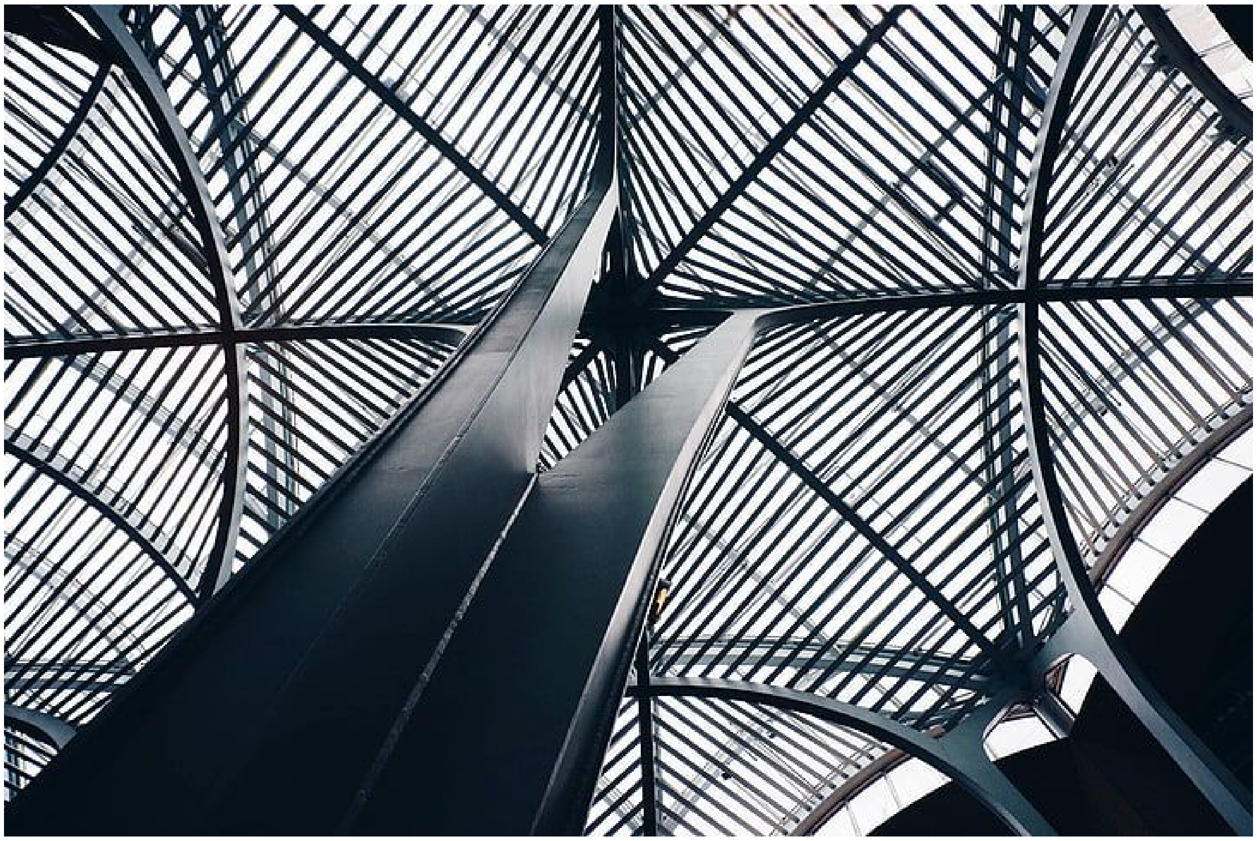


Digital Image Processing & Analysis - Assignment 1 Report

07.02.2020

Hansin Ahuja
2018csb1094

Overview

We were given 8 basic image processing techniques that we needed to implement from scratch. These 8 being:

1) Image resizing using nearest neighbour interpolation
2) Image resizing using bilinear interpolation
3) Image rotate
4) Bit plane slicing
5) Inverse affine transform using tie points
6) Histogram equalization
7) Histogram matching
8) Adaptive histogram equalization

Goals

1) To be able to generate images that are either aesthetically pleasing, or convenient/easy for the computer to process.
2) To be able to replicate the inbuilt Matlab functions for the aforementioned 8 functions.
3) To visualise the results produced by my code, along with the the results produced by Matlab's internal functions, and compare the two according to various metrics like root mean square, average grey level, etc.

Notes and Assumptions

1) It has been assumed that the input will be a grayscale image.
2) All errors are calculated on the 0-255 scale rather than the 0-1 scale.
3) All functions return the RMSE between my output image and the internal functions' output image, and the dimensions of the output image (not for all functions).
4) The functions can be run with the following command:

```
[err, dim] = A1_Hansin_2018CSB1094_2019_CS517(qID,
             fname_inp1, fname_inp2, fname_out, prmts,
             toshow)
```

1) Image resizing using nearest neighbour interpolation

```
err = A1_Hansin_2018CSB1094_2019_CS517(1, fname_inp, ‘’,
      fname_out, output_shape, toshow)
```

Method

For every pixel in the output image, map the pixel value to a corresponding coordinate in the input image. Find the nearest pixel in the input image to this mapped value, then assign this pixel value to our target pixel.

Observations:

1) The method taught in the classroom seems to be accurate only for upsampling, i.e. output shape >= input shape. The inbuilt function seems to apply extra operations in order to smoothen the output image in case of downsampling.
2) For upsampling, I’ve managed to replicate the inbuilt function, with an error of zero for all the testing that I did.
3) This method is inferior when compared to bilinear and bicubic interpolation, with visible jagged edges being produced in the output image.

Results:

Image A

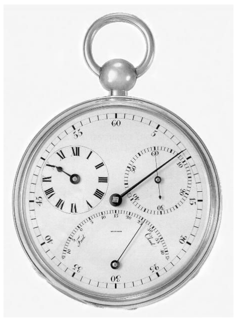
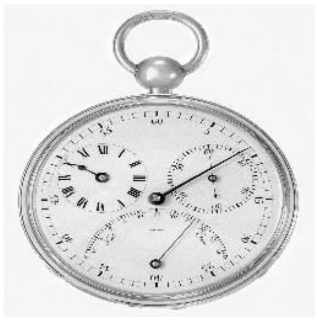
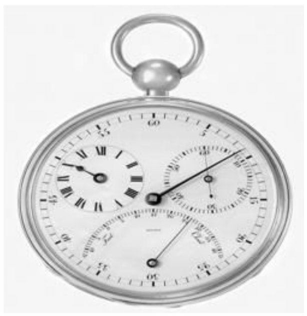
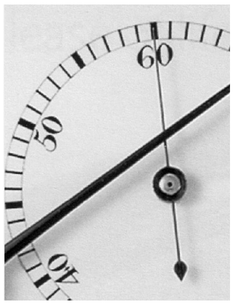
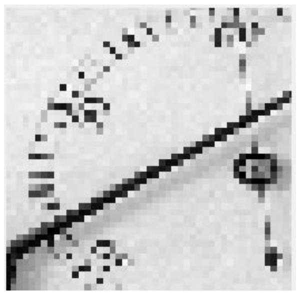
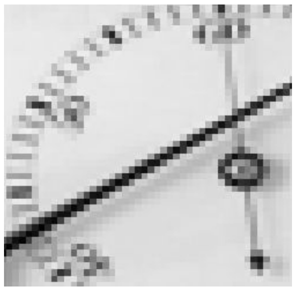
Image B

Image C

Image D

Image	A	B	C	D
Description	Input image	My output image	Internal function output	Internal function output using bicubic interpolation
Dimensions	258 x 350	512 x 512	512 x 512	512 x 512
Mean gray level	49.05	49.01	49.01	49.06
Error (RMSE)	Between B and C: 0			
	Between B and D: 4.75			
	Between C and D: 4.75			

As evident by the zero error between B and C, and equal error between B & D and C & D, we've been successful in replicating Matlab's internal function.

Here we have the original image with perfectly smooth edges, and the downsized images produced by nearest neighbour and bicubic interpolation, respectively. The second image has much worse quality and jagged edges. The third image, even though not perfect, is much better. This shows the superiority of the bicubic method as compared to the nearest neighbour method.

2) Image resizing using bilinear interpolation

```
err = A1_Hansin_2018CSB1094_2019_CS517(2, fname_inp, ‘’,
      fname_out, output_shape, toshow)
```

Method

1) For every pixel in the output image, map the pixel value to a corresponding coordinate in the input image.
2) Find the four nearest neighbours of this mapping, and apply bilinear interpolation to these 4 pixel values based on distance.
3) Assign this interpolated value to the target pixel.
4) In case of only 2 neighbours, apply linear interpolation and in case of only 1 neighbour, we apply nearest neighbour interpolation.

Observations:

1) The method taught in the classroom seems to be accurate only for upsampling, i.e. output shape >= input shape. The inbuilt function seems to apply extra operations in order to smoothen the output image in case of downsampling.
2) For upsampling, there are two cases: the input image is in uint8 format and when it is in double format. For the second case, we match the inbuilt function perfectly with 0 error. However, in the first case, Matlab’s inbuilt function rounds off some of the intermediate values, leading to some error. The mismatching pixels differed by at most 1, so we can conclude that the implementation methods aren’t too different.
3) This method is superior to nearest neighbour interpolation, but inferior to bicubic interpolation. Bilinear interpolation improves upon some of the jagged edges, but not as good as bicubic interpolation.

Results:

Image A

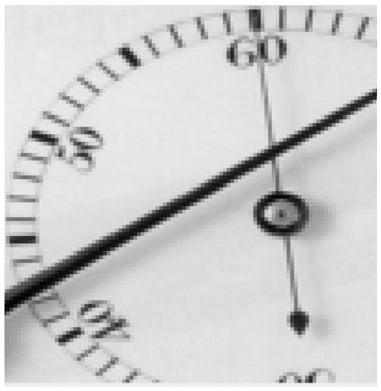
Image B

Image C

Image D

Image	A	B	C	D
Description	Input image	My output image	Internal function output	Internal function output using bicubic interpolation
Dimensions	256 x 256	512 x 512	512 x 512	512 x 512
Mean gray level	77.26	77.26	77.26	77.26
Error (RMSE)	Between B and C: 0			
	Between B and D: 1.26			
	Between C and D: 1.26			

As evident by the zero error between B and C, and equal error between B & D and C & D, we've been successful in replicating Matlab's internal function.



This is the same image shown in the previous section, but resized using bilinear interpolation. We can see that it improves the jagged edges shown by the nearest neighbours method, but not as good as bicubic interpolation.

3) Image rotate

```
err = A1_Hansin_2018CSB1094_2019_CS517(3, fname_inp, ‘’,
      fname_out, angle, toshow)
```

Method:

1) Calculate the size of the output image using simple geometry.
2) For every pixel in the output image, map the pixel value to a corresponding point in the input image using the following inverse affine transform:

$$T = \begin{bmatrix} cosx & sinx \\ -sinx & cosx \end{bmatrix}$$

3) This transformation gives us the spatial position of the target pixel in the input image’s frame. To get the pixel intensity, we apply bilinear interpolation using the inverse mapped point.

Observations:

1) The shape of the output image can’t be calculated using the template formula. Special cases we need to handle: theta = 90, 180, 270...
2) Mapping the top-left position of the pixel using the matrix above gives decent results, with most of the error being concentrated at the edges in the image. However, excellent results are obtained by mapping the central coordinates of the pixels instead.
3) Simply applying the transformation matrix also has one more problem: it would rotate the image about the top left pixel. To obtain the sense of rotation about the central pixel, we need to shift the origin to the central pixel before transforming, and then shift the origin back to the top-left pixel once we’re in the domain of the input image.

Results:

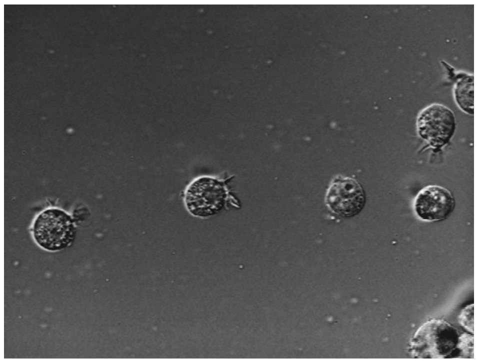


Image A

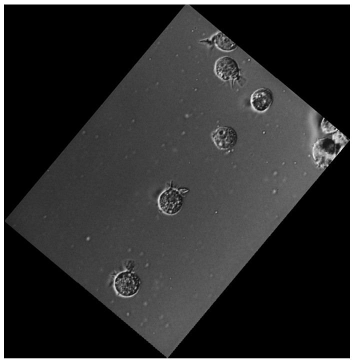


Image B

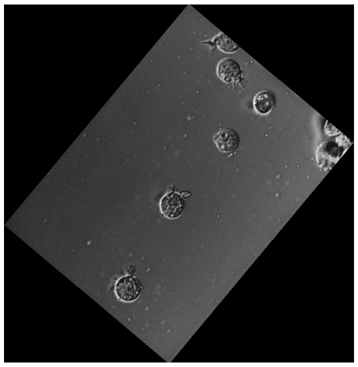
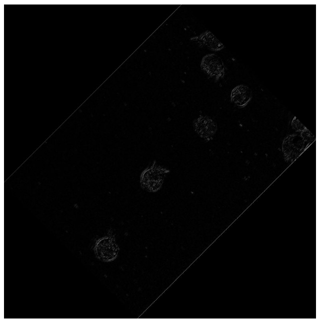


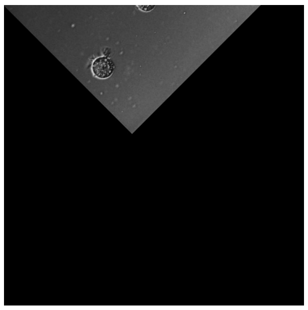
Image C

Image	A	B	C
Description	Input image	My output image	Internal function output
Dimensions	480 x 640	799 x 780 with theta = 50	799 x 780 with theta = 50
Mean gray level	76.56	37.74	37.74
Error (RMSE)	Between B and C: 0.2 (Errors as low as 0.04 were observed as well)		

Upon inspection, I found out that Matlab's internal function uses nearest neighbour interpolation while rotating an image. Perhaps we could get better results using nearest neighbour interpolation.



Normalized error image if we take top left pixel coordinate instead of central coordinate.



Result when we don't shift the origin, i.e. rotation about top-left corner.

4) Bit plane slicing

```
err = A1_Hansin_2018CSB1094_2019_CS517(4, fname_inp, ‘’,
      fname_out, reference_numbers, toshow)
```

Method:

1) Extract the switched on bits in the reference number.
2) Retrieve only the planes based on the aforementioned bits.
3) Multiply each plane by the requisite power of two and add them up.

Observations:

1) Most of the information is stored in the higher order planes.
2) Visualising the lower order bits would require multiplying the plane with a constant. Otherwise, we would get an output of extremely low gray levels.

Results:

Image A

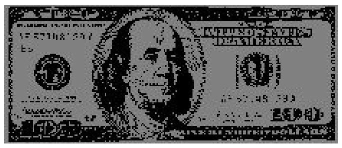


Image B

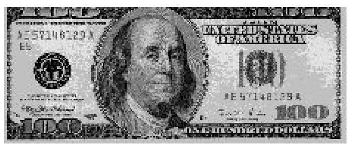


Image C

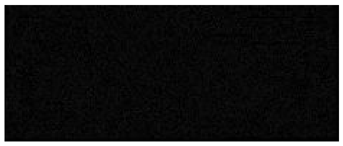


Image D

Image	A	B	C	D
Description	Input image	Plane 8	Planes 6, 7, 8	Planes 1, 2, 3, 5
Dimensions	500 x 1192	500 x 1192	500 x 1192	500 x 1192
Mean gray level	151.49	83.58	136.98	6.87
Error (RMSE) vs Input image	-	75.25	17.56	160.23

5) Inverse affine transform using tie points

```
err = A1_Hansin_2018CSB1094_2019_CS517(5, reference_image,
        input_image, fname_out, tie_points, toshow)
```

Method:

1) Use the tie points to find a bilinear transformation from the reference image to the input image, by setting up the following matrix equation:

$$\begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & x_1y_1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_1 & y_1 & x_1y_1 & 1 \\ x_2 & y_2 & x_2y_2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_2 & y_2 & x_2y_2 & 1 \\ x_3 & y_3 & x_3y_3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_3 & y_3 & x_3y_3 & 1 \\ x_4 & y_4 & x_4y_4 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_4 & y_4 & x_4y_4 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \end{bmatrix}$$

2) For every pixel in the output image, map the coordinates into the input image's domain.
3) Use bilinear interpolation to find the pixel intensity of the target pixel.

Observations:

1) The tie point selection is crucial. Inaccurate selection would lead to undesired warping of the output image.
2) The extent to which we can improve the output also depends upon the spatial resolution of the input image. We would get a better and more accurate transformation if we were able to pinpoint the tie points with better accuracy. The gray levels of a corner, for example, would get distributed to multiple pixels after warping. Identifying the exact point where the corner was mapped to would require higher spatial resolution.

Results:

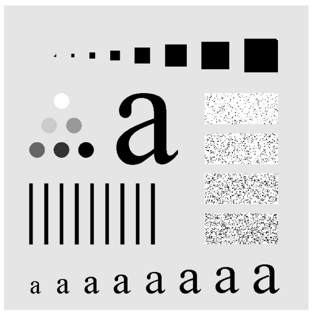
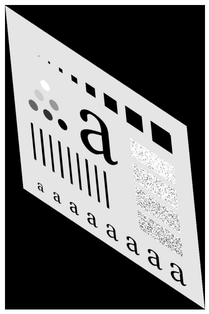


Image A


Input image

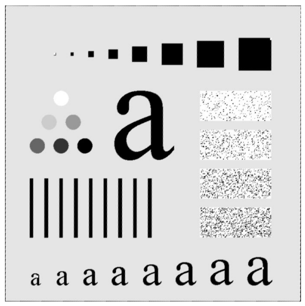


Image B

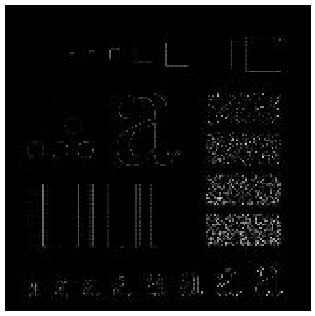
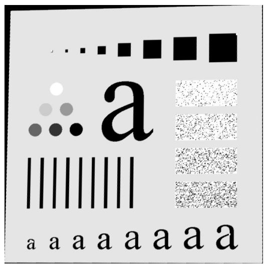
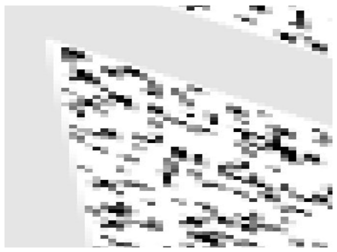

Error image

Image	A	B
Description	Reference image	Output image
Dimensions	688 x 688	688 x 688
Mean gray level	207.35	206.24
Error (RMSE) vs Input image	-	23.99

The higher error can be attributed to the nature of the original image:

1) The spatial resolution doesn't allow us to pick the most accurate tie points.
2) The image involves large sections where there is a black and a white pixel on two sides of an edge. Even the slightest offset leads to these pixels contributing to the maximum possible values to the RMSE. This is evident by the error image shown above.


Inaccurate tie point selection leads to warping of image.


Lower image resolution leads to inaccurate tie point selection.

6) Histogram equalization

```
err = A1_Hansin_2018CSB1094_2019_CS517(6, fname_inp,
      ‘’, fname_out, [], toshow)
```

Method:

1) Find the cumulative distribution function of the input image.
2) Scale the values to 0-255 and round them off.
3) Map the pixels of the input image using the function described above.

Observations:

1) Histogram equalization helps in contrast enhancement.
2) However, sometimes in images with stark intensity differences with high frequency, histogram equalization might not always work. An example is given later.

Results:

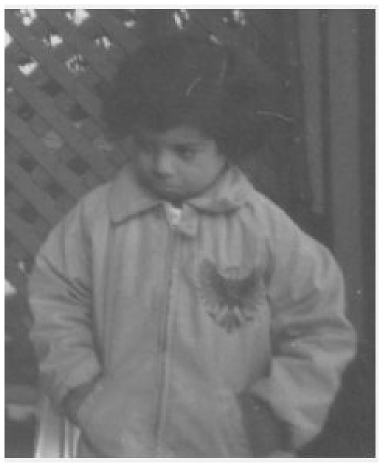


Image A

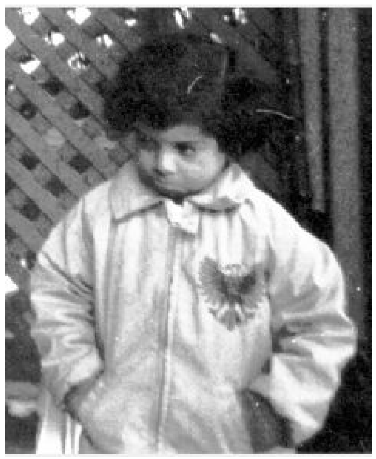


Image B

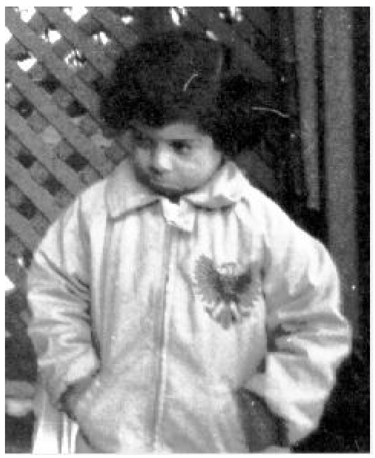
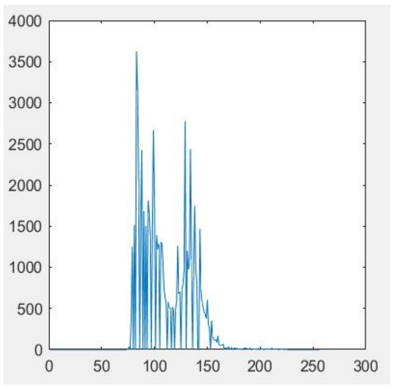
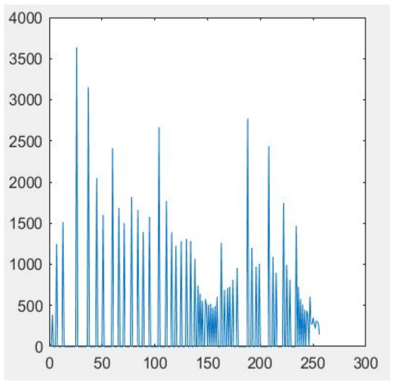


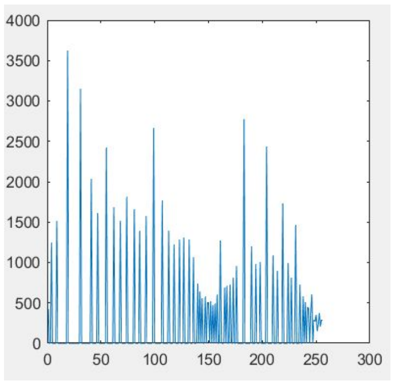
Image C



Histogram for image A

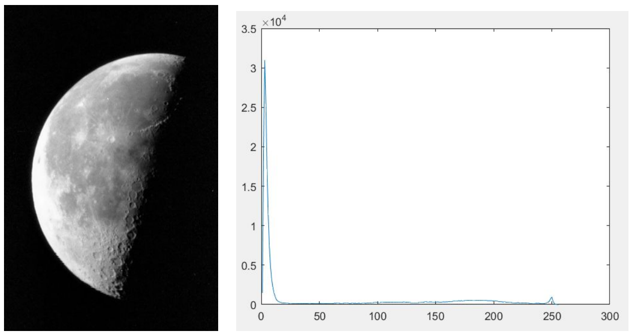


Histogram for image B

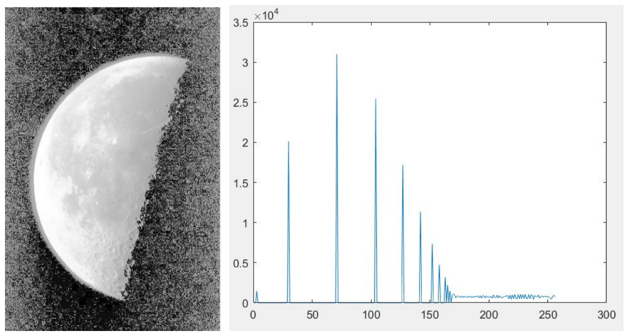


Histogram for image C

Image	A	B	C
Description	Reference image	My output image	System output image
Dimensions	291 x 240	291 x 240	291 x 240
Mean gray level	110.30	130.41	127.41
Error (RMSE)	Between B and C: 3.53		



Input image and its histogram



Output image and its histogram

As shown above, histogram equalization doesn't always give aesthetically pleasing results. The above image is very lopsided to the left in its histogram. If we apply histogram equalization for contrast enhancement, it would produce undesirable artifacts as shown on the right. Histogram matching might be a better fit for this kind of an image.

7) Histogram matching

```
err = A1_Hansin_2018CSB1094_2019_CS517(7, fname_inp1,
      fname_inp2, fname_out, [], toshow)
```

Method:

1) Apply histogram equalization to both the input images.

2) Find an inverse function which maps each pixel value from the input image straight to the output image.

3) Given that this is the discrete image, an inverse might not always exist. In that case, we choose the smallest pixel value in case of a many-to-one mapping.

Observations:

1) Given that we are working in the discrete domain, we have to round off quite a few values in intermediate steps. This leads to us not being able to obtain a perfect inverse, and we are forced to resolve conflicts arbitrarily.

2) We could perhaps try avoiding rounding off in intermediate steps in order to get better results. The inbuilt function seems to be doing this as well.

Results:

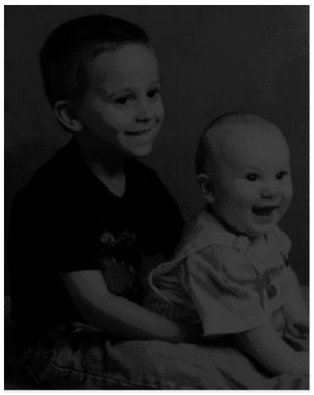


Image A

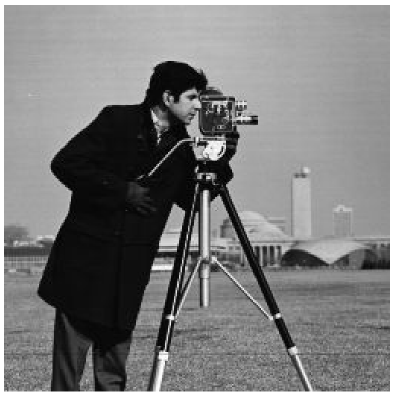


Image B

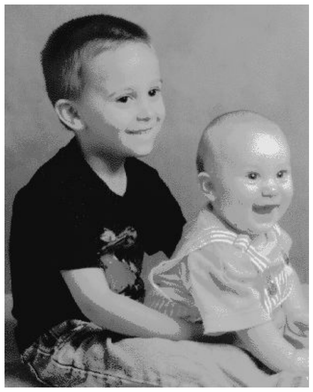
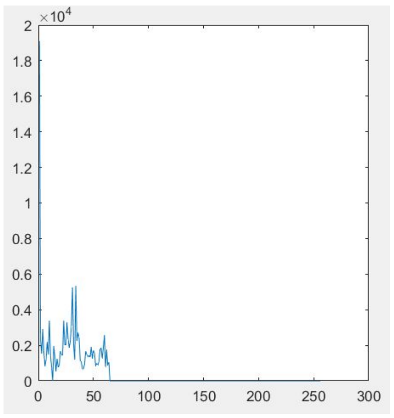
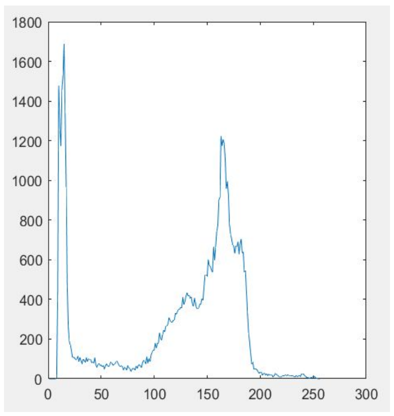


Image C


Histogram for image A


Histogram for image B

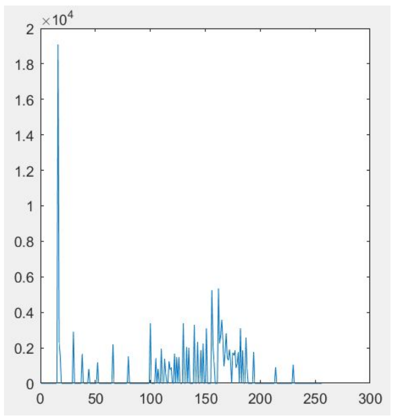

Histogram for image C

Image	A	B	C
Description	Input image 1	Input image 2	Output image
Dimensions	400 x 318	256 x 256	400 x 318
Mean gray level	26.13	118.72	120.33
Error (RMSE)	Between histograms of B and C: 0.01		

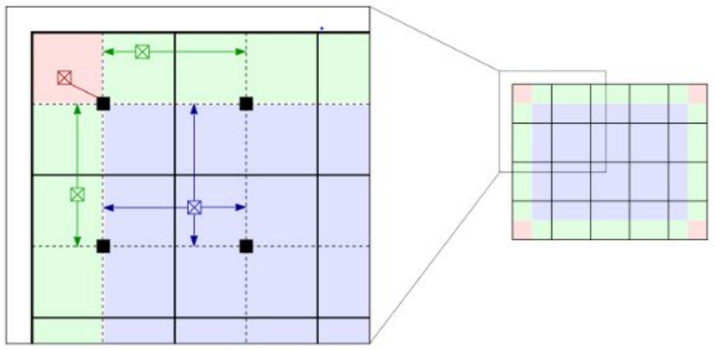
Note: The error was calculated after normalizing the two histograms using their respective images. This is only logical. Take the example of a 1x1 black image and a 256x256 black image. Even though the histograms of these two images match perfectly, we would have an error of close to 6.5e4.

8) Adaptive histogram equalization

```
err = A1_Hansin_2018CSB1094_2019_CS517(8, fname_inp,
      ‘’, fname_out, [], toshow)
```

Method:

1) This is similar to histogram equalization, except that we transform each pixel using a function derived from its neighbourhood.
2) We divide the image up into grids of size nxn (n = 8 in my implementation), and get a transformation function from each of these grids.
3) For each pixel in the output image, we get the central coordinates of the four nearest grids, and use bilinear interpolation on these four transformation values to get the pixel intensity of the target pixel.
4) For pixels with only 2 nearest grids, we apply linear interpolation and for pixels with only 1 nearest grid, we apply nearest neighbour interpolation, as shown below:



Observations:

1) This method is suitable for enhancing the local contrast and enhancing the definitions of edges in each region of an image.
2) The usage of bilinear interpolation makes sure that the delineations of the tiles are smooth, and hence avoiding a “blocky” look for the image.

Results:

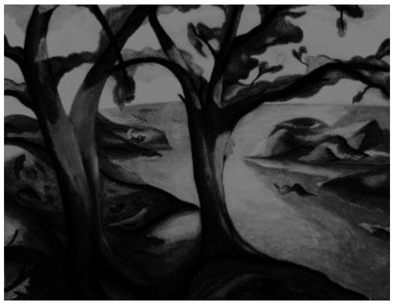


Image A

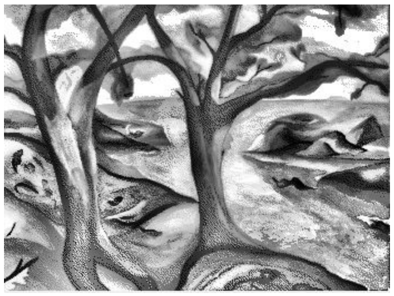


Image B

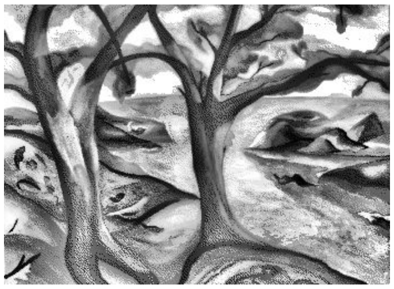
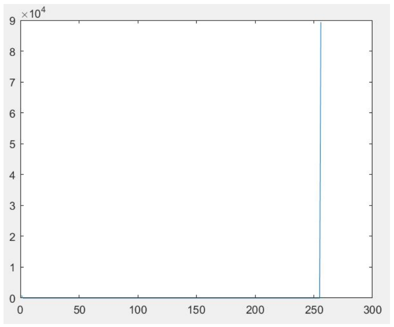
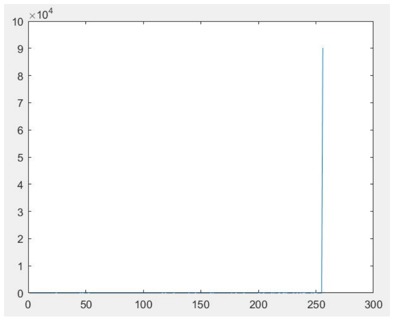


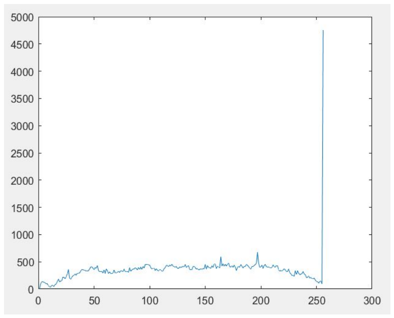
Image C



Histogram of image A



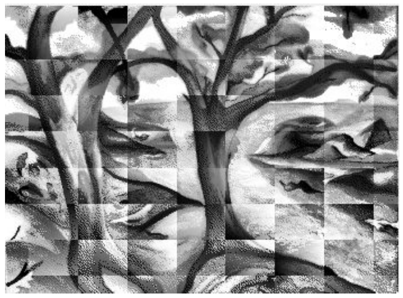
Histogram of image B



Histogram of image C

Image	A	B	C
Description	Input image	My output image	System output image
Dimensions	258 x 350	258 x 350	258 x 350
Mean gray level	49.05	140.48	140.61
Error (RMSE)	Between B and C: 7.54		

Note: The Matlab internal function implements clip-adjusted adaptive histogram equalization. Setting clip limit = 1 during function call makes it equivalent to adaptive histogram equalization.



The "blocky" effect mentioned earlier, which could arise if we didn't use bilinear interpolation to find our pixel values. Across the edge of each tile, the transformation function changes, thus leading to sudden intensity jumps across a tile, leading to a "blocky" effect.