王　兆東

マルチメディア信号解析

2020年5月5日

Report of Second Assignment

In this report I will discuss how I design my program to complete the assigned task, it include how to read an image by program and several image transferring operation like histogram , rotation and margin detection.

1. Environment Introduction

· Programming language: Python3.6

· library used: OpenCV

· text book：jupyter

2. Implementation

The procedure of decoding were used like before, however for the gray image the level of the pixel has changed from RGB array into gray level (0-255).

As we calculate every gray level’s ratio as Y axis, the gray level as X axis, the histogram has been shown as below.

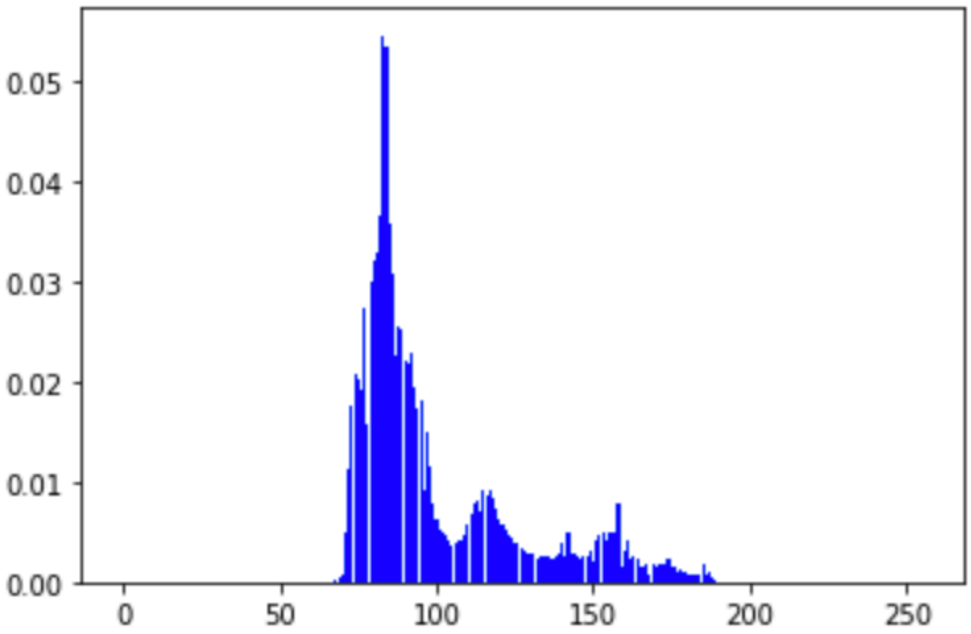


Figure 1: the original histogram of image

Figure 2: original image of a Marshall amp

As we calculate the range of gray level of X axis, we can use a certain percentage to change this level so we can realize the equalization.

```
for i in range(0, 255):
    if count[i] != 0:
        min = i
        print('the minist number is ',min)
        break
```

```
the minist number is  59
```

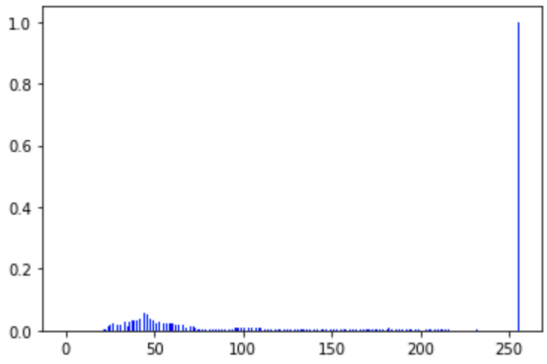
```
for i in range(0,255):
    if count[255-i] != 0:
        max = 255-i
        print('the max number is ', max)
        break
```

```
the max number is  197
```

```
dst_img = np.zeros((height,width), np.uint8) #0-255
for i in range(0, height):
    for j in range(0, width):
        dst_img[i,j] = ((img[i,j]-min)/(max-min))*255
```

Picture3: calculate the gray level range and equalization

After the equalization the range of gray level has Been expanded and the contrast of image has promoted, the histogram of destining image and the image was shown as below.



Picture4: histogram after equalization

Picture5: picture after equalization

The next way of equalization is kind of complex, we need first find out a CDF(累積分布関数) function and use that mapping to reconstruct every pixel. For the details, assume the first gray level's ratio to be the A, and so the next gray level B would sum the first one to be the A+B finally, and so on the third one C would become A+B+C, that is to say the later one would be the sum of previous gray level's ratio, and in the last the ratio would become 1, the code is as below.

```
count2 = np.zeros(256,float)
```

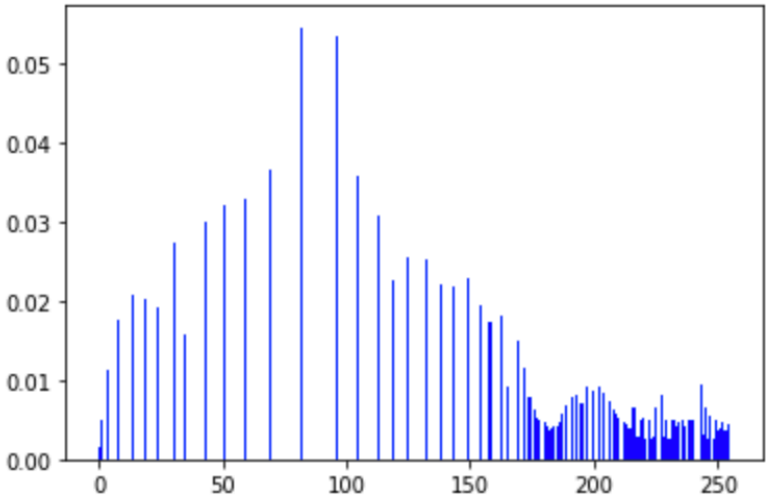
```
for i in range(0, height):
    for j in range(0, width):
        pixel = img[i,j]
        index = int(pixel)
        count2[index] = count2[index]+1
```

```
for i in range(0, 255):
    count2[i] = count2[i]/(height*width)
```

```
sum1 = float(0)
for i in range(0, 255):
    sum1 = count2[i]+sum1
    count2[i] = sum1
```

Picture6: CDF function

After calculated the ratio we multiply the 255, so the new_array ’s index and value become the mapping function. We use every pixel to complete the mapping and the final results are as belows.



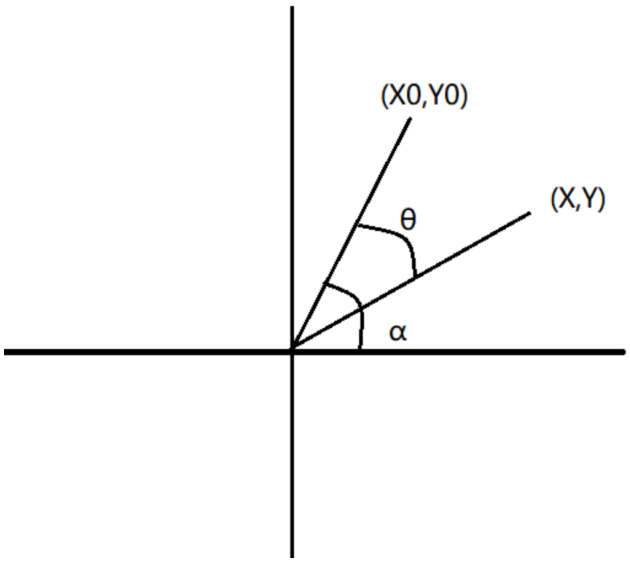
Picture7: histogram after equalization

Picture 8: picture after equalization

Conclusion: the two ways can greatly promote the contrast of image, for this image the first way makes the image more gentle and the second way makes the image more sharp.

Rotation

for the target we want to rotate (X0, Y0) into (X,Y), like the image below:



Picture 9: rotation in axis

We can calculate the two matrix transforming between source and destination:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x_0 & y_0 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation matrix in the previous part is derived from the math coordinate system, and the image coordinate system is the image coordinate system with the upper left corner as the origin. We need to convert the image coordinate system to the math coordinate system. Assuming that the original picture size is W, H, the minimum rectangle size of the picture after rotation is W ′, H′ . So the formula of rotation should be as follows.[1]

$$\begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -0.5W & 0.5H & 0 \end{bmatrix} * \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0.5W' & 0.5H' & 1 \end{bmatrix}$$

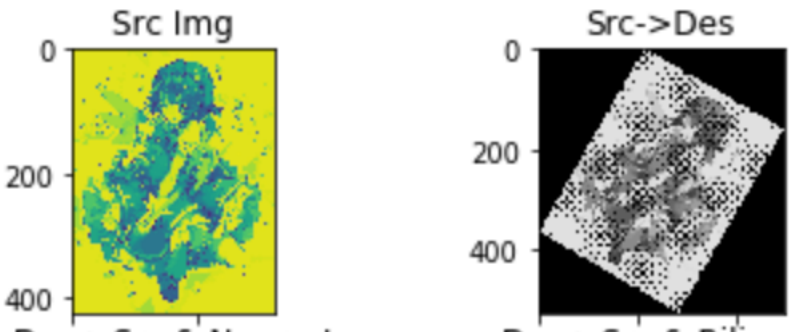
$$\begin{bmatrix} x_0 & y_0 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -0.5W' & 0.5H' & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0.5W & 0.5H & 0 \end{bmatrix}$$

```
#scr -> des
trans1 = np.array([[1,0,0],[0,-1,0],[-0.5*w,0.5*h,1]])
trans1 = trans1.dot(np.array([[np.cos(angle),-np.sin(angle),0],[np.sin(angle),np.cos(angle),0],[0,0,1]]))
trans1 = trans1.dot(np.array([[1,0,0],[0,-1,0],[0.5*newW,0.5*newH,1]]))
```

```
#des -> src
trans2 = np.array([[1,0,0],[0,-1,0],[-0.5*newW,0.5*newH,1]])
trans2 = trans2.dot(np.array([[np.cos(angle),np.sin(angle),0],[-np.sin(angle),np.cos(angle),0],[0,0,1]]))
trans2 = trans2.dot(np.array([[1,0,0],[0,-1,0],[0.5*w,0.5*h,1]]))
```

Picture 10: rotation matrix and code

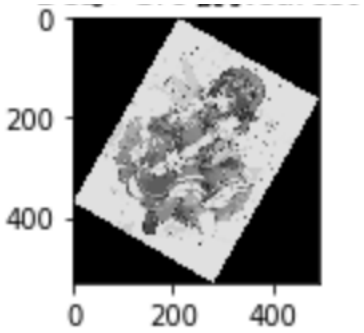
The normal way of forward mapping would generate some loss, like some snowblowers, the reason is because the rotated coordinates obtained in the forward mapping are floating-point numbers, but the pixels can only be integers, therefore the missing pixel.



Picture 11: rotation picture and noise

The back-forward mapping method could improve the quality.

```
for x in range(newW):
    for y in range(newH):
        srcPos = np.array([x,y,1]).dot(trans2)
        if srcPos[0] >= 0 and srcPos[0] < w and srcPos[1] >= 0 and srcPos[1] < h:
            newimg2[y][x] = img[int(srcPos[1])][int(srcPos[0])]
```



Picture 11: rotation which use back-forward mapping

Margin

Margin algorithm we have to compare the two near pixel and calculate the gray level's distance, then add a large number to be the new pixel.

```
for i in range(0, height):
    for j in range(0,width-1):
        imgP0 = int(img[i,j])
        imgP1 = int(img[i,j+1])
        newP = imgP0 - imgP1 + 150
        if newP > 255:
            newP = 255
        if newP < 0:
            newP = 0
        dst[i,j] = newP
```

Picture 12: calculation of margin

The final result looks like a sculpture effect like below:

Picture 13: Picture of margin detection

3. Running method

The program can run with at least the OpenCV and numpy library, make sure the picture and code are in the same folder.

[1] 图像旋转算法原理, https://blog.csdn.net/liyuan02/article/details/6750828