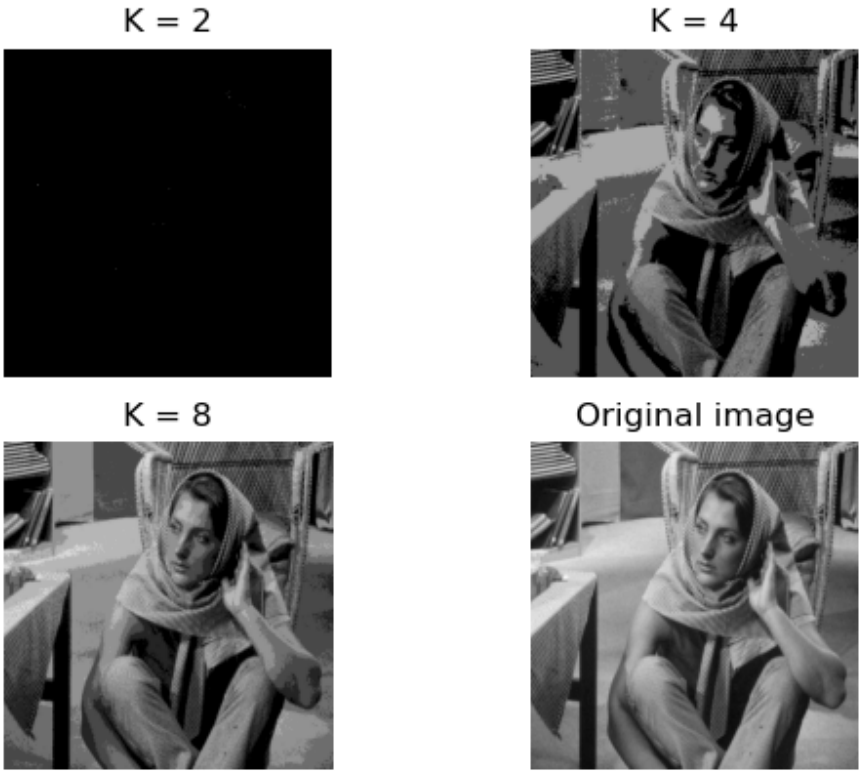
Assignment 1

Saurabh Burewar (B18CSE050)

Q1: Quantization

1. Quantize image with intensity values

   This is done by taking an evenly k spaced array from 0 to max intensity and digitizing it using numpy. Then, use a simple vectorized function on that image array to get a resulting image which has the required bits of intensity values.
   Result:

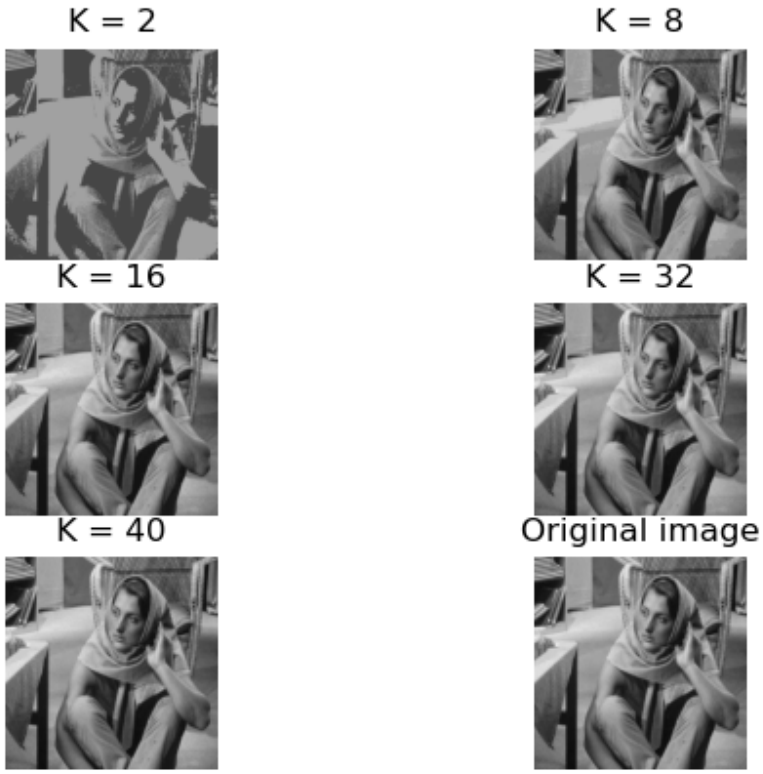


   Observations: This gives us images of different contrast as we change k. Since k = 2 means that there are 2 bits of intensity values, it gives a black image. Jumping it to 4, there is a slight variety in the intensities present in the image but still far from the original image.

For 8, the image looks very similar but since there are less intensities available, the shadows and gradients in the image are not smooth.

2. Color quantization using K-means

For quantizing into colors, we are using K-means. For that we reshape the image into the original dimension numpy array and take a sample of it. Then, use this sample to train our K-means algorithm for k clusters. Then, we predict the labels on the actual image and plot the clusters given by K-means.
Result:



Observations: For K=2, it is clearly visible as there are only 2 colors in the image due to which it loses details as well. For K=8, there is a lot less difference with the original but the gradients are not smooth but sharp. For K > 16, the image is very similar to the original since the original image itself doesn't have many colors. Therefore, when compressing, K=16 is enough to make the image as close to the original as possible without losing many details.

Q2: Interpolation

1. Write kernel functions

   The kernel function for both interpolations is written as functions where the inputs are the pixels and we output the pixel in the middle. For linear interpolation we use the formula to create our kernel function.

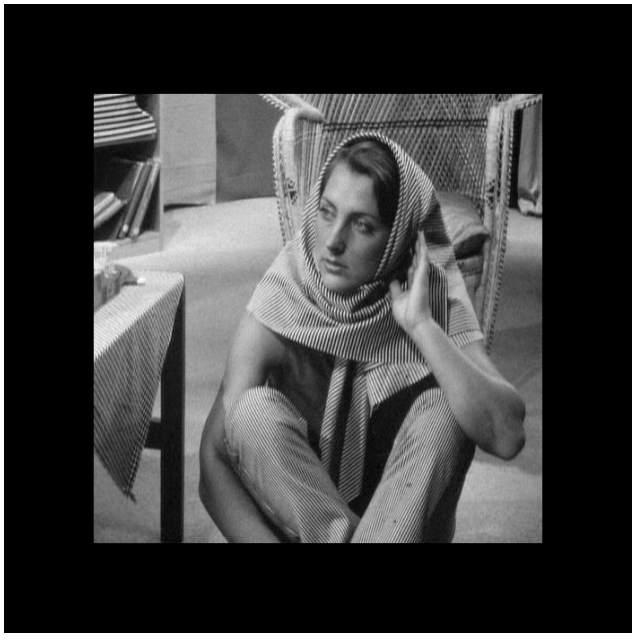
   $$f(n_{mid}) = (1 - a) * f(n_1) + a * f(n_2)$$

   For bicubic interpolation kernel, we write it according to the formula which is as follows.

   $$u(s) = \begin{cases} (a+2)|s|^3 - (a+3)|s|^2 + 1 & (0 < |s| \leq 1) \\ -a|s|^3 - 5a|s|^2 + 8a|s| - 4a & (1 < |s| \leq 2) \\ 0 & (2 < |x|) \end{cases}$$

2. Add padding

   To add padding to the image, we make a new zero array of the dimension required for a padded image. So, for a 100 pixel padding on 500 pixel image, we create an array of 700 pixels. We add our image starting from the position [100, 100] to the position [600, 600]. This leaves a padding of 100 pixels in all directions in the new image.
   Result:

3. Perform interpolation

   For bicubic interpolation we take the new pixel and use the values of 16 pixels around it as

$$dst(x,y) = \begin{pmatrix} u(x_1) & u(x_2) & u(x_3) & u(x_4) \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} & f_{14} \\ f_{21} & f_{22} & f_{23} & f_{24} \\ f_{31} & f_{32} & f_{33} & f_{34} \\ f_{41} & f_{42} & f_{43} & f_{44} \end{pmatrix} \begin{pmatrix} u(y_1) \\ u(y_2) \\ u(y_3) \\ u(y_4) \end{pmatrix}$$

   Here, $u$is the bicubic interpolation kernel function.

Q3: Affine Transformation

1. Translation by 2 pixels in any direction

   We use a transformation matrix which is $\left[\left[1\ 0\ k_x\right], \left[0\ 1\ k_y\right]\right]$. Make a zero array and loop through it to fill out all pixels. We just do a dot product of transformation matrix and original image to fill out. Then, write the translated image.

2. Scale by a factor of 2 in the x direction

   We use a transformation matrix which is $\left[\left[\ k_x\ 0\right], [0\ 1]\right]$. We do the same as in translation for the rest of the process.

3. Rotate by 30 degrees anti-clockwise

   We use a transformation matrix which is $[[\ cos(30)\ sin(30)], [-\ sin(30)\ cos(30)]]$. We take the center since we are rotating the image about the center and fill out the pixels. The process is similar to before.

4. Above three transformations together

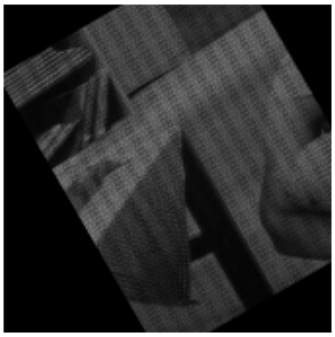
   Since the order was not mentioned, I just did it in the order in which the above three are mentioned.
   Result:

Translation by 2 pixels

Scaling by factor of 2

Rotation by 30 degrees

Combination of the all three



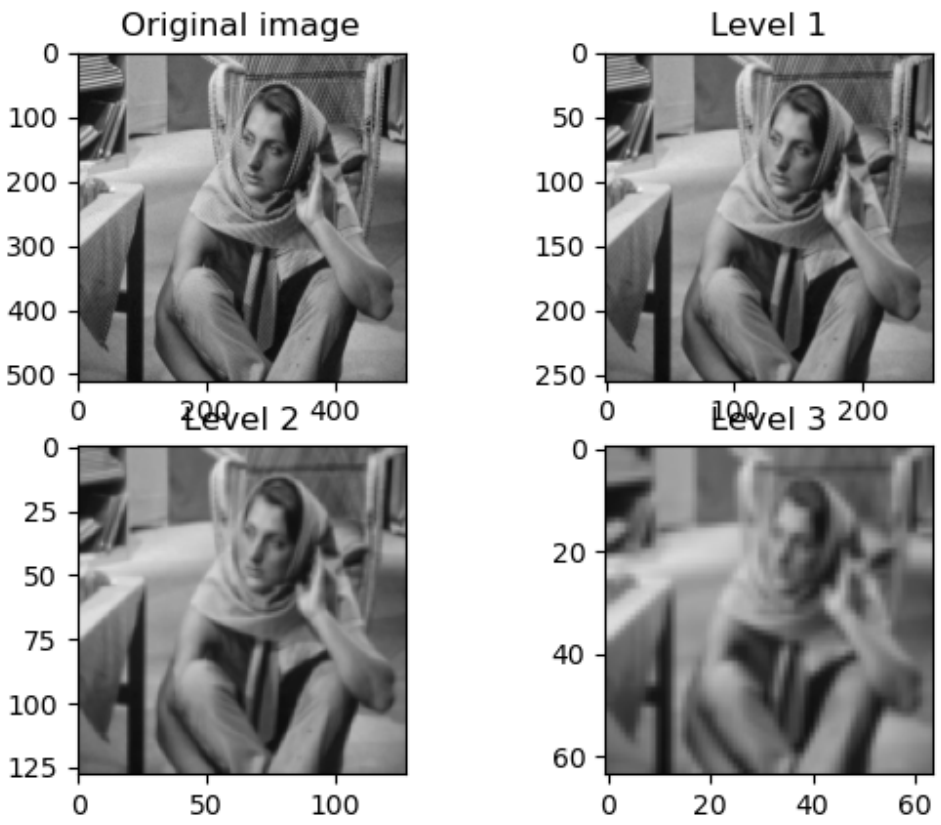
Observations: The translation, since it is 2 pixels, is not clearly visible (there is a small black slit on the left side, if we look closely). The scaling is done in the x direction which basically stretches the image towards the x direction. The rotation is about the center which is clear. The combination is translation followed by scaling followed by rotation. If we change the order, the image will look a lot different. For example, if we do rotation first, then scaling will cause scaling of the rotated image, so the black back on the left won't be there.

Q4: Gaussian and Laplacian pyramid

1. Gaussian Pyramid

   We use OpenCV to read the image and go through the gaussian pyramid. So, we go through 3 levels of the pyramid and store it all in a list which is plotted below. If we use level 3 as a blurred image it is very low resolution. Therefore we go up a level to increase resolution but the details stay the same, so it gives a smooth blur.

Result:



Gaussian Pyramid

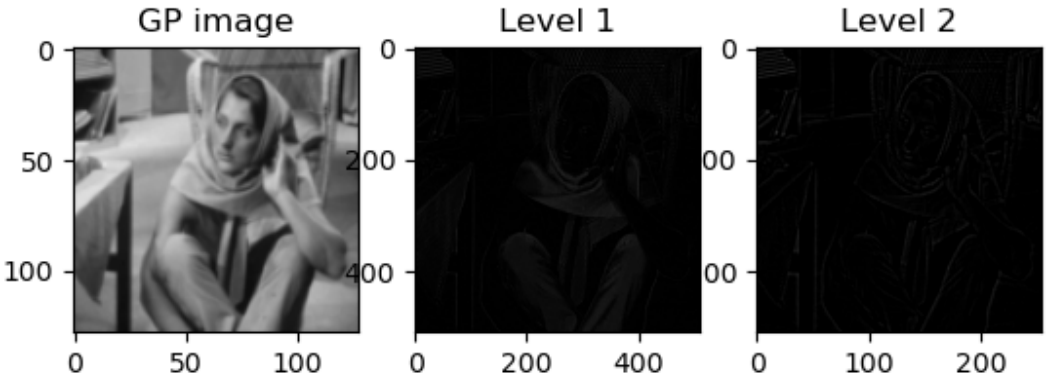
Blurred image

Observations: The image we get is blurred and smooth since we increased resolution by going one level up and since the information in the image is still the same as the lower level, we get a smooth blurred image.

2. Laplacian Pyramid

We get the laplacian pyramid from the gaussian pyramid. We take the less information image from the gaussian pyramid and make an extended gaussian by looping through the gaussian pyramid and subtracting the original from the extended. This basically subtracts the less detailed image from the more detailed one, leaving out the edges. This gives us the laplacian.

Result:



Laplacian Pyramid

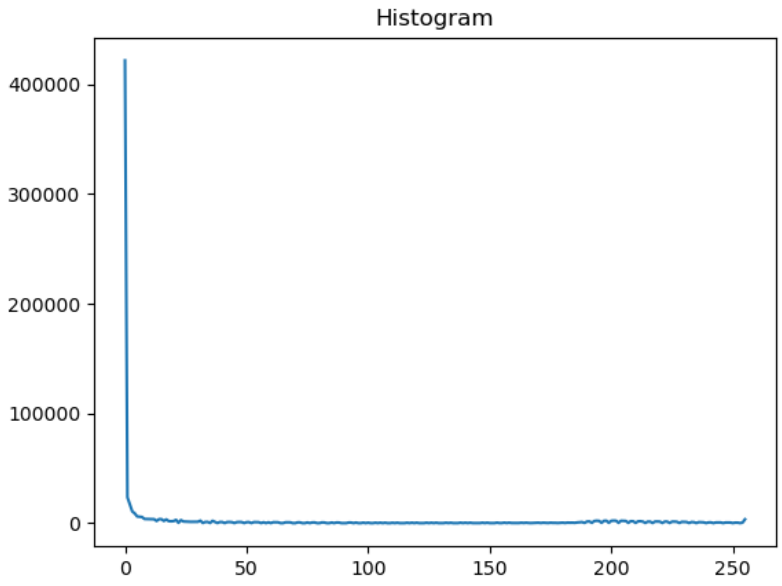
Laplacian image

Observations: We can see how taking the laplacian gives us the edges since we are subtracting blurred images from detailed images.

Q5: Histogram

1. Plotting histogram

   A histogram is a plot of the number of pixels of each intensity. So, we loop through all the pixels in the image and count the number of pixels with every intensity value. We maintain a dictionary of counts for this and then plot this dictionary to get the histogram.
   Result:



   Observations: We have 0 to 255 intensity values on the x-axis and the number of pixels of each intensity on the y-axis. We see that there are a lot of pixels with low intensity (even 0). This is because the image is dark and has a lot of black area which is basically 0 intensity.

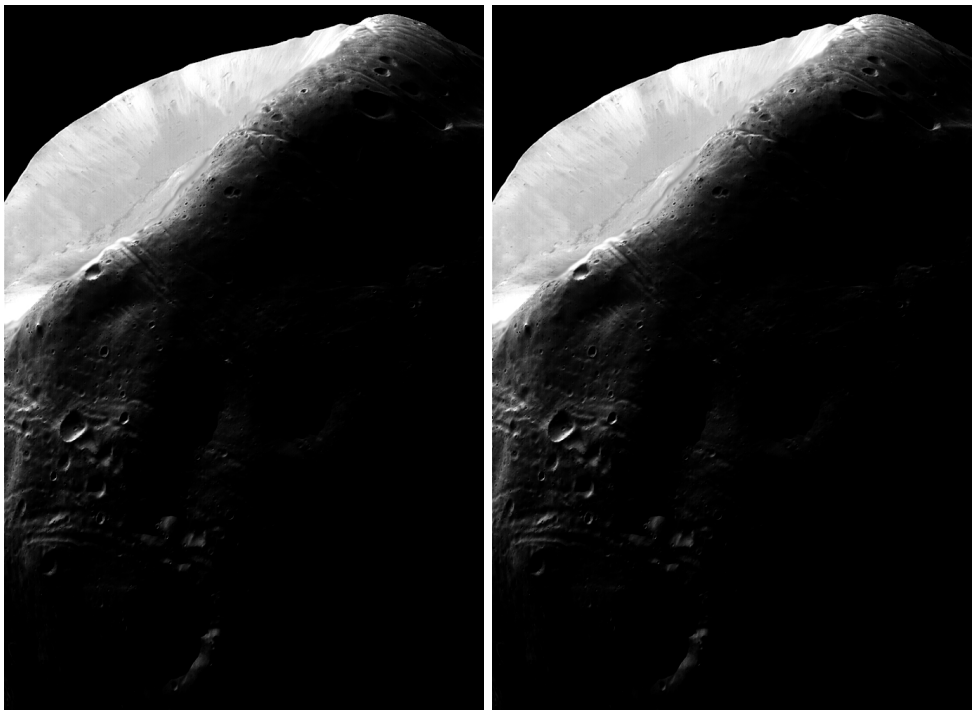
2. Contrast stretching

   We take the min and max intensity values in the image and get a constant value from the below formula -

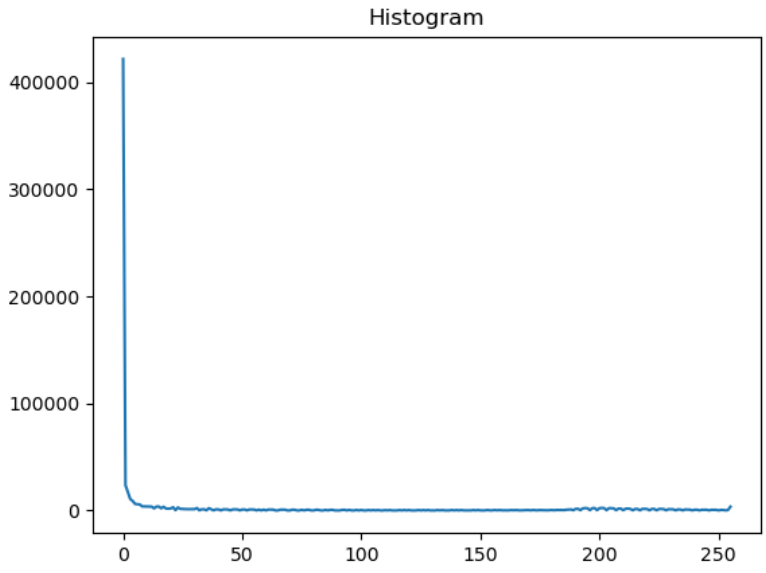
$$C = (255 - 0) / (max\_intensity - min\_intensity)$$

   Now, we multiply the image with this constant to get our stretched image.

Result:



Original *Stretched*



Histogram of the stretched image

Observations: In this case, the max intensity is 255 and min is 0 which causes the constant to be 1. This makes the stretched image the same as the original. The histogram of this stretched image is, therefore, the same as our original image.

Q6: Watermarking

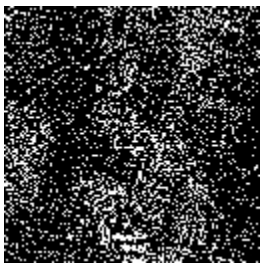
1. Watermarking

   We first resize the images and convert them to an image array. We use the DWT function by pywavelets to perform DWT. We do it up to level 4 and then perform a DCT on the first output coefficient of DWT. This is done using Scipy functions. We loop through the first coefficient's values to get the output. Now, we embed the watermark. We take the output of our DCT and embed our watermark image in a flattened state. Then, we perform inverse DCT in a similar fashion as DCT and put this watermarked output back in the first coefficient in the DWT output array. Finally, we reconstruct the image using the pywavelets inverse DWT function and clip the image to make sure there are no values outside the 0 to 255 range. This gives us the watermarked image. This whole algorithm was taken from the paper on DWT-DCT watermarking algorithm (referenced below).
   Result:

   Observations: Since this is an invisible watermark, the watermark is not visible but can still be extracted since we know how it was embedded.

2. Recovering watermark

   To get the watermark, we do the DWT using the pywavelets. Then, we perform a DCT similar to how we did before. Then we go through the output of the DCT, similar to the embedding process and slice the image to get a watermark. Then, we return our watermark.
   Result:



   Observations: In this case, the watermark did not turn out clear and some experiments with the level of DWT and resolutions are needed to get the best result.

References

- https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
- https://medium.com/swlh/image-processing-with-python-digital-image-sampling-and-quantization-4d2c514e0f00
- https://ieeexplore.ieee.org/document/1163711
- https://docs.opencv.org/3.4/d4/d1f/tutorial_pyramids.html
- https://github.com/diptamath/DWT-DCT-Digital-Image-Watermarking
- Al-Haj, Ali. (2007). Combined DWT-DCT digital image watermarking. Journal of Computer Science. 3(9). 10.3844/jcssp.2007.740.746.

Code

The code for this report is available in the zip folder named “B18CSE050assignment1”.