(image)
image_fft_shifted = np.fft.fftshift(image_fft)
image_fft_shifted_plot = np.log(1+np.absolute(image_fft_shifted))
```

Section b

We've implemented zero padding on the high frequencies. Resulting in a larger image but no additional data

```
image_zero_pad = np.pad(image_fft_shifted, ((image.shape[0]//2, image.shape[0]//2), (image.shape[1]//2, image.shape[1]//2)), 'constant', constant_values=0)
image_zero_pad_plot = np.log(1 + np.abs(image_zero_pad))
```

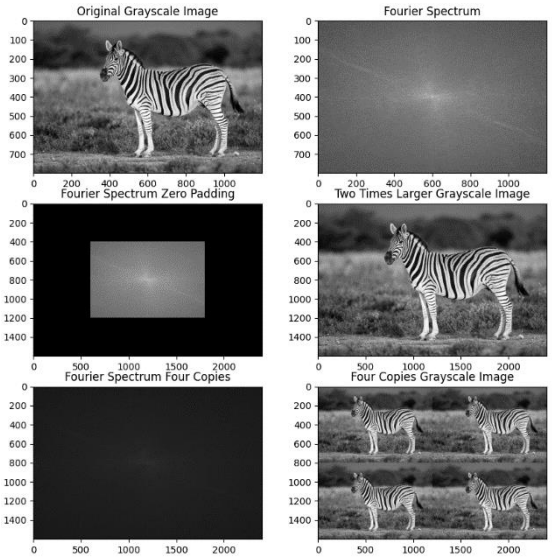
Section c

Scaling to generate four copies – we've inserted a zero value between each pixel to increase each wavelength, resulting in a four-way image. The FFT image is dark since we've added many blank values

```
four_copy_fft = np.fft.fft2(image)
four_copy_fft = np.fft.fftshift(four_copy_fft)
newH = image.shape[0]*2
newW = image.shape[1]*2
four_copy_image = np.zeros( shape: (newH, newW), dtype=complex)

for i in range(image.shape[0]):
    for j in range(image.shape[1]):
        four_copy_image[i*2, j*2] = four_copy_fft[i, j]
```

Results



Explaining the difference between the two methods of scaling

- Zero padding – a much easier implementation and does not modify any existing frequencies. It adds additional bins but they are blank, therefore the image information remains but at a larger image
- Using the scaling formula – directly applied to modify the frequency. In the example, we've scaled twice the size, meaning each u or v are now at 2u or 2v. This alters the wavelength and phase and could result in changing the image and to create aliasing (depending on the sampling rate)

The two methods we've used in section b and c are similar but with different results. Zero padding we've already discussed. Placing a zero pixel between each cell is causing a similar effect as the scaling formula, as we've shifted u and v by 2. In the scaling formula we also need to divide by the magnitude which we haven't done in section c. So the results is that we've increased the magnitude by four, and instead of increasing the image we've multiplied it. Each frequency is four times the size it was before, explaining our result.

**Problem 3**

Baby

To correct the image we first used affine transformation with warpPerspective on all three images

- Top left – scaled to 256x256
- Top right – rotated and scaled to 256x256
- Bottom right – affine transformation and scaled to 256x256

The alignment was to take the corners of each image and warp it to corners of the target image. We also make sure we use float32 as dtype in order not to lose information

```
input_pts1 = np.float32([[77, 162], [146, 115], [244, 160], [132, 244]])  # bottom right
input_pts2 = np.float32([[181, 4], [249, 71], [176, 120], [121, 50]])  # top right
input_pts3 = np.float32([[5, 19], [110, 19], [110, 130], [5, 130]])  # straight baby - left

output_pts = np.float32([[0, 0], [255, 0], [255, 255], [0, 255]])  # entire frame
```

Similar to HW3, we've used np.mean to take the mean of each pixel of the three images

```
new_image = np.median( a: [img_proj1, img_proj2, img_proj3], axis=0)
```

There were three approaches for applying a fix

- Apply the fix prior to np.mean
- Apply the fix after np.mean
- Apply both before and after

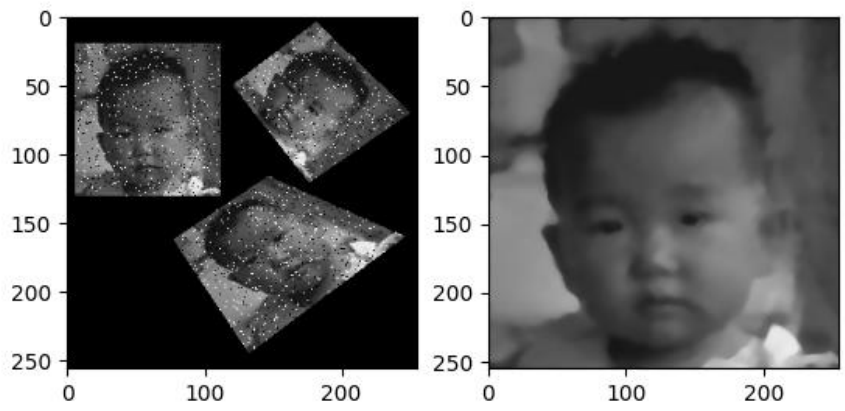
After testing all three we've settled on applying the filter before and after. This made us lose the least information. Our issue was mainly in the left ear

```
# apply median blur before combining all images
img_proj1 = cv2.medianBlur(img_proj1,  ksize: 5)
img_proj2 = cv2.medianBlur(img_proj2,  ksize: 5)
img_proj3 = cv2.medianBlur(img_proj3,  ksize: 5)
```

```
# apply filters to correct the final image
new_image = cv2.medianBlur(new_image.astype(np.uint8),  ksize: 5)
new_image = cv2.bilateralFilter(new_image.astype(np.uint8),  d: 7,  sigmaColor: 20,  sigmaSpace: 150, borderType=cv2.BORDER_WRAP)
```

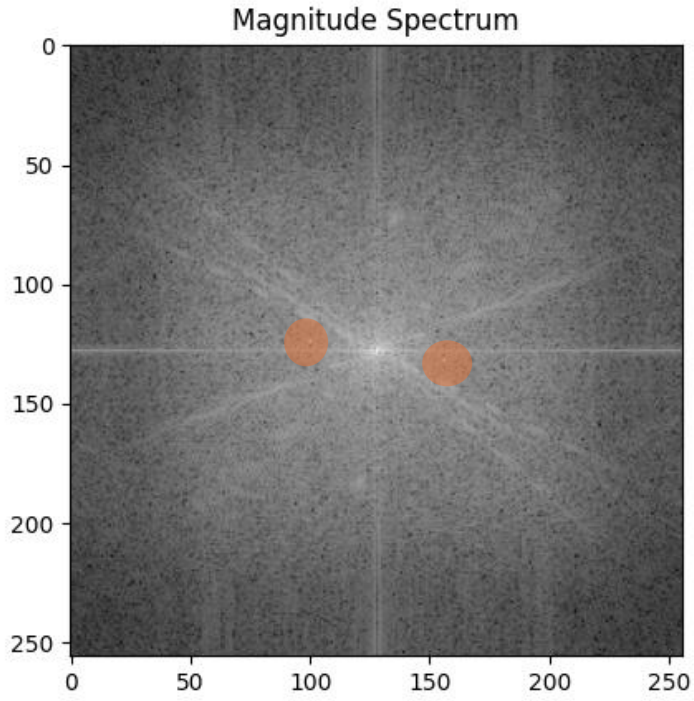
We've applied median filter before np.mean, followed by another medianBlur and bilateral filter

Results:



Windmill

The image appears to have noise at a specific frequency. After printing the FFT spectrum graph we notice two “white dots” which presents high coefficients on higher frequency parts of the spectrum.




To locate them we’ve prepared a method to test the magnitude of each point compared to the interior neighbors. As we know, the further away from the DC the lower the coefficient should be since there are less high frequency wavelengths. The threshold was adjusted until a clear image was created

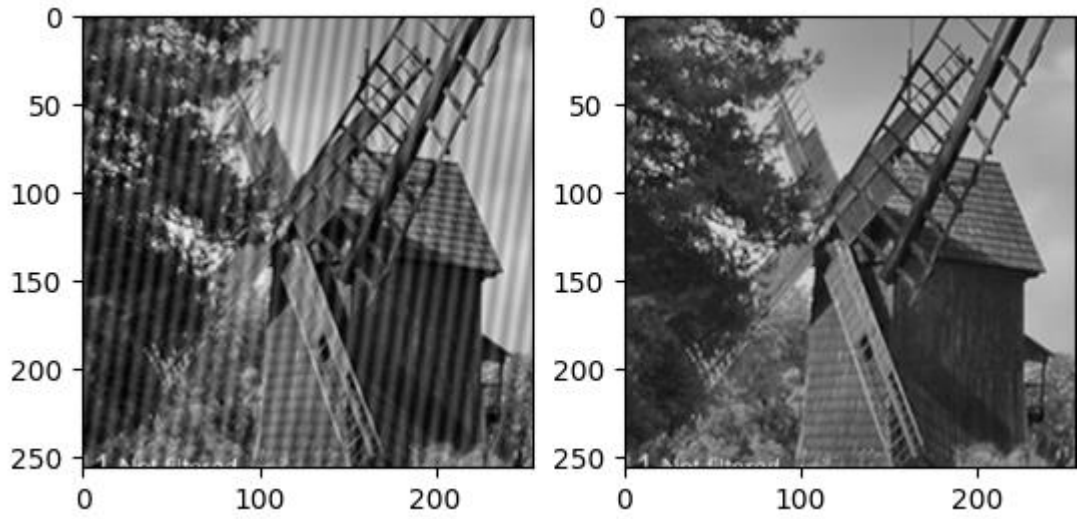
```
peaks = [] # array to hold peak values which are noise
threshold = 20 # define threshold to locate peaks
# run on half of the image and check for high magnitude indexes
for i in range(0, im.shape[0]//2):
    for j in range(0, im.shape[1]//2):
        # if the coefficient is higher than three neighbors then it may be noise, assuming that further away
        # from the DC the coefficients should reduce
        if ((magnitude[i,j]>threshold*magnitude[i+1,j+1]) & (magnitude[i,j]>threshold*magnitude[i,j+1])
                & (magnitude[i,j]>threshold*magnitude[i+1,j])):
            peaks.append([i,j])
```

After we have the peaks, we correct by taking the average from four neighbors to smooth the frequency, and since it’s a mirror image, we also correct the opposite coefficient.

```
# Calculate the average of specific high-frequency components to mitigate noise or unwanted details
avg = (image_fourier[i-1, j] + image_fourier[i, j-1] + image_fourier[i+1, j] + image_fourier[
    i, j+1]) / 4

# Replace specific high-frequency components with their averages to smooth out these frequencies
image_fourier[i, j] = avg
image_fourier[inverse_i, inverse_j] = avg
```

Results:

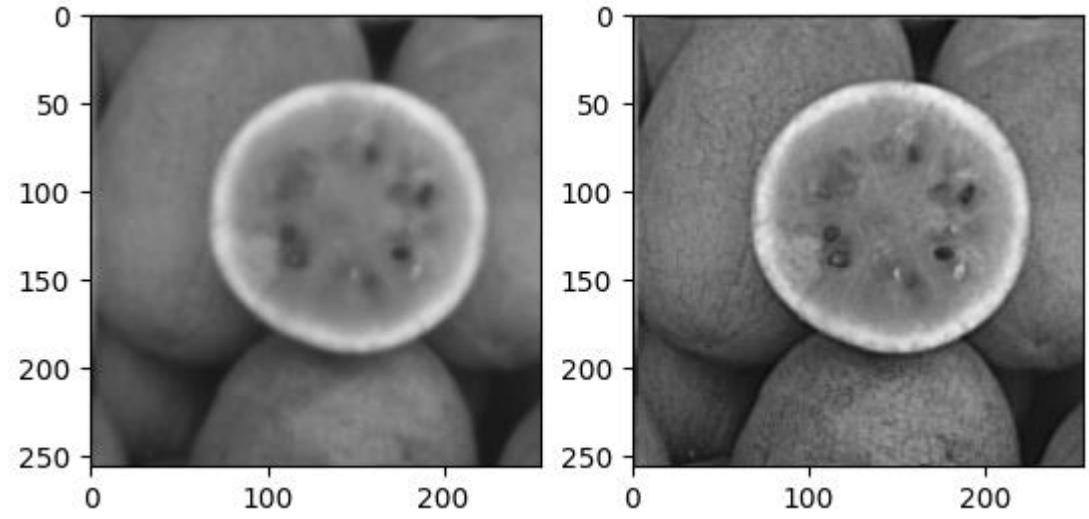



Watermelon

For the watermelon we've used a sharpening kernel

| | | |
|---|---|---|
| -1 | -1 | -1 |
| -1 | -9 | -1 |
| -1 | -1 | -1 |

Results:




Umbrella

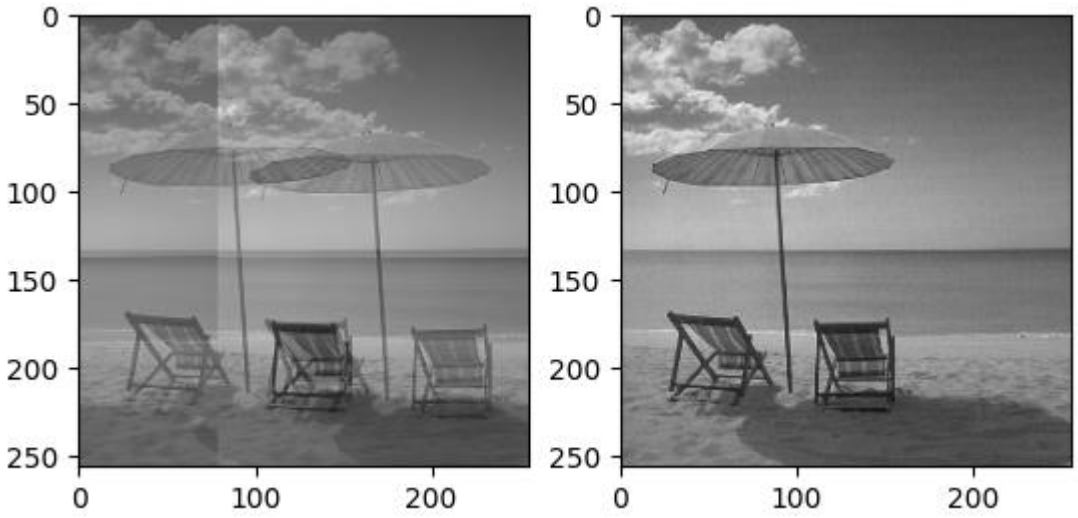
The umbrella is a blend of an image with a shifted version of it. A similar scenario was presented in class, but we haven't been able to receive any results using the equation of the average between the images. Mainly since once both images are blended the frequencies align and as is the case with pixels, once it's changed it's hard to know what the original value was. So, we took a different approach and since we know the shift is 79 pixels to the right and 4 pixels down, we've first made a delta kernel to account for half the value of each pixel in the spatial domain

```
kernel = np.zeros(im.shape)
kernel[0, 0] = 0.5
kernel[y_shift, x_shift] = 0.5
```

In Hagit's lecture on frequency processing we took the inverse filtering approach

```
f_cleaned_image = (np.conj(delta_fourier) /
                   (np.conj(delta_fourier) * delta_fourier + regularization_param)) * image_fourier
```

Results:




Flag

For the flag we first took out the stars section which didn't need cleaning. The pixel values were determined with Paint

```
roi = im[0:89, 0:141]
```

The idea is to go over the image horizontally and take the median value. This will smooth the writing and will not be affected by the vertical change of colors

First we've defined a kernel, with trail and error, 15 was the best value to clean the image and not loss too much information

```
kernel_width = 15
```

We then needed a new image to save the results + since we're using a filter and would like to remain in the same image shape, padding is needed

```
fixed_im = np.zeros_like(im) # new image to save the result
im_padded = np.pad(im,  pad_width: ((0,0), (pad_width, pad_width)),  mode: 'reflect')
```

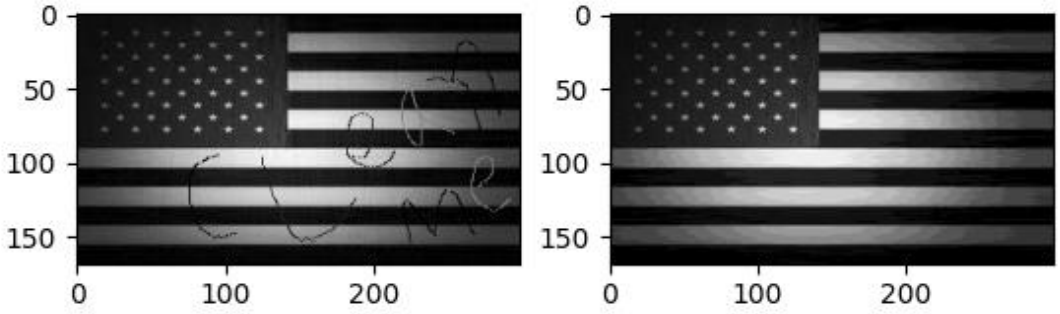
Since there is no python function to take a median filter, not a convolution, we've iterated on the image manually

```
for i in range(im.shape[0]):
    for j in range(pad_width, im.shape[1]+pad_width):
        # Extract the neighborhood
        window = im_padded[i, (j):(j + kernel_width)]
        # Compute the median and set the pixel value
        fixed_im[i, j-pad_width] = np.median(window)
```

Once that was done, we attached the start section since they were blurred from our median filter

```
fixed_im[0:89, 0:141] = roi
```

Results:




House

The basic idea to solve this, is that we want to divide the image by 10 to get the average. It is hard to find a mask that would take that average, therefore we move to the frequency domain

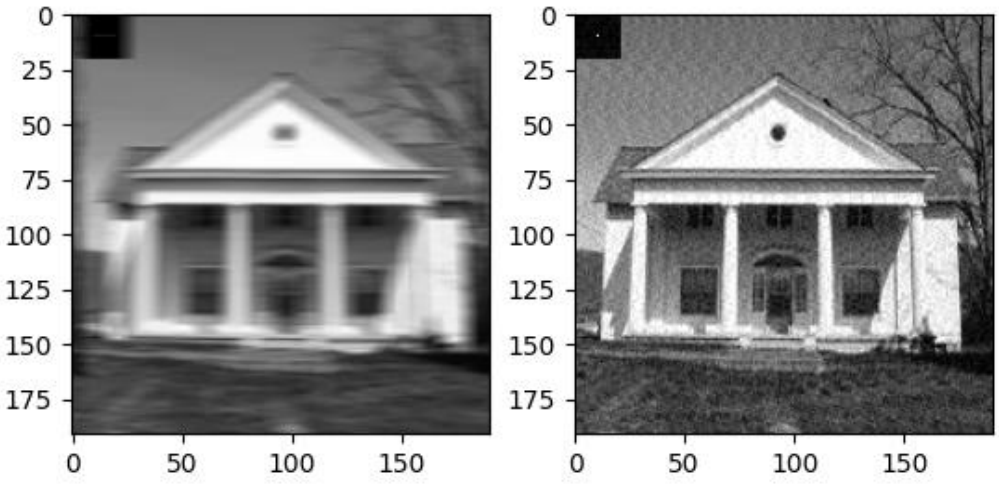
We've make an FFT of a mask that would reduce the frequencies by a factor of 10

```
mask_fft = np.fft.fft2(np.ones((1, 10)) * 0.10, im.shape)
```

Since we have 10 overlapping images, we think that reducing them to the same amplitude and phase that a single image would generate

```
fixed_im = image_fft / mask_fft
```

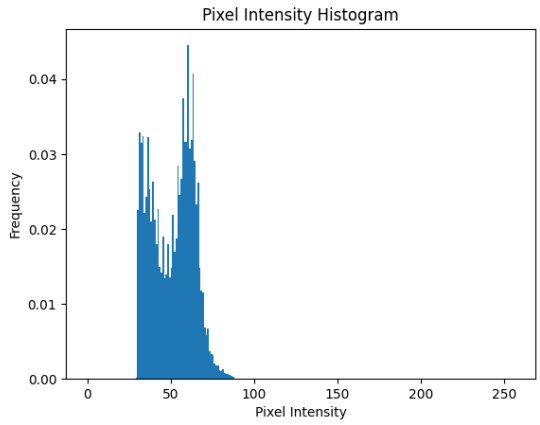
Results:



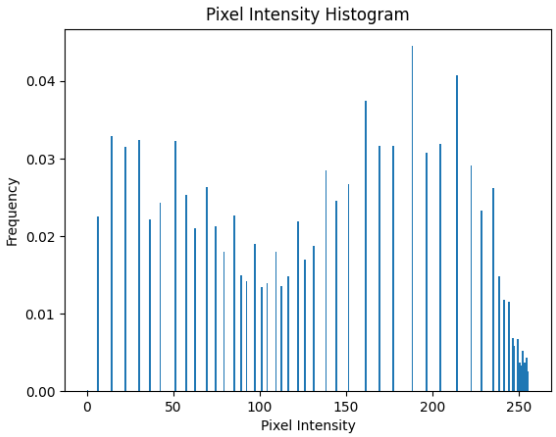
Bears

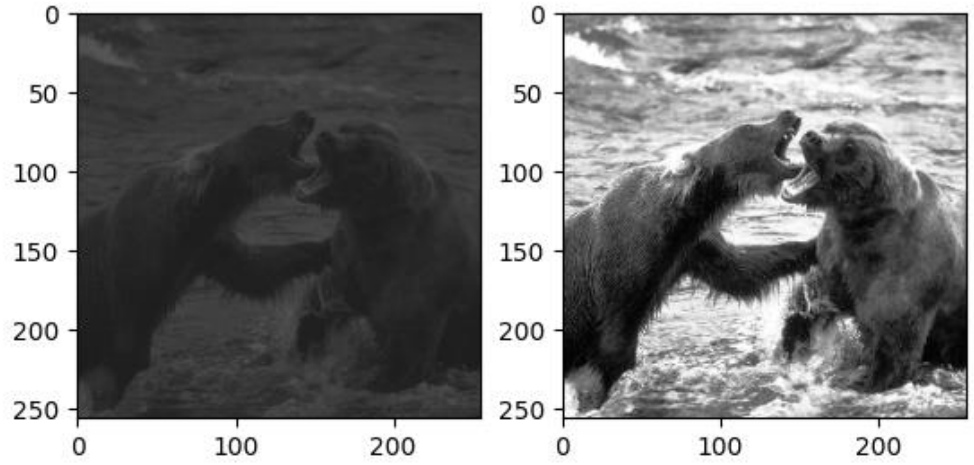
We've printed the histogram of the image, and the gray level is low and very dense, so the direction is to make more contrast. The first try was with cv2.histEqual which yielded the result below. It looks ok besides the very white areas.

Original




After histogram equalization

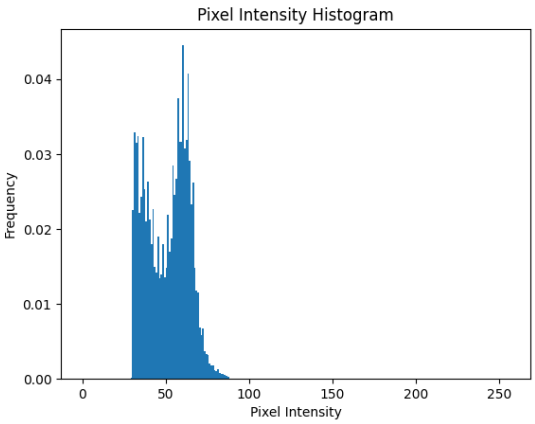



So we took the previous approach from HW2 and did an adaptive histogram equalization

```
def adaptEqual(image):
    clahe = cv2.createCLAHE(clipLimit=3.5, tileGridSize=(8, 8))
    equalized = clahe.apply(image)
    return equalized
```

Original




After adaptive equalization

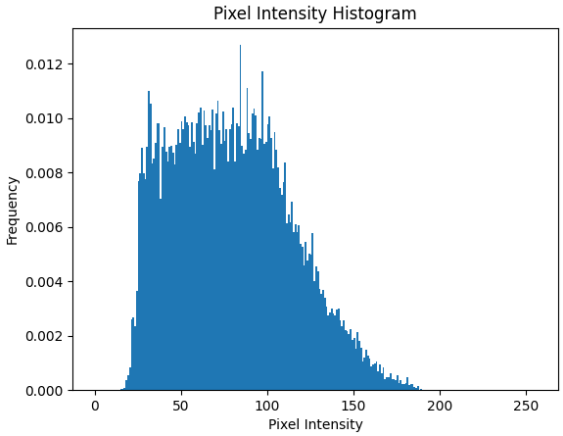
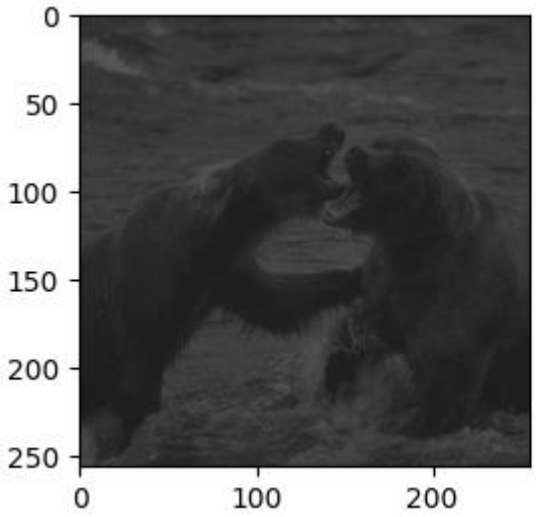
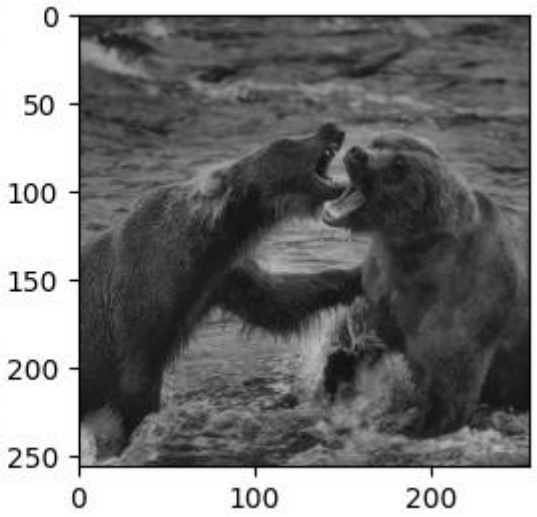



Image still not clear enough

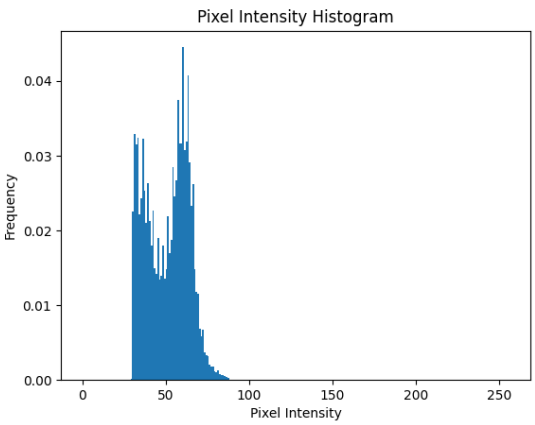


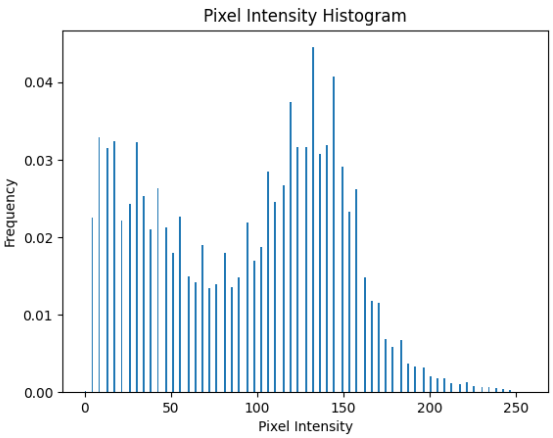

So lastly we've played with the contrast to receive this following

```
fixed_im = cv2.convertScaleAbs(im, alpha=4.25, beta=-123.25)
```

Original




After contrast adjustment



Results: