Data Visualization HW2

(Cheng Zhong 16307110259)

1 Restate the Basic Global Thresholding (BGT) algorithm so that it uses the histogram of an image instead of the image itself. (Hint: Please refer to the statement of OSTU algorithm)

1. Select an initial threshold T_0 (e.g. the mean intensity)
2. Find the smallest and biggest pixel value a,b
3. Denote the Frequency and Mean value in class [a,b]:
   Frequency: $\omega_{ab} = \sum_{i=a}^{b} P(i); P(i) = \frac{n_i}{N}$
   Mean: $\mu_{ab} = \sum_{i=a}^{b} iP(i)/\omega_{ab}$
4. Calculate the mean intensity values μ_{aT_0} and $\mu_{T_0 b}$
5. Select a new threshold $T_i = (\mu_{aT_0} + \mu_{T_0 b})/2$
6. Repeat steps 3-5 until $T_i = T_{i-1}$

3 Design an algorithm with the function of locally adaptive thresholding (e.g. based on moving average or local OSTU); implement the algorithm and test it on exemplar image(s).

```python
def OTSU(img):
    '''
    img: the image need to be processed by OTSU
    output: a numpy array with 0,1 , shows the value of new picture
    '''
    # find the shape of img
    width, height = img.shape
    # find the maximun and minimun pixel value of image
    max_value = img.max()
    min_value = img.min()
    # store the best threshold value
    max_threshold = 0
    max_loca = 0
    # Total number of pixels
    total = width * height

    for i in range(min_value, max_value):
        small = []
        big = []
        for j in range(width):
            for k in range(height):
                if img[j,k] <= i:
                    # Divide the image into 2 classes
                    small.append(img[j,k])
                else:
                    big.append(img[j,k])
        # Frqency
```

```
        w_s = len(small)/total
        w_b = len(big)/total
        # Mean of each classes
        mean_s = sum(small) / (total*w_s)
        mean_b = sum(big) / (total*w_b)
        # Calculate the between class variance
        sigma_bet = w_s * w_b * (mean_s - mean_b) ** 2
        if sigma_bet > max_loca:
            max_loca = sigma_bet
            max_threshold = i
    # Plot the binary image
    new_img = np.zeros(img.shape)
    for i in range(width):
        for j in range(height):
            if img[i,j] > max_threshold:
                new_img[i,j] = 1
    return new_img


def Q3():
    print('Question 3 is processing.')
    img = RGB2Gray("Q3picture1.png")
    # Find the shape of the image
    width, height = img.shape
    # Create the new image
    threshold_img = np.zeros(img.shape)
    # Create the local subimage and using OTSU to calculate the Binarization
        Image
    for i in range(0,width,50):
        for j in range(0,height,50):
            # Update
            threshold_img[i:i+50, j:j+50] = OTSU(img[i:i+50, j:j+50])
        time.sleep(0.1)
        end_str = '100%'
        process_bar(i/width, start_str='', end_str=end_str, total_length=15)
    # Show the Binarization Image by the color of black and white
    threshold_img *= 255
    im = Image.fromarray(threshold_img)
    im = im.convert('L')
    im.show()
    im.save('Q3.png')
```

Here is the image before and after local interpolation. We can find that the main features of the image are well retained by OTSU

(a) Before local OTSU (b) After local OTSU

Figure 1: The image before and after local OTSU

4 Implement a linear interpolation algorithm and apply: Read an image, use linear interpolation algorithm to enlarge the picture spatial resolution by N times, and then save the picture.

```
def Interpolation(new_img, img, n):
    '''
    new_img: array to store the new image, shape = (img.shape * n)
    img: the image need to interpolation
    n : How many times the sharpness needs to rise
    '''
    width, height = new_img.shape
    width_old, height_old = img.shape
    for i in range(width):
        for j in range(height):
            i_old = i / n
            j_old = j / n
            x1 = int(i_old)
            y1 = int(j_old)
            x2 = int(min(i_old + 1, width_old-1))
            y2 = int(min(j_old + 1, height_old-1))
            # Locate the four points that needed in interpolation
            p1, p2, p3, p4 = img[x1,y1], img[x1,y2], img[x2,y1], img[x2,y2]
            # the distances in interpolation
            u = i_old - x1
            v = j_old - y1
            # the new value
            value = p1 * (1-u) * (1-v) + p2 * (1-u) * v + p3 * u * (1-v) + p4 * u
                * v
            new_img[i,j] = value
        time.sleep(0.1)
        end_str = '100%'
        process_bar(i/width, start_str='', end_str=end_str, total_length=15)
    return new_img

def Q4(n):
    print('Question 4 is processing.')
```

```
img = RGB2Gray("Q4.png")
width, height = img.shape
new_img = np.zeros((width*n, height*n))
new_img = Interpolation(new_img, img, n)
# Show the new image
im = Image.fromarray(new_img)
im = im.convert('L')
im.show()
im.save('Q4.png')
```

(a) Before Interpolation

(b) After Interpolation with N=2

Figure 2: The image before and after interpolation

Here I chose a grey picture of Zibin building as an example. It can be seen that even after the image is doubled, the sharpness has not been significantly reduced after linear interpolation. You can see more details about the picture from the submitted file.