EN2550 Fundamentals of Image Processing and Machine Vision
Assignment 01

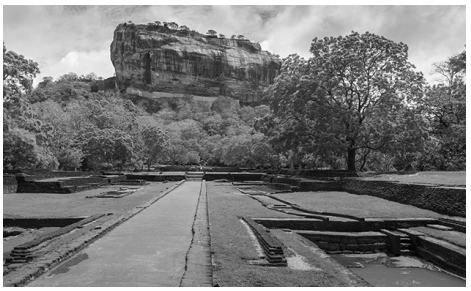
180616T P.M.P.H.Somarathne

full code available at `https://github.com/PamudithaSomarathne/EN2550/tree/master/1%20Basics`

1 Basic processing

Here, we'll carry-out the point operations: histogram equalization, intensity windowing, gamma correction and spatial filtering techniques: unsharp masking, Gaussian filtering, median filtering and bilateral filtering on the gray-scale or color image of Sigiriya.

(a) *Color image*

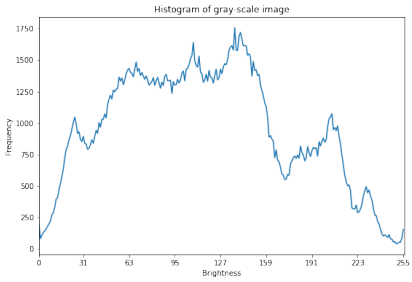


(b) *Gray-scale image*

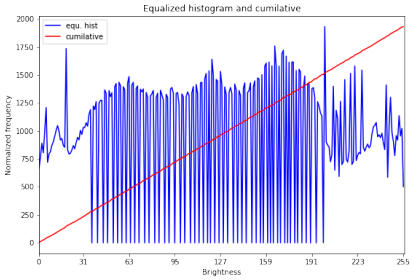
Figure 1: Original images

1.1 Histogram Equalization

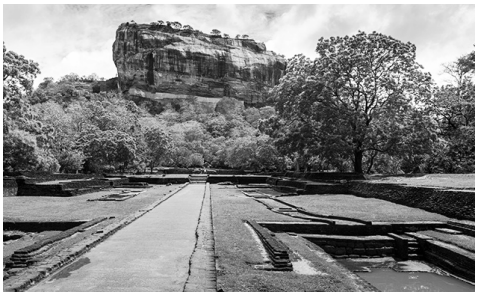
We equalize the histogram to have a linear cumulative, to get an image with better highlights.



(a) *Histogram of original image*



(b) *Equalized histogram(blue) and cumulative(Red)*

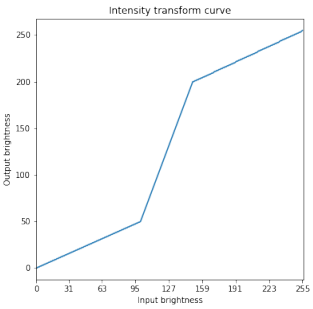


(c) *Image after equalizing the histogram*

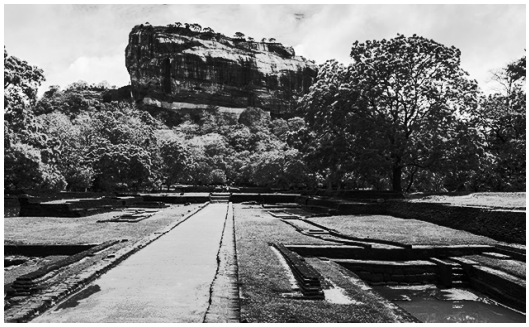
Figure 2: Process of histogram equalization

1.2 Intensity windowing

In intensity windowing we map each brightness level to a new level using a windowing function like the one shown in fig (3a). These can be used to make arbitrary effects like improving a brightness range or highlighting a specific range in the histogram. The transform used here is has a linear curve between the points (0,0)-(100,50), (100,50)-(150,200) and (150,200)-(255,255)



(a) *Intensity windowing function*

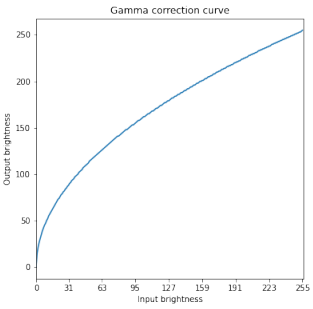


(b) *Image after applying intensity transform*

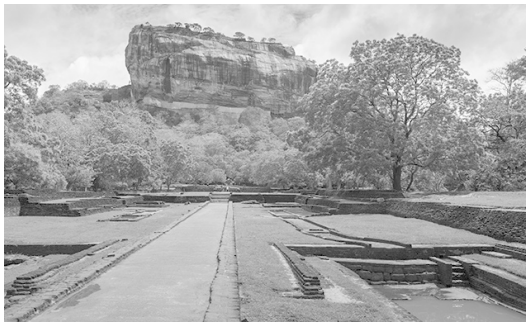
Figure 3: Intensity windowing

1.3 Gamma correction

In gamma correction, brightness levels are given to new values using $g = 255(\frac{f}{255})^{\gamma}$ function. When $\gamma < 1$ the dark pixels are mapped into a larger range while bright pixels are mapped into a smaller range. This has the overall effect of increasing the brightness of the image as most of the pixels get values higher than their previous value. $\gamma > 1$ gives the opposite effect. $\gamma = 1$ is the identity transform. The following image is corrected with $\gamma = 0.5$ and the gamma function calculated for this is shown in fig 4a.



(a) *Gamma function*



(b) *Gamma corrected image*

Figure 4: Gamma correction

1.4 Gaussian filtering

Gaussian filtering is used to remove random Gaussian noise from an image. Here a Gaussian kernel is generated with a given σ and convolved with the noisy image. Larger σ values can remove more noise but the output will be blurrier. Here we generate a noisy image(fig 5a) with $\sigma = 0.05, \mu = 0$ and filter it using the Gaussian kernel(size = 5, $\sigma = 2$).

(a) *Noisy image used for filtering*

(b) *Image after Gaussian filtering*

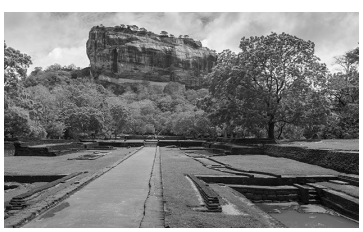
Figure 5: Gaussian Filtering

```
noise = np.random.normal(mean,sigma,image.shape)*255
noisy_image = cv.normalize(cv.add(image.astype(np.float),noise),\
        None, 0, 255, cv.NORM_MINMAX).astype(np.uint8)
gaussian_kernel = cv.getGaussianKernel(kernel_size, filter_sigma)
filtered_image = cv.sepFilter2D(noisy_image, -1, gaussian_kernel, gaussian_kernel)
```

1.5 Unsharp masking

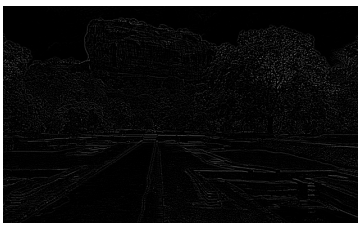
Unsharp masking is a method used to sharpen an image. First, a blurred version of the image is taken(ref fig 6b, blur size = 5, $\sigma = 2$) and subtracted from the original image(ref fig 6c). The sharpened image is obtained by adding a weighted version the difference image to the original image. The logic behind unsharp masking is that the difference between an image and its blurred version is the sharp content of that image. Therefore, by adding the difference image, we are adding sharp content to the image.

```
blur_kernel = cv.getGaussianKernel(blur_size, blur_sigma)
blurred_image = cv.sepFilter2D(image, -1, blur_kernel, blur_kernel,\
        anchor=(-1,-1), delta=0, borderType=cv.BORDER_REPLICATE)
difference_image = image.astype(np.float32) - blurred_image.astype(np.float32)
sharpened_image = cv.addWeighted(image.astype(np.float32), 1.0,\
        difference_image, 0.7, 0).astype(np.uint8)
```



(a) *Original image*

(b) *Blurred image*



(c) *Difference image*

(d) *Sharpened image*

Figure 6: Steps of unsharp masking

1.6 Median filtering

Median filtering is applied to remove salt-pepper noise which cannot be effectively removed using Gaussian filtering. Median filtering is a non-linear filtering method where each pixel is given the median value from its original image. Since salt-pepper noise dots are outliers in the image, they'll be removed easily. Here, we generate a noisy image with 10% noise with equal amounts of white and black noise(fig 7a). Then we use a median filter of size 3 to remove this noise effectively.

```
median_filtered_image = cv.medianBlur(salt_image, median_filter_size)
```

(a) *Image with salt-pepper noise*

(b) *Image after applying median filtering*

Figure 7: Median Filtering

1.7 Bilateral filtering

Bilateral filtering is a non-linear filter used to remove noise from images. It computes the new value of the pixel with a weighted average of its original neighbors. This weights depend no only on the radial distance but also parameters like RGB value, depth etc. Bilateral filtering has little effect on the edges because of this extra parameters. Therefore, the bilateral filtered images are sharper than the Gaussian filtered images.

```
bilateral_filtered_image = cv.bilateralFilter(noisy_image,\
        bilateral_filter_size, bilateral_sigma_color, bilateral_sigma_space)
```

(a) *Noisy image generated for filtering*

(b) *Image after applying bilateral filtering*

Figure 8: Bilateral Filtering

2 Counting rice grains in the given image

We have count the number of rice grains in the given figure 9. First step is to generate a binary image which shows rice grains separate from the background. Next, we apply connected component analysis to detect each rice grain individually. Original image has considerable noise along with uneven lighting. So, the histograms for noise and rice grains overlap and this prevents the use of a simple threshold method to get binary image. Here, we'll take two approaches, one by using techniques mentioned in "part 1" above. Other approach will use an adaptive filter along with the threshold operation.

Figure 9: Rice.png

2.1 Approach 01: Using basic techniques to get the binary image

First, I balanced the uneven lighting condition by adding weights to the darker areas. After brightness adjustment, the brightness levels of all the noise aligned together and levels of all rice grains aligned together. Then it was easier to threshold the image at brightness level 155. After threshold operation, cv.connectedComponents() was used to detect the rice grains individually. This approach detected 99 rice grains. Two pairs of grains have been detected as single grains since they were close in the image and thresholding combined them into one.

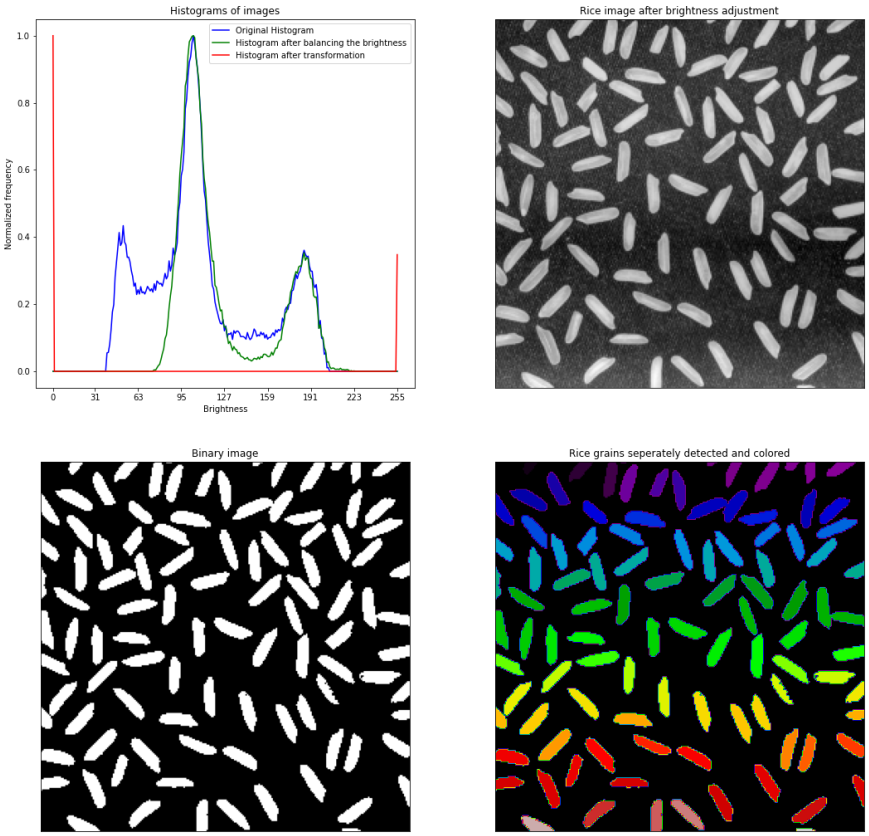


Figure 10: Connected component analysis - approach 1

```
T1, briAdj = np.zeros(256,np.uint8),np.zeros(rice_image.shape,dtype=np.uint8)
T1[155:] = 255
for i in range(100): briAdj[i+156,:] = i/1.3
rice_image = cv.add(rice_image,brightnessAdjust)
enhanced = T1[rice_image]
num_labels,labels = cv.connectedComponents(enhanced, connectivity = 4)
```

2.2 Approach 02: Using adaptive threshold to get the binary image

We can use cv.adaptiveThreshold to get a thresholding along with Gaussian smoothing. The binary image generated using this method detected 101 grains. However, there is a noise dot detected as a grain and two grains have been detected as one. Therefore, even-though the number of grains is correct, the grains have not been detected with 100% accuracy.

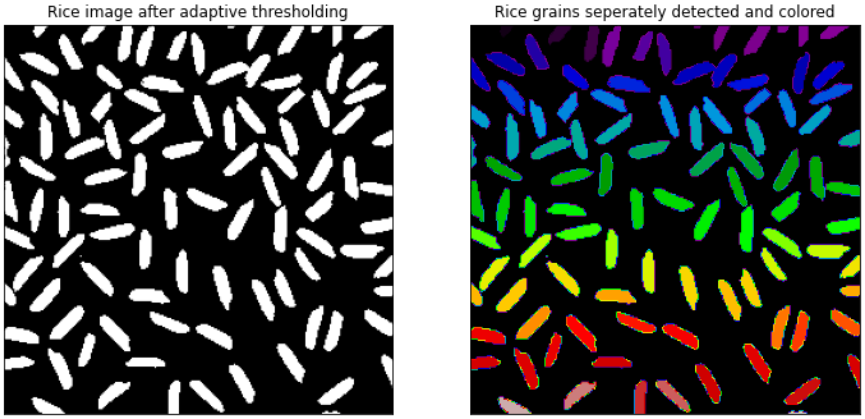


Figure 11: Connected component analysis - approach 2

```
enhanced_image = cv.adaptiveThreshold(rice_image, 255,\
        cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY, 61, -30)
num_labels2,labels2 = cv.connectedComponents(enhanced_image, connectivity = 4)
```

2.3 Connected component analysis

Connected component analysis from graph theory can be used to detect items separately from a binary image. The algorithm categorizes the pixels and their neighbors with the same value with the same label. Categorization can be done for images with either 4-connectivity or 8-connectivity basis. In 4-connectivity mode, first each pixel is given a label based on its top and left neighbors. Then the labels are merged by considering all 4 neighbors.

3 Image zooming

3.1 Nearest neighbors

In nearest neighbors interpolation, the pixel value of the zoomed image is equal to the pixel that is nearest to it when mapped into the original image. It can be implemented in two ways: taking floor value for non-integer indexes when mapping & rounding off the non-integer indexes when mapping. These two methods give slightly different results. The code is as follows

Floor values as index

```
for x in np.arange(scaled_width):
  for y in np.arange(scaled_height):
        x_0,y_0 = int(x/scale),int(y/scale)
        scaled_image[x,y] = image[x_0,y_0]
```

Rounded index

```
for x in np.arange(scaled_width):
  for y in np.arange(scaled_height):
    x_0,y_0 = np.round(x/scale).astype(int),np.round(y/scale).astype(int)
        if (x_0==image.shape[0]): x_0 = x_0-1
    if (y_0==image.shape[1]): y_0 = y_0-1
        scaled_image[x,y] = image[x_0,y_0]
```

3.2 Bi-linear interpolation

In bilinear interpolation, for each pixel the corresponding index in the original image is calculated. If the index is an integer, the pixel value is directly assigned. For non-integer indexes, the pixel value is calculated with bilinear approximation from the four nearest points with integer indexes. If the pixel that need to determined is located at (x_i, y_i) and the four nearest integer indexed pixels have coordinates $(x_0, y_0), (x_0, y_1), (x_1, y_0), (x_1, y_1)$, then

$$Im[x_i, y_i] = (y_1 - y_i)[(x_1 - x_i)Im[x_0, y_0] + (x_i - x_0)Im[x_1, y_0]] \\ +(y_i - y_0)[(x_1 - x_i)Im[x_0, y_1] + (x_i - x_0)Im[x_1, y_1]]$$

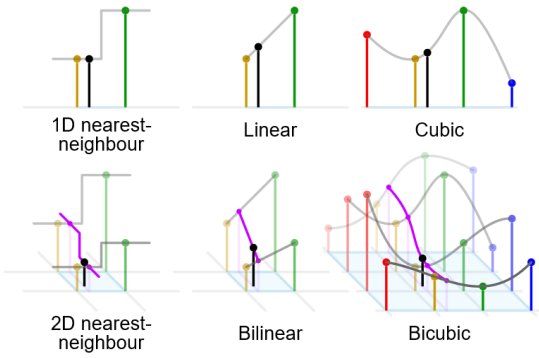


Figure 12: Bilinear interpolation vs other interpolation methods [3]

```
for x in np.arange(scaled_width):
  for y in np.arange(scaled_height):
    x_i,y_i = x/scale,y/scale
    x_0,y_0 = int(x_i),int(y_i)
    if (x_i==x_0 and y_i==y_0): scaled_image[x,y] = image[x_0,y_0]
    else: x_1,y_1 = x_0+1,y_0+1
      if (x_1==image.shape[0] or y_1==image.shape[1]):
        scaled_image[x,y] = image[x_0,y_0]
      else: scaled_image[x,y] =\
        (y_1-y_i)*((x_1-x_i)*image[x_0,y_0]+(x_i-x_0)*image[x_1,y_0]) +\
        (y_i-y_0)*((x_1-x_i)*image[x_0,y_1] + (x_i-x_0)*image[x_1,y_1])
```

3.3 Zooming results

First, let's look at the percentage sum of squared difference(SSD%) of five different implementations.

Image	NN coded(%)	NN coded with np.round(%)	Bilinear coded(%)	cv NN(%)	cv Bilinear(%)
im01.png	0.21	0.393	0.308	0.21	0.177
im02.png	0.041	0.099	0.075	0.041	0.028
im04.png	1.211	1.531	1.39	1.211	1.204
im05.png	0.44	0.588	0.517	0.44	0.43
im06.png	0.289	0.492	0.393	0.289	0.249
im07.png	0.289	0.413	0.359	0.289	0.282

Table 1: The SSD%s of different images when zoomed

The cv implementation for nearest neighbors has same SSD as the nearest neighbors implementation with floor values. The rounded nearest neighbors implementation has highest SSD. Bilinear implementation also has relatively high SSD. The reason for these two errors can be the inability to process the edges correctly. The cv implementation of bilinear interpolation has lowest SSD.
Below are the outputs for the two images with lowest and highest SSD%s from nearest neighbors implementation with floor value and bilinear implementation. Image 04 has a sharp background in the hi-res image. The zoomed images cannot effectively capture this sharp details and hence have higher errors. When zooming into the NNzoom images, you'll observe large squares of same color generated by the nearest neighbors interpolation. If you zoom into Bizoom images, you'll see the blurred edges due to the averaging done by bilinear interpolation

(a) *Hi-res image* (b) *NNzoom image* (c) *Bizoom image*

Figure 13: Image 02

(a) *Hi-res image* (b) *NNzoom image* (c) *Bizoom image*

Figure 14: Image 04

Note: You can zoom these images to view them with finer details.

References

[1] Bilateral filter - Wikipedia
`https://en.wikipedia.org/wiki/Bilateral_filter`

[2] Connected Component Labeling - Wikipedia
`https://en.wikipedia.org/wiki/Connected-component_labeling`

[3] Bilinear Interpolation - Wikipedia
`https://en.wikipedia.org/wiki/Bilinear_interpolation`