Department of Electronic and Telecommunication Engineering

University of Moratuwa, Sri Lanka

EN2550 - Fundamentals of Image Processing and Machine Vision

Assignment 01

Submitted by

Thalagala B.P. 180631J

Submitted on

March 1, 2021

Contents

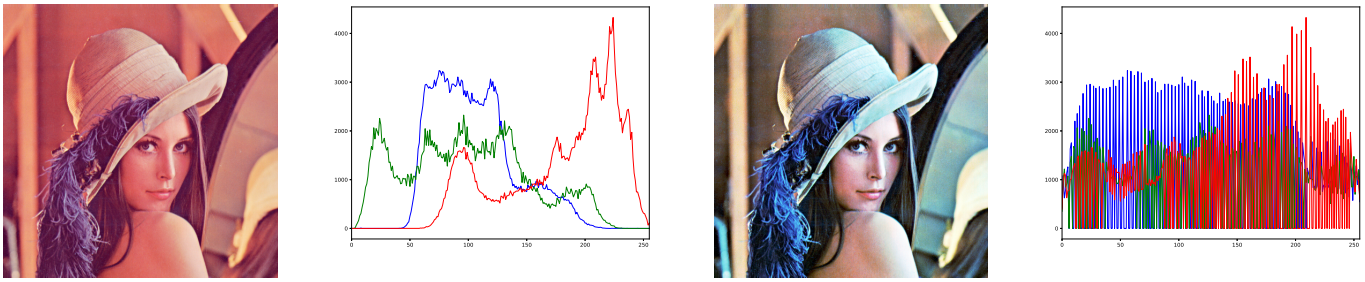
**** PDF is clickable***

Without the title page and the content table page there are only 6 pages in this Assignments. Only the important parts of the codes are given here due to page limitation. Complete executable code can be found at https://github.com/bimalka98/Computer-Vision-and-Image-Processing/blob/main/EN2550Assignments/A1/180631J_a01.ipynb

1 Part 1: Basic Operations

1.1 Histogram computation and Histogram equalization

Histogram of an image represents the intensity distribution(*intensity vs number of pixels*) over the intensities[0, 255]. Histogram equalization makes uneven histograms more or less flat. Figure (b) depicts the uneven histogram of the original image. There, most of the red pixels are in the right region(*biased to the right*) which makes the original image looks more reddish than natural. Through histogram equalization as depicted in the figure (d) the distribution can be made more uniform and it has given the image a more natural look.



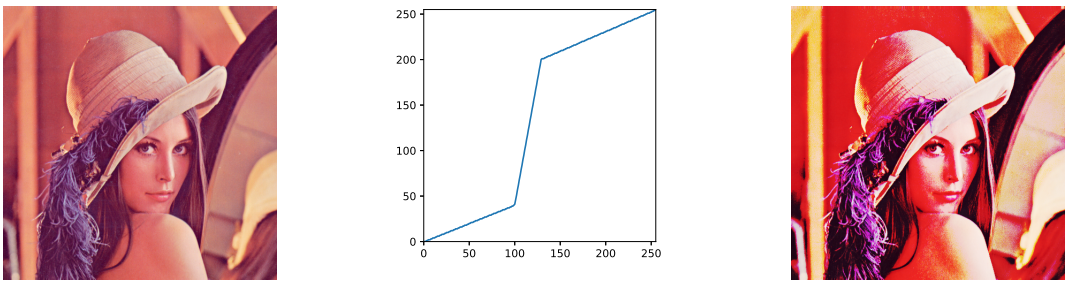
(a) Original Image (b) Hist: of the Original Image (c) Hist: Equalized Image (d) Equalized Histogram

Figure 1: Histogram computation and Histogram equalization

[1]:
```
channels = ['b', 'g', 'r']# since opencv reads img as BGR channel order
for i, channel in enumerate(channels):
    histogram = cv.calcHist([img], [i], None, [256], [0,256])# Calculating Histogram of each channel.
equalized_channels = []
for channel in cv.split(img):
    equalized_channels.append(cv.equalizeHist(channel))# Each channel must be equalized seperately.
equalized_img = cv.merge(equalized_channels)
```

1.2 Intensity transformations

Intensity transformations is basically a look up table operation. It maps each pixel's intensity to some other intensity depending on a predefined transformation function which only depends on the pixel of interest. Following figures illustrates a transformation consists of three sections(*piece-wise continuous function*). It maps intensities in the ranges [0, 99],[100, 129] and [130, 255] into the ranges [0, 40], [41, 200] and [201, 255] respectively as shown in the transformation function. This kind of transformation is called as ***Intensity windowing.***



(a) Original Image (b) Transformation (c) Transformed Image

Figure 2: Intensity transformations

[2]:
```
a, b = [40,200]
tr1 = np.linspace(0,a,100).astype('uint8')# Declaring tranformations for each region
tr2 = np.linspace(a+1,b,30).astype('uint8')
tr3 = np.linspace(b+1,255,126).astype('uint8')
transf = np.concatenate((tr1, tr2), axis = 0)# Concatenating above transfer functions
transf = np.concatenate((transf, tr3), axis = 0)
transformed_img = cv.LUT(img, transf)# Apply Transformation
```

1.3 Gamma correction

Image transformation Depends on the value of γ as follows,

- If $0 < \gamma < 1$, then function maps narrow range of dark pixels to a wider range of dark pixels which increases the brightness of the image as depicted in figures (c) and (d).
- If $\gamma > 1$, then opposite of the above process happens and it makes the image darker as in the figure (a).

(a) Gamma Corrected Image $\gamma = 1.3$ (b) Original Image $\gamma = 1$ (c) Gamma Corrected Image $\gamma = 0.7$ (d) Gamma Corrected Image $\gamma = 0.4$

Figure 3: Gamma correction

[3]:
```
def gammacorrect(image, gamma):
    # Calculating the lookup table for a given gamma value.
    table = np.array([(i/255.0)**(gamma)*255.0 for i in np.arange(0,256)]).astype('uint8')
    img_gamma = cv.LUT(image, table)
    return img_gamma
```

1.4 Gaussian smoothing $\sigma = 1.5$

Gaussian smoothing is used to reduce sharp transitions in intensities in images(blurring). The extent of blurring depends on both the kernel size and the standard deviation σ. Each pixel's value is recalculated as the weighted sum of its neighbor pixels where weights are determined by the Gaussian Normal function. Weight decreases as the distance between central pixel and a given pixel increases. Also if the σ increases, weights given to the neighbor pixels increases and their contribution to determine the central pixel value increases which makes the image more blur. The effect of kernel size is depicted in the following figures. As kernel size increase more pixels contribute to the calculation of central pixel's value which makes the image more blur.

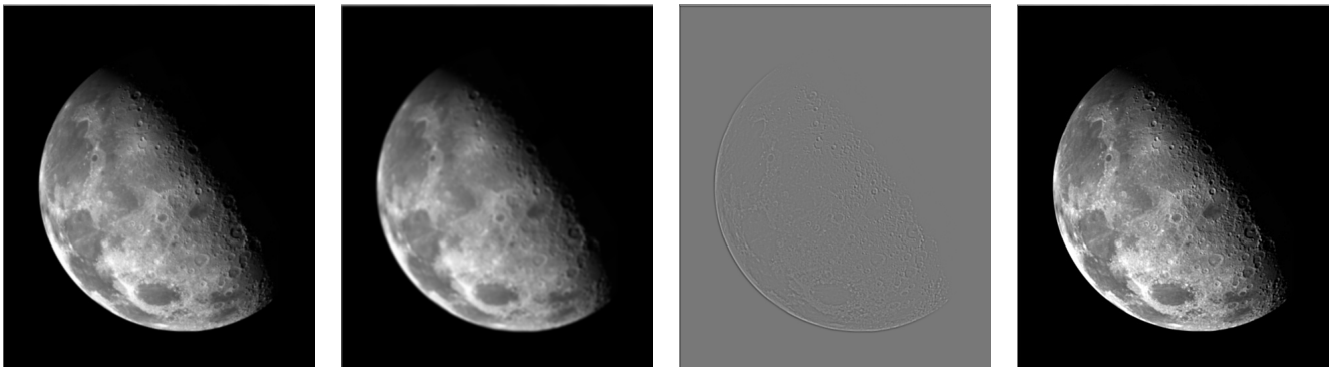
(a) Original Image (b) Kernel Size = 7 (c) Kernel Size = 13 (d) Kernel Size = 21

Figure 4: Gaussian smoothing $\sigma = 1.5$

[4]:
```
def gaussianSmooth(image, kernelSize, sigma):
    max_abs = np.floor(kernelSize/2) # If kernelSize = 11
    x_range = np.arange(-max_abs,max_abs +1,1) #(from -5 to +5 range)
    y_range = np.arange(-max_abs,max_abs +1,1) #(from -5 to +5 range)
    X,Y = np.meshgrid(x_range, y_range)
    kernel = np.exp((-(X**2 + Y**2))/(2*sigma**2))/(2*np.pi*sigma**2)# Bulding Gaussian kernel
    smoothed_img = cv.filter2D(image,-1,kernel)# Convolution of the image with the kernel
    return smoothed_img
```

1.5 Unsharp masking

Unsharp masking is an image sharpening method which is used to highlight intensity transitions in the image. It has following three main steps as depicted in the figures. **(1.)** Original image is blurred using an averaging kernel(*here Gaussian kernel is used.*). **(2.)** Blurred image is subtracted form the original image to obtain the "mask". **(3.)** "Mask" is then added to the original image to obtain the sharpened image. To eliminate overflow occurs in the addition of two images OpenCV's `addWeighted` function is used.



(a) Original Image (b) Gaussian Blurred (c) Mask + 125 (d) Sharpened Image

Figure 5: Unsharp masking

[5]:
```
sigma = 2
kernel = cv.getGaussianKernel(5, sigma) #create the gaussian kernel
blurred = cv.sepFilter2D(img, -1, kernel, kernel, anchor=(-1,-1), delta=0, borderType=cv.
 ↪BORDER_REPLICATE)
mask = img.astype('float32') - blurred.astype('float32')
k = 1 # when k =1 : unsharp masking, when k>1 : Highboost Filtering
sharpened = cv.addWeighted(img.astype('float32') , 1., mask, k, 0)
```

1.6 Median filtering

Non-linear Median filter is heavily used to remove random noises like salt and pepper noise from images due to its excellence noise-reduction capability over any other averaging filter. As its name implies it replaces value of the central pixel by the median of the intensity values in the neighborhood of that pixel. As following figures depict noise in the figure (b) is completely removed by a 5×5 median filter which is not possible with an averaging filter like Gaussian.

(a) Original Image (b) Adding Salt & Pepper Noise (c) Kernel Size = 3 (d) Kernel Size = 5

Figure 6: Median filtering

[6]:
```
saltpepper = noisy("s&p", img)# noise generated using noisy fun @:stackoverflow.com/a/30609854
kernelSize = 3
medianfiltered_image = cv.medianBlur(saltpepper, kernelSize)
```

1.7 Bilateral filtering

Bilateral filter extends the idea of Gaussian smoothing by adding ***edge preserving capability*** which is not possible in the Gaussian filtering because there, only the Spatial distance to the pixels' from the central pixel were considered when

applying the weights(*Same Gaussian kernel everywhere regardless of the pixel's intensity*) and therefore ***Gaussian filter averages across the edges*** assuming that pixels' intensity value do not change rapidly over the window. As a consequence the same weight may be applied to a pixel on an edge(say high intensity) and a pixel near an edge(say low intensity) when calculating the central pixel's value of a window centered at a pixel near an edge.

In the bilateral filter ***intensities of the neighborhood pixels are also considered*** when applying the weights, in addition to the spatial distance. This makes it possible to ***eliminate the averaging across edges***. As following figures shows when σ_r approaches ∞, bilateral filter gives the result of the Gaussian filter. Here σ_s is spatial deviation(*spatial extent of the kernel*) while σ_r is the color space deviation(*minimum amplitude of an edge*).

(a) Original Image (b) Gaussian Kernel Size = 9, $\sigma_s = 6$ (c) Bilateral Kernel Size = 9, $\sigma_s = 6$, $\sigma_r = 100$ (d) Bilateral Kernel Size = 13, $\sigma_s = 6$, $\sigma_r = 10$

Figure 7: Bilateral filtering

[7]:
```
#dst = cv.bilateralFilter(src, d(KernelSize), sigmaColor, sigmaSpace)
bilateral1 = cv.bilateralFilter(img,9,100,6)# Edges are blurred. Almost Gaussian
bilateral2 = cv.bilateralFilter(img,13,10,6)# Edges are preserved
```

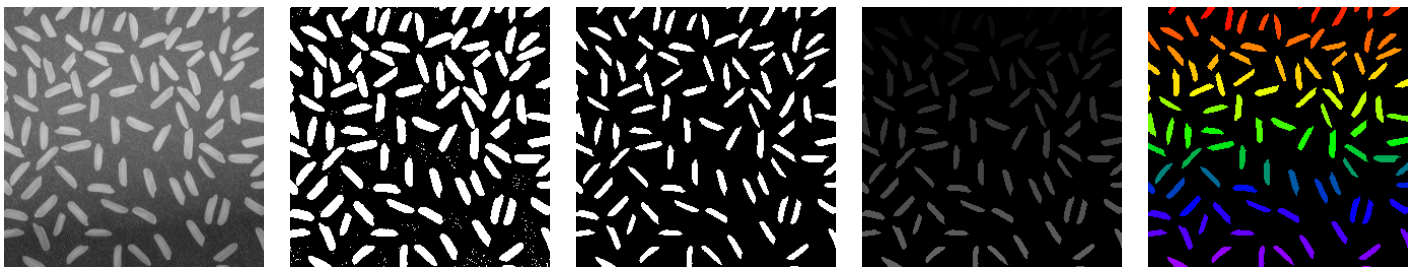
---

2 Part 2: Count the rice grains in the rice image

The objective of counting the rice grains can be achieved through binary segmentation as we have only two different object types namely rice grains and the background. Since the image has different lighting conditions, ***adaptive thresholding*** must be used for binary segmentation. Optimum parameter values for the thresholding function was obtained though trial and error. To get rid of the unwanted white noise in the background and to detach connected grains as depicted in the figure (b) ***erosion morphological transformation*** is used(*Rectangular kernel is used over elliptic and cross shape kernels since it was found to be the best at achieving the aforementioned requirements*).

After the above post-processing what is left is just to count the distinct white objects(rice grains) in the figure (c). For that Connected Components Analysis(CCA) is used. It gives each distinct grain a distinct label(*an integer starting form 0(for background) to some number*). This can be done through OpenCV's `connectedComponents` function which returns total number of labels and the labeled image. Since this total number of labels includes the background as well, we need to subtract 1 from it to get the grain count. All the components in the labeled image can be easily visualized through a proper color map as depicted in figure (e). Before map the colors, figure (d) is normalized to get a uniform intensity distribution in the range[0, 255] which makes the color mapping more distributed.

`Through the following algorithm it was found that there are 100 grains in the given image.`



(a) Gray Scale Image (b) Adaptive Thresholding (c) Erosion (d) Connected Components (e) Color Mapped Image

Figure 8: Counting the rice grains

[8]:
```
# reading the image as an eight bit grayscale image
img = cv.imread("../a01images/rice.png", cv.IMREAD_GRAYSCALE)
# adaptive thresholding due to non uniform illumination in the image
kernelSize, C = 25,-10
img_adapt_thresh = th2 = cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_MEAN_C, cv.
 ↪THRESH_BINARY,kernelSize, C)
# Morphological transf: Erosion to eliminate white noise and detach connected objects
ksize = 3
kernel = cv.getStructuringElement(cv.MORPH_RECT,(ksize,ksize))
eroded_img = cv.erode(img_adapt_thresh, kernel)
#-----------------Connected components Analysis(CCA)----------------------
num_labels, labeledImg = cv.connectedComponents(eroded_img)
# Background is considered as another object in CCA. Therfore it needs to be subtracted
num_grains = num_labels -1; print(num_grains)
#-----------------Show components using a color Map.----------------------
(minVal, maxVal, minLoc, maxLoc) = cv.minMaxLoc(labeledImg)# find min,max pixel values & locations
labeledImg = (255/(maxVal-minVal)) * (labeledImg - minVal)# Normalizing image so, min= 0 & max = 255.
imgColorMap = cv.applyColorMap(labeledImg.astype('uint8'), cv.COLORMAP_RAINBOW)# Applying color map
imgColorMap[labeledImg==0] = 0# Making the background black
plt.imshow(imgColorMap[:,:,::-1])# Display colormapped labels
```

---

3 Part 3: Zoom Images

Following code implements ***Nearest Neighbor*** and ***Bi-linear Interpolation*** methods of image zooming. As following figure depicts bi-linear interpolation method is better than nearest neighbor method since it interpolates each pixel's intensity value using its neighbors' values rather than simply taking the nearest neighbor's value when assigning values to the pixels in the zoomed image. Because of this reason continuous nature of intensities is violated and pixel's rectangular structure is clearly visible in the image, zoomed using the nearest neighbor method which is not the case in the bi-linear interpolation method.

(a) Nearest Neighbor Method

(b) Bilinear Interpolation Method

Figure 9: Zooming Images

These two methods can be easily explained using a count example as follows. Consider a pixel in a zoomed image(1920×1200) whose coordinates are $(235, 526)$. Assume this image was obtained by scaling an image(480×300) by a factor of 4.

- First step is common to the both methods, that is calculating the corresponding pixel in the original source image by dividing the coordinates by the scaling factor. Therefore source coordinates = $(235/4, 526/4) = (58.75, 131.5)$
- In the `Nearest Neighbor Method` these coordinates are rounded to the nearest integers within the source image dimensions. Then corresponding pixel in the source image = $(59, 132)$ and intensity of that pixel is assigned to the pixel$(235, 526)$ in the zoomed image.
- But in the `Bi-linear Interpolation Method` it considers intensities of all of its neighbors. That is intensities of, $(58, 131)$,$(58, 132)$,$(59, 131)$ and $(59, 132)$ are considered and it interpolates the intensity of an imaginary pixel

at the coordinates$(58.75, 131.5)$ using those intensities. That intensity value is then assigned to the corresponding pixel$(235, 526)$ in the zoomed image.

Because of this difference in intensity assigning procedure, Bi-linear Interpolation Method is computationally expensive and its quality is much better than the Nearest Neighbor Method.

[9]:
```
def zoom(image, scaling_factor, method):
    img = image
    sf = scaling_factor
    # Determining dimensions of the zoomed image
    if len(img.shape) == 2: # for GRAYSCALE images
        zoomedImgDims = [int(dim*sf) for dim in img.shape]
    else: # for COLOR images
        zoomedImgDims = [int(dim*sf) for dim in img.shape]
        zoomedImgDims[2] = 3
    zoomedImg = np.zeros(zoomedImgDims, dtype = img.dtype)# declaring an empty array to store values
#====================Nearest Neighbour Mehtod(NNM)=========================
    if method == 'nn':
        for row in range(zoomedImg.shape[0]):
            source_row = round(row/sf)# Calculating corresponding pixels in original image
            if source_row > img.shape[0]-1: source_row = img.shape[0]-1# Overflow handling
            for column in range(zoomedImg.shape[1]):
                source_column = round(column/sf)# Calculating corresponding pixels in original image
                if source_column > img.shape[1]-1: source_column = img.shape[1]-1# Overflow handling
                # Assigning pixel values
                if len(img.shape) == 2:
                    zoomedImg[row][column] = img[source_row][source_column]
                else:
                    for channel in range(3):
                        zoomedImg[row][column][channel] = \
                                            img[source_row][source_column][channel]
#======================Bilinear Interpolation Mehtod(BIM)==================
    if method == 'bi':
        for row in range(zoomedImg.shape[0]):
            row_position = row/sf# Calculating corresponding row in original image
            row_below = int(np.floor(row_position))
            row_up = int(np.ceil(row_position))
            if row_up > img.shape[0]-1: row_up = img.shape[0]-1# Overflow handling
            for column in range(zoomedImg.shape[1]):
                column_position = column/sf# Calculating corresponding column in original image
                column_previous = int(np.floor(column_position))
                column_next = int(np.ceil(column_position))
                if column_next > img.shape[1]-1: column_next = img.shape[1]-1

                diff1 = row_position - row_below
                diff2 = column_position - column_previous
                if len(img.shape) == 2: # for GRAYSCALE images
                    interVal1 = img[row_below][column_previous]*(1-diff1)\
                                + img[row_up][column_previous]*(diff1)
                    interVal2 = img[row_below][column_next]*(1-diff1)\
                                + img[row_up][column_next]*(diff1)
                    zoomedImg[row][column] = (interVal1*(1-diff2)\
                                              + interVal2*(diff2)).astype('uint8')
                else: # for COLOR images
                    for channel in range(3):
                        interVal1 = img[row_below][column_previous][channel]*(1-diff1)\
                                + img[row_up][column_previous][channel]*(diff1)
                        interVal2 = img[row_below][column_next][channel]*(1-diff1)\
                                + img[row_up][column_next][channel]*(diff1)
                        zoomedImg[row][column][channel] = (interVal1*(1-diff2)\
                                                           + interVal2*(diff2)).astype('uint8')
    return zoomedImg
```