EN2550 — Fundamentals of Image Processing and Machine Vision

Student Name: Nuwan Bandara
Student Number: 180066F

Submitted Date: March 2, 2021
Assignment Number: 1

Problem 1 Execution of defined primary operations on the selected image: given img01.png

The initial task is to display the selected image using Python OpenCV and matplotlib libraries. Here, it is essential to consider the input color format of each library since OpenCV utilizes BGR format while matplotlib implements RGB format.

Entire code flow is executed and accessed on the Google Colaboratory platform (Colab) via, `https://colab.research.google.com/drive/1PDDkABUmk4vBPT--yPF9T3t8Pep9EQdS?usp=sharing`

```
import cv2
import matplotlib.pyplot as plt
img = cv2.imread ( "/content/im01.png" , cv2.IMREAD_COLOR)
img_new = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  #color_conversion since we use matplotlib library for plotting
plt.rcParams['figure.figsize'] = [12, 12]  #change the default plot size
fig, ax = plt.subplots(figsize=(12,12))
ax.imshow(img_new)
ax.set_title("Selected Image: im01.png")
plt.show()
```

Figure 1: Python code for displaying the selected using using necessary libraries

(a) Histogram computation.
In the image processing, a histogram represents the intensity distribution over a defined range of intensities of an image. The inbuilt function, *calchist* has been utilized for this purpose which gets the image, channels, mask, histogram size and ranges as its inputs.

```
color = ('b','g','r')
for i,c in enumerate(color):
  hist = cv2.calcHist([img],[i], None, [256],[0,256])
  plt.plot(hist, color = c)
  plt.xlim([0,256])
plt.show()
```

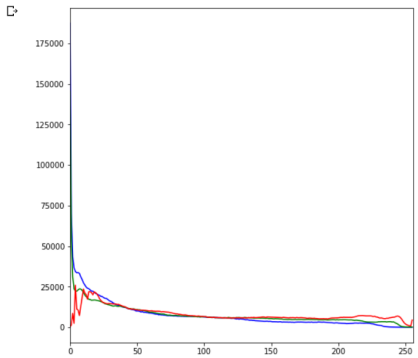
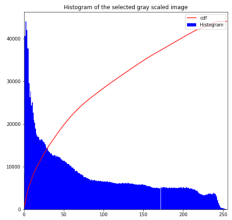
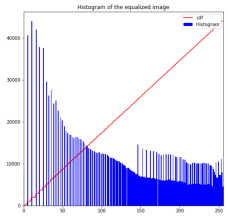


Figure 2: Histogram representation of im01.png with corresponding python implementation

(b) Histogram equalization. Show the histogram before and after.
Histogram equalization is considered as a gray-level transformation that results in an image with a more or less flat histogram. The main in-buit function for histogram equalization is *equalizeHist* which takes a gray-level image as the input.



(a) Histogram of the selected gray-scaled image



(b) Histogram of the equalized image

(c) Results comparison

Figure 3: Histogram equalization

(c) Intensity transformations. Show the transformation function as well.
Intensity transformation is an image processing technique in which the output is only dependant on the input value such that $g(x) = T(f(x)$. Here, three different transformations have been implemented: Identity ($g(x) = f(x)$), Negative ($g(x) = -f(x)$) and Intensity windowing with,

$$g(x) = \begin{cases} \frac{5}{101} f(x) & \text{if 0 to 100} \\ \frac{49}{10} f(x) & \text{if 101 to 150} \\ \frac{1}{21} f(x) + 200 - \frac{50}{7} & \text{if 151 to 255} \end{cases} \quad (1)$$

(d) Gamma correction. State γ.
This is a power-law transformation with $\gamma = 3$ which satisfies $g = f^{\gamma}$ $(f \in (0, 1))$

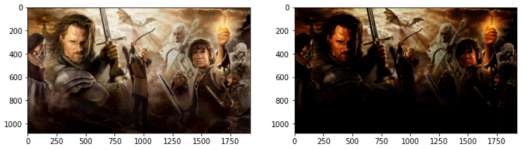
(e) Gaussian smoothing. State kernel size and σ.
Inbuilt *GaussianBlur* is implemented using the kernel size of (1211,1211) (for better observation of blurring effect) with equal σ value of 5. This smoothing technique is very effective in images which has more Gaussian noise.

(f) Unsharp masking.
This is implmented using *addWeighted* function (input parameters of α and β for adding two images weightedly) along with *GaussianBlur* inbuilt function.

(g) Median filtering. State kernel size.
The median filter operates over a window by selecting the median intensity in the window. *medianBlur* is utilized with a kernel size of (11,11). This is considered to be highly effective against salt-and-pepper noise in an image.

(h) Bilateral filtering. Explain the theory of this as well.
Bilateral filtering is highly efficient in removing the noises in the images while keeping the edges of those images undisturbed (not loosing its sharp edges). This unique capability of bilateral filter has been obtained through two Gaussian filter: Gaussian filter in space and the Gaussian filter which is a function of pixel difference, such that, the Gaussian function of space ensures that only nearby pixels are considered for blurring, while the Gaussian function of intensity difference ensures that only those pixels with similar intensities to

(a) After intensity windowing (gray image)



(b) After gamma correction

Figure 4: Result image comparison

(a) After Gaussian smoothing

(b) After unsharp masking with $\alpha = 1.5$ and $\beta = -0.5$

(c) After median filtering with kernel (11,11)

Figure 5: Result images after filtering

the central pixel are considered for blurring. This manipulates the preserving of edges since the pixels at the edges have large intensity variation.

The inbuilt *bilateralFilter* gets several input arguments such as source image, destination image, diameter of each pixel neighbourhood, standard deviations in the color space and the coordinate space to execute this filtering.

Figure 6: Result image after bilateral filtering

```
thrh, output = cv2.threshold(img_rice, 127, 255, cv2.THRESH_BINARY)
output_fixed = cv2.cvtColor(output, cv2.COLOR_BGR2RGB)
fig, ax = plt.subplots()
ax.imshow(output_fixed)
ax.set_title("Binary Segmentation (Fixed Threshold)")
plt.show()
```

(a) Fixed threshold method

```
img_rice_gray = cv2.cvtColor(img_rice, cv2.COLOR_BGR2GRAY)
output_adaptive = cv2.adaptiveThreshold(img_rice_gray, 255.0, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 51, -20.0)
output_adaptive_RGB = cv2.cvtColor(output_adaptive, cv2.COLOR_BGR2RGB)
fig, ax = plt.subplots()
ax.imshow(output_adaptive_RGB)
ax.set_title("Binary Segmentation (Local adptive threshold)")
plt.show()
```

(b) Local adaptive threshold method

Figure 7: Python implementations of threshold methods

(a) After fixed thesholding

(b) After local adaptive thresholding

Figure 8: Result images after thresholding

Problem 2 Count the rice grains in the rice image (given). Show the components (after connected-component analysis) using a color map.

This task highlights the importance of segmentation where important distinct values are extracted such that each value represents a meaningful object. In here, rice.png (rice image) introduces two meaningful regions: the group of pixels belong to rice grains and the group of pixels belongs to the background. Therefore, a suitable approach for the above task is to attempt via binary segmentation in which the input gray scale is transformed into pure black and white image such that white regions belongs to rice gains and the black region to the background.

Binary segmentation could be executed via OpenCV in-built *threshold* function where the threshold value (here, 127), threshold type (here, THRESH_BINARY since this is a binary segmentation task) and the maximum value to be used with thresholding type (here, 255) to be inserted as the input arguments. Howsover, this fixed threshold value method does not facilitate the optimum segmentation since some of the rice gains are also marked as black in the output image.

Therefore, there is an essential need for optimizing the value of threshold in different areas of the image as per the distribution of pixel shades within a limited rectangular region surrounding the pixel. Local adaptive threshold (with adaptive type: ADAPTIVE_THRESH_MEAN_C and threshold type: THRESH_BINARY and block size: 51) is a suitable candidate for this task since it evaluates a statistically suitable threshold for each pixel.

Even after local adaptive thresholding, there exists some outlying specks that need to be removed. For this, *erode* is used with a rectangular operator (5,5). Finally, connected-component analysis is executed using *for* loops and the results show that there exists **102** rice grains in the image.

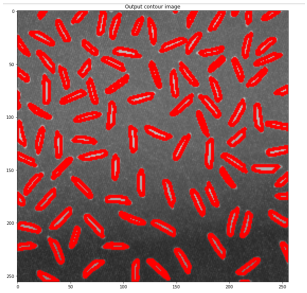
Eventually, a contour detection logic (using *findContours* with contour retrieval mode: RETR_EXTERNAL and contour approximation method: CHAIN_APPROX_SIMPLE) is executed in order to identify the foregroundbackground boarders to encode the external contour shape until the borders of all foreground objects are covered. This ensures that the number of unique external contours detected is same as the number of grains of rice.

```
label_image = output_eroded.copy()
count = 0
rows, cols = label_image.shape
for j in range(rows):
    for i in range(cols):
        pixel = label_image[j, i]
        if 255 == pixel:
            count += 1
            cv2.floodFill(label_image, None, (i, j), count)
print("Number of foreground objects", count)
```

```
Number of foreground objects 102
```

Figure 9: Results after connected-component analysis

(a) After morphological erosion



(b) Output contour image

Figure 10: Result images after each stage

Problem 3 Write a program to zoom images by a given factor in (0,10]. You must use a function to zoom the image, which can handle (by using the included four images, two large originals, and there zoomed-out versions. Test you algorithm by computing the sum of squared difference (SSD) when you scale-up the given small images by a factor of 4 by comparing with the original images):

(a) nearest-neighbor
This is one of the simplest techniques to resample the pixel values present in the input image. This non-adaptive interpolation method utilizes a determination of the "nearest" neighbouring pixel and an assumption of its intensity value.

Since the task needs to zoom the input image upto 10 (greater than 0), the zooming scale is defined to be (0,10]. Afterwards, the out image size is determined and then each pixel in the output image is "assumed" through the values of the input pixels. The obtained results were compared using the given corresponding largescale image by using sum of squared difference (SSD).

```
scale = float(input("Enter scale between 0 and 10: "))
if(scale<=0 or scale>10):
  print("Invalid input scale")
else:
  rows = int(scale*img_forzoom.shape[0])
  cols = int(scale*img_forzoom.shape[1])

  img_zoomed_nearest=np.zeros((rows, cols), dtype = img_forzoom.dtype)

  for i in range(0,rows):
    for j in range(0,cols):
      img_zoomed_nearest[i,j]=img_forzoom[int(i/scale),int(j/scale)]

  img_zoomed_nearest_RGB = cv2.cvtColor(img_zoomed_nearest, cv2.COLOR_BGR2RGB)
  fig, ax = plt.subplots()
  ax.imshow(img_zoomed_nearest_RGB)
  ax.set_title("Zoomed Image (nearest neighbour)")
  plt.show()
```

(a) Nearest-neighbour interpolation using for loops

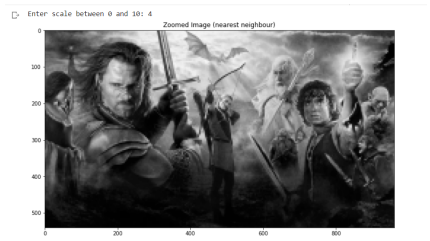
Calculate sum of squared difference

```
[ ] ssd1 = np.sum((img_large[:,:]-img_zoomed_nearest[:,:])**2)
    print("Sum of squared difference for im01.png :", ssd1)

    Sum of squared difference for im01.png : 21937586
```

(b) SSD value for im01.png and zoomed version of im01small.png

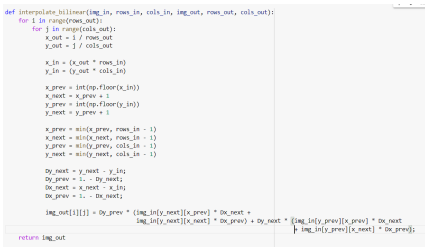
Figure 11: Nearest-neighbour interpolation and SSD



(a) Zoomed im01small.png by scale 4

(b) Zoomed im02small.png and corresponding SSD value

Figure 12: Results from nearest-neighbour interpolation



```
def interpolate_bilinear(img_in, rows_in, cols_in, img_out, rows_out, cols_out):
    for i in range(rows_out):
        for j in range(cols_out):
            x_out = i / rows_out
            y_out = j / cols_out

            x_in = (x_out * rows_in)
            y_in = (y_out * cols_in)

            x_prev = int(np.floor(x_in))
            x_next = x_prev + 1
            y_prev = int(np.floor(y_in))
            y_next = y_prev + 1

            x_prev = min(x_prev, rows_in - 1)
            x_next = min(x_next, rows_in - 1)
            y_prev = min(y_prev, cols_in - 1)
            y_next = min(y_next, cols_in - 1)

            Dy_next = y_next - y_in;
            Dy_prev = 1. - Dy_next;
            Dx_next = x_next - x_in;
            Dx_prev = 1. - Dx_next;

            img_out[i][j] = Dy_prev * (img_in[y_next][x_prev] * Dx_next +
                                       img_in[y_next][x_next] * Dx_prev) + Dy_next * (img_in[y_prev][x_prev] * Dx_next
                                                                                      + img_in[y_prev][x_next] * Dx_prev);
    return img_out
```

(a) Foor loop implementation

```
scale = float(input("Enter scale between 0 and 10: "))
if(scale<=0 or scale>10):
  print("Invalid input scale")
else:
  img_zoomed_bilinear = cv2.resize(img_forzoom,None, fx=scale, fy=scale, interpolation = cv2.INTER_LINEAR)

  img_zoomed_bilinear_RGB = cv2.cvtColor(img_zoomed_bilinear, cv2.COLOR_BGR2RGB)
  fig, ax = plt.subplots()
  ax.imshow(img_zoomed_bilinear_RGB)
  ax.set_title("Zoomed Image (Bilinear Interpolation)")
  plt.show()
  ssd = np.sum((img_large[:,:]-img_zoomed_bilinear[:,:])**2)
  print("Sum of squared difference for im01.png :", ssd)
```

(b) OpenCV inbuilt resize implementation

Figure 13: Different implementation modes for bilinear interpolation

(b) bilinear interpolation

This is an intuitive algorithm for image resizing which generalizes the linear interpolation. This technique is primarily utilized in many 2-D array resizing occasions and is considered to be performed in various manners such as within for loops and using vectorizations (Ex. numpy implementation).

As in the nearest-neighbor method, the scale is to be between 0 and 10 and in this method, the pixels of the output image is determined using four nearest pixels of the input corresponding image. The bilinear interpolation could be implemented using for loops which is relatively slow as compared with the other implementation techniques. Here, both for loop implementation and the inbuilt *resize* method have been deployed to interpolate. The deviations from the corresponding large scale image were calculated using SSD same as the earlier utilization.

(a) Result zoomed image for im01small.png with SSD

(b) Result zoomed image for im02small.png with SSD

Figure 14: Result zoomed images through bilinear interpolation and their corresponding SSD values