CV701: Computer Vision Assignment 1

Abdul Ahad Butt
abdul.butt@mbzuai.ac.ae

Ilmuz Zaman Mohammed Zumri
mohammed.zumri@mbzuai.ac.ae

Muhammad Abdullah Sohail
mabdullah.sohail@mbzuai.ac.ae

(a) Taken during the day

(b) Taken at night

Figure 1. Images of our chosen object(flag) taken during the day and the night

Settings for 1a	
Shutter Speed	1/500
Aperture	f14
ISO	400
Settings for 1b	
Shutter Speed	1/13
Aperture	f4.5
ISO	6400

Table 1. Settings used for each image

1. Task 1: Exploring Different Camera Settings

The objects we chose are the flags hoisted up near the Abu Dhabi Quality and Conformity Council QCC. The images we took from our camera are show in 1. Their corresponding settings are shown in table 1.

Image 1a was taken at around 11 AM when there was abundant lighting. Therefore, we reduced the ISO to a low value of 400 to ensure that there is minimum digital noise. To capture the flag in motion not distorted by movement, we used a high shutter speed. Since a higher shutter speed allows light to enter the sensor only for a small time frame, we compensated it by having a sufficiently wide aperture. This ensured that the image was well-exposed, while capturing the flag in motion with minimum noise.

Image 1b was taken during the night when the lighting was quite low. To compensate, we increased the ISO to 6400 to boost the sensitivity of the camera's sensor. There was a gust of wind blowing during the time of capture and since we needed to capture the flag in motion, we used a higher shutter speed. However, due to the lower light, the image was underexposed. To counter this, we significantly increased the aperture. We experimented with various combinations of high shutter speed and high aperture until we found the optimal settings.

2. Task 2: White-Balance Algorithms and Unsharp Masking

White balance algorithms are strategies utilized in image processing to rectify color casts and guarantee that the colors in an picture show up as they would beneath impartial lighting conditions. The objective of these calculations is to form objects that are white in reality moreover show up white within the picture, in any case of the lighting conditions. There are two fundamental categories of white adjust calculations: white-patch and grey-world.

2.1. Grey World Algorithm

The Grey World Algorithm is works on the belief that the mean worth of the scene came about at an optimal lighting level is neutral gray. This algorithm is able to correct colour casts if the colour channels are manipulated in such a way

as to make the average value of a given image appear as a neutral gray.

2.1.1 Methodology

Let the image be represented in a 3D matrix format as $I(x, y, c)$ where x and y are the pixel coordinates of the image and c represents the 'Color Channels' that are Red, Green, Blue respectively. As with most post-processing tools, the goal of the Gray World Algorithm is simple and simple – to try to balance the color scheme of the image by assuming that the average color within the scene should be some shade of gray.

1. **Calculating Mean** Compute the mean intensity value of all pixels in the image for each color channel $c \in \{R, G, B\}$:

$$\mu_c = \frac{1}{N} \sum_{x=1}^{W} \sum_{y=1}^{H} I(x, y, c)$$

where W and H are the width and height of the image, and $N = W \times H$ is the total number of pixels.

2. **Find Global Mean a.k.a. Gray Level:** Global mean μ_{gray} is the average of the means of Red, Green, and Blue channels:

$$\mu_{gray} = \frac{1}{3}(\mu_R + \mu_G + \mu_B)$$

3. **Scale Color Channels:** Scale each color channel $c \in \{R, G, B\}$ by a factor determined by the ratio of the global mean to the channel mean:

$$I'(x, y, c) = I(x, y, c) \times \frac{\mu_{gray}}{\mu_c}$$

This adjusts the average color intensity for each channel to correspond the global gray level.

4. **Clipping Pixel Values:** Clip the adjusted pixel values to ensure they remain within the valid range $[0, 255]$:

$$I'(x, y, c) = \min(255, \max(0, I'(x, y, c)))$$

2.1.2 Code

```
def grey_world_algorithm(image):
    # Compute the average of each channel (B, G,
    ↪  R)
    avg_b = np.mean(image[:, :, 0])
    avg_g = np.mean(image[:, :, 1])
    avg_r = np.mean(image[:, :, 2])

    # Compute the overall average
    avg_all = (avg_b + avg_g + avg_r) / 3

    # Scale each channel to match the overall
    ↪  average
    scale_b = avg_all / avg_b
    scale_g = avg_all / avg_g
    scale_r = avg_all / avg_r

    # Apply scaling
    balanced = np.zeros_like(image,
    ↪  dtype=np.float32)
    balanced[:, :, 0] = image[:, :, 0] * scale_b
    balanced[:, :, 1] = image[:, :, 1] * scale_g
    balanced[:, :, 2] = image[:, :, 2] * scale_r

    # Clip and convert back to uint8
    return np.clip(balanced, 0,
    ↪  255).astype(np.uint8)
```

2.2. White Patch Algorithm

White patch algorithm is one of the color correctors applied to an image in order to balance the white in an image. It assumes that the 'brightest area in the picture as white and recalculates and balances all the shades to bring out what it identifies as 'white'.

2.2.1 Methodology

The image can therefore be represented as a three dimensional matrix $I(x, y, c)$ where x and y represent the pixel locations while c is the color channels being R, G, B. The White Patch Algorithm also presupposes that white is the maximum measurable pixel value in at least one channels, and which it scales the image with.

1. **Calculate Max Pixel:** Compute the maximum pixel intensity value for each color channel $c \in \{R, G, B\}$:

$$M_c = \max_{x,y} I(x, y, c)$$

where M_c is the maximum value in the color channel c.

2. **Channel Scaling:** Use this maximum pixel value as a factor to scale each pixel in the image for its respective color channel:

$$I'(x, y, c) = I(x, y, c) \times \frac{255}{M_c}$$

This step makes the highest intensity pixel in each channel scale up to the maximum value of 255 (pure white).

3. **Clip Pixel Values:** Clip the adjusted pixel values to make them within the valid range of $[0, 255]$:

$$I'(x, y, c) = \min(255, \max(0, I'(x, y, c)))$$

2.2.2 Code

```
def white_patch_algorithm(image,
↪  gamma_correction=True):
    # Convert image to float32 for precision
    img_float = image.astype(np.float32)

    # Find maximum values for each channel (R, G,
    ↪  B)
    max_r = np.max(img_float[:, :, 2])  # Red
    ↪  channel
    max_g = np.max(img_float[:, :, 1])  # Green
    ↪  channel
    max_b = np.max(img_float[:, :, 0])  # Blue
    ↪  channel

    # Scale each channel by its maximum value to
    ↪  achieve white balance
    img_float[:, :, 2] = img_float[:, :, 2] /
    ↪  max_r * 255.0  # Red channel
    img_float[:, :, 1] = img_float[:, :, 1] /
    ↪  max_g * 255.0  # Green channel
    img_float[:, :, 0] = img_float[:, :, 0] /
    ↪  max_b * 255.0  # Blue channel

    # Clip values to stay within the valid range
    ↪  [0, 255]
    img_float = np.clip(img_float, 0, 255)

    # Optionally apply gamma correction for
    ↪  better contrast
    if gamma_correction:
        gamma = 1.7  # Gamma value (you can
        ↪  adjust this value)
        img_float = ((img_float / 255.0) ** (1.0
        ↪  / gamma)) * 255.0

    # Convert back to unsigned 8-bit integer
    ↪  array
    img_balanced = img_float.astype(np.uint8)

    return img_balanced
```

Remark: A gamma correction factor is added to make the results of the white patch algorithm more noticeable.

2.3. Unsharp Masking

Unsharp masking is an enhancement technique, which enhances an image that has been blurred, by subtracting a blurred image from the sharp one. This process brings forward the outline and small features and makes the image to look sharper.

2.3.1 Methodology

1. **Image Blur:** Smooth the using a Gaussian blur with a kernel K and a standard deviation σ. The blurred image I_{blur} is given by:

$$I_{blur} = I * K$$

where $*$ denotes the convolution operation.

Figure 2. Original Image

2. **Generating Mask:** Subtract the blurred image from the original image to obtain the mask:

$$M = I - I_{blur}$$

3. **Mask Amplification:** Multiply the mask M by a scaling factor α to have a restraint on the amount of effect it produces on the image:

$$M_{enhanced} = \alpha \cdot M$$

4. **Mask Integration:** Merge the enhanced mask to the original image to obtain the sharpened image I_{sharp}:

$$I_{sharp} = I + M_{enhanced}$$

```
def unsharp_masking(image, sigma=1.0,
↪  strength=1.5):
    # Convert image to float32 for precision
    image = image.astype(np.float32)

    # Create a blurred version of the image
    blurred = cv2.GaussianBlur(image, (0, 0),
    ↪  sigmaX=sigma, sigmaY=sigma)

    # Calculate the sharpened image using the
    ↪  formula
    sharpened = cv2.addWeighted(image, 1.0 +
    ↪  strength, blurred, -strength, 0)

    # Clip values to valid range and convert back
    ↪  to uint8
    return np.clip(sharpened, 0,
    ↪  255).astype(np.uint8)
```

2.3.2 Results

Figure 2 and 3 show the original image and the output image after applying unsharp masking respectively.

2.4. White Balance Algorithm Results

Figure 4 and 5 show the results of applying grey world and white balance algorithms on the sharpened image (from unsharp masking) respectively.

Figure 3. Unsharp Masked Image

Figure 4. Grey World Filter Output

Figure 5. White Patch Filter Output

In Figure 4 we can notice that the overall color spectrum has become a bit dull shifting towards the gray spectrum. Further, Figure 5 portrays an overall increase in white spectrum of the image giving a brighter look because all the pixel values were scaled up with respect to the brightest pixel.

3. Task 2.2: Histogram Equalization and Contrast Stretching

3.1. Histogram Equalization

Histogram Equalization is an image processing method to increase the global contrast of an image using it's intensity histogram. The goal is to make the intensity distribution of the image's pixels more uniform, which can improve the visibility of of features in the image.

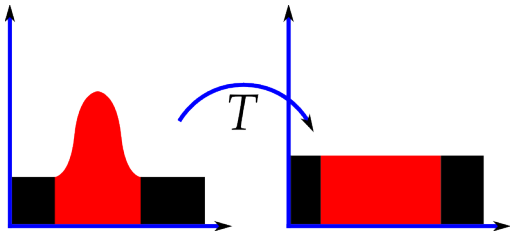


Figure 6. Pixel intensity histograms of a grayscale image before and after histogram equalization. The transform T "evens" out the intensity histogram for the image

Figure 6 shows the effect of histogram equalization when applied to a grayscale image.

However, for the RGB color space, histogram equalization should not be applied separately to each individual channel. This is because the pixel values in an RGB image are not indicative of the intensity (or brightness) of that pixel. Each channel of the R, G, and B represents the intensity of the related color, not the intensity/brightness of the image as a whole.

We have added a section in the appendix showcasing applying histogram equalization to each channel in an RGB image.

Therefore, before applying histogram equalization to our image, we first convert it into YUV Space [3], and then apply Histogram Equalization to the Y (or luma) component [1,2]. We then convert our image back into RGB space.

3.1.1 Methodology

More concretely, let $\mathcal{I} \in \mathbb{R}^{H,W,C}$ be the input image where H, W is the height and width of the image respectively, and C is the number of channels (3 for RGB). Moreover, let the intensity of the pixels range from 0 to $(L-1)$, with L possible intensity values, usually 256.

We first convert our Image from the RGB color space to the Y'UV color space, $\hat{\mathcal{I}}$.

We define p as the cumulative frequency distribution of $\hat{\mathcal{I}}$ as:

$$p_n = \frac{\text{number of pixels with intensity n}}{\text{total number of pixels}}$$
$$n = 0, 1, \ldots 255$$
For Component Y'

For histogram equalization, each pixel value in Y' will

be mapped to it's cumulative density value as follows:

$$\hat{\mathcal{I}}^*_{i,j} = \text{round}\left((L-1)\sum_{n=0}^{\hat{\mathcal{I}}_{i,j}} p_n\right)$$
$$\text{for each pixel in} Y'$$

Then we convert back from Y'UV space to RGB space.

3.1.2 Code

All the following codeblocks are applied to the Y' channel of our Y'UV image.

To plot the luminance values for our Image before and after applying histogram equalization, we use the following code:

```
def
↪ plot_histogram(channel_intensities:np.ndarray,
↪ title:str, save=True):
    hist, bins =
    ↪ np.histogram(channel_intensities.ravel(),
    ↪ bins=256, range=(0,256))
    cdf = hist.cumsum()
    cdf_norm = cdf * float(hist.max()) /
    ↪ cdf.max()


    plt.hist(channel_intensities.ravel(),
    ↪ bins=256, range=(0,255))
    plt.stairs(cdf_norm)
    plt.title(title)
    plt.xlim([0, 256])
    plt.xlabel('pixel intensities')
    plt.ylabel('frequency')
    plt.legend(('cdf','histogram'), loc = 'upper
    ↪ left')
    if save:
        plt.savefig(f'data/{title}.png')

    plt.show()
```

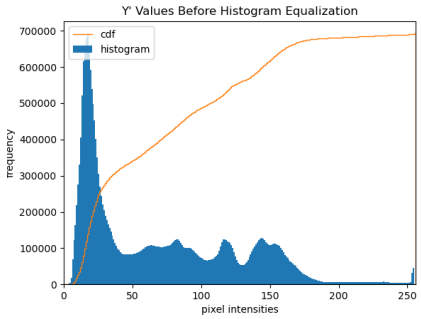
The corresponding Y' Histograms are displayed in figure 7.

We first flatten the image into a 1D array and calculate the cumulative distribution function (cdf) for the image. The code to calculate the cdf is attached below:

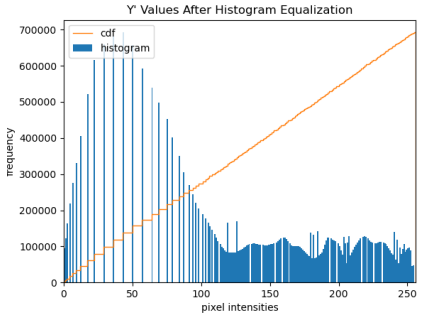
```
def calc_cdf(count):
    cdf = np.cumsum(count) / sum(count)
    return cdf
```

We use the following code to apply histogram equalization to our image $\mathcal{I}$.

```
def
↪ histogram_equalization_channel(image:np.ndarray,
↪ nbins=255):
    hist, bin_centers =
    ↪ np.histogram(image.ravel(), bins=nbins)
    # bin_width = bin_centers[1] - bin_centers[0]
    cdf = calc_cdf(hist)
```



(a) luma values before histogram equalization



(b) luma values after histogram equalization

Figure 7. Y' Channel Intensities before and after applying histogram equalization

```
    flattened_img = image.ravel()
    out = []
    for pix in flattened_img:
        out.append(cdf[pix-1])

    out = (np.array(out).reshape(image.shape) *
    ↪ 255).astype('uint8')
    return out
```

Finally, we convert our image back to RGB space.

```
def histogram_equalization(img:np.ndarray):
    '''
    Image in RGB Space
    '''
    # ? Converting from BGR color space to YUV
    ↪ color space
    img_yuv = cv2.cvtColor(img,
    ↪ cv2.COLOR_BGR2YUV)
    # ? Applying histogram equalization to Y'
    ↪ channel (see codeblock above)
    img_yuv[:,:,0] =
    ↪ histogram_equalization_channel(img_yuv[:,:,0])
    # ? Converting from YUV back to BGR
    out = cv2.cvtColor(img_yuv,
    ↪ cv2.COLOR_YUV2BGR)
    return out
```

The Final Image is shown at 8. As we can see the final image has a much more even contrast than the original

Figure 8. Image after histogram equalization applied

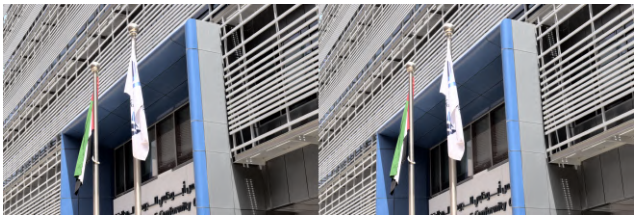


Figure 9. Comparison with OpenCV: Ours is on the left

Mean Squared Error	PSNR
15.15	36.33 db

Table 2. Distance metrics between the output image of our implementation and OpenCV's

image.

We have also applied the same algorithm to the image taken during the night and added it in the appendix. You can check out the night image's output at 17

3.1.3 Comparison with OpenCV

We compare our method with the library function provided by OpenCV [5]. We use the following wrapper function to apply cv2's histogram equalization function to our RGB image

```
def equalize_histogram_cv(img:np.ndarray):
    img_yuv = cv2.cvtColor(img,
    ↪  cv2.COLOR_BGR2YUV)
    img_yuv[:,:,0] =
    ↪  cv2.equalizeHist(img_yuv[:,:,0])
    out = cv2.cvtColor(img_yuv,
    ↪  cv2.COLOR_YUV2BGR)
    return out
```

The output images for our algorithm and OpenCV's implementation are attached at 9.

To calculate the difference between the output from our method and the output from OpenCV's implementation, we calculated distance metrics and show them in table 2.

3.2. Contrast stretching

Contrast stretching adjusts pixel intensities, making darker regions appear darker and lighter regions appear brighter. This increases the perceptual differences between intensity levels leading to a clearer, more visually distinct representation. A linear step-wise stretching function was used here (see Figure 10).

Let L be the maximum intensity level (255 for an 8-bit image). The linear piece-wise contrast stretching function [4] can be expressed as:

$$I_{\text{out}}(x) = \begin{cases} \frac{b_1}{a_1} \cdot x & \text{for } 0 \leq x \leq a_1 \\ \frac{b_2 - b_1}{a_2 - a_1} \cdot (x - a_1) + b_1 & \text{for } a_1 < x \leq a_2 \\ \frac{L-1-b_2}{L-1-a_2} \cdot (x - a_2) + b_2 & \text{for } a_2 < x \leq L-1 \end{cases}$$

where x represents the input pixel value and $I_{\text{out}}(x)$ represents the output pixel value.

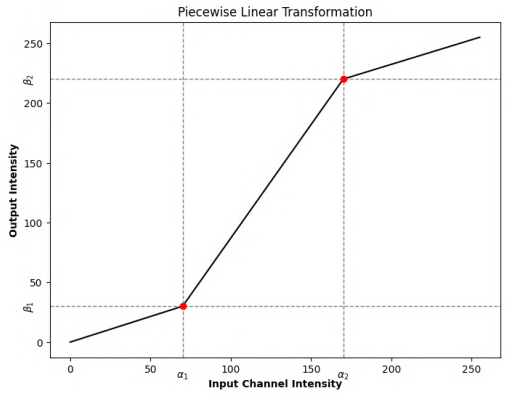


Figure 10. Graph of the piece-wise contrast stretching function

3.2.1 Methodology

Contrast stretching is applied separately to each of the three RGB channels. The process involves splitting the image into its individual channels, applying the contrast stretching function to each channel, and then merging the channels back together.

The RGB image is split into its individual channels using the following code:

```
channels = cv2.split(rgb_img)
```

The code for the contrast stretching function is provided below.

```
def contrast_stretch(channel, a1, a2, b1, b2,
↪  L=256):
    output_image = np.zeros(channel.shape,
    ↪  dtype=np.uint8)

    #0 to alpha1
```

```
    output_image[channel <= a1] = (b1 / a1) *
    ↪  channel[channel <= a1]

    #alpha1 to alpha2
    mask = (channel > a1) & (channel <= a2)
    output_image[mask] = ((b2 - b1) / (a2 - a1))
    ↪  * (channel[mask] - a1) + b1

    #alpha2 to L-1
    output_image[channel > a2] = ((L-1 - b2) /
    ↪  (L-1 - a2)) * (channel[channel > a2] -
    ↪  a2) + b2

    return output_image
```

The function is applied to the three channels separately.

```
cont_stretched_channels =
↪  [contrast_stretch(channel, a1, a2, b1, b2)
↪  for channel in channels]
```

The contrast stretched channels are then merged together to get the final stretched image.

```
stretched_img_rgb =
↪  cv2.merge(cont_stretched_channels)
```

3.2.2 Impact of different alpha and beta values

The table 3 shows the different alpha and beta value pairs that we experimented on.

Set	α_1	α_2	β_1	β_2
1	30	180	10	220
2	50	180	10	220
3	30	150	10	220
4	30	180	20	200

Table 3. Alpha and Beta values for contrast stretching.

Figures (11-14) show the results for different alpha and beta values.

Figure 11. Contrast stretched Image with $\alpha_1 = 30$, $\alpha_2 = 180$, $\beta_1 = 10$, and $\beta_2 = 220$.

Figure 12. Contrast stretched Image with $\alpha_1 = 50$, $\alpha_2 = 180$, $\beta_1 = 10$, and $\beta_2 = 220$.

Figure 13. Contrast stretched Image with $\alpha_1 = 30$, $\alpha_2 = 150$, $\beta_1 = 10$, and $\beta_2 = 220$.

Figure 14. Contrast stretched Image with $\alpha_1 = 30$, $\alpha_2 = 180$, $\beta_1 = 20$, and $\beta_2 = 200$.

The alpha and beta values used in Figure 11 resulted in an image with good color contrast without distortion. Figure 12 demonstrates that increasing α_1 causes darker regions to become significantly darker, particularly noticeable in the flag's background, due to a larger range of pixels being pulled towards the darker end. Figure 13 illustrates the effect of lowering α_2, which enhances brightness in lighter regions, as more pixels are shifted towards the lighter end.

In Figure 14, an increase in β_1 and a decrease in β_2 reduce contrast, leading to less pronounced dark and light areas compared to the other images.

We also performed contrast stretching by converting the RGB image to Y'UV color space and stretching the Y' channel. This enhances the luminance without affecting the colour information. The result are shown in Figure 20 in the appendix.

3.2.3 Histogram visualization of contrast stretching

In order to better visualise what the linear step-wise contrast stretching function does to the pixel values, we plotted the histograms for the three channels of the original and contrast stretched images.

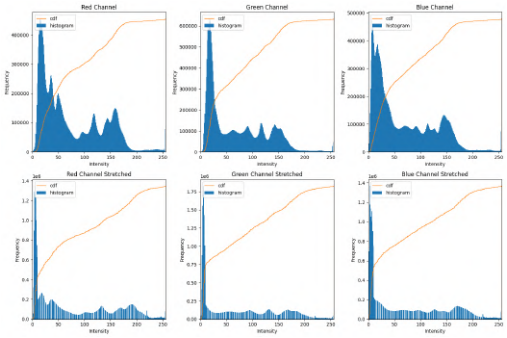


Figure 15. Histograms of the original and contrast stretched images

We can clearly observe that the pixel frequency in the darker and lighter range have increased significantly when compared to the original image.

3.3. Comparison between Histogram equalization and contrast stretching results

(a) Contrast stretched Image with $\alpha_1 = 30$, $\alpha_2 = 180$, $\beta_1 = 10$, and $\beta_2 = 220$.

(b) Histogram output image.

Figure 16. Side-by-side images of contrast stretching and histogram output.

In the histogram-equalized image, features such as the window behind the flag and the object in the bottom right are more discernible. In contrast, these regions have become darker and less visible in the contrast-stretched image. This occurs because the contrast-stretching algorithm, applied to an already dark area, further reduces its brightness, which may lead to a loss of information. The contrast-stretched image is more visually appealing due to the enhanced contrast between darker and lighter regions. The flag appears more pronounced and highlighted, adding to the overall visual impact.

Histogram Equalization enhances in areas with lower contrast but may sometimes result in a lost of details in very bright or dark areas. Contrast Stretching allows control over which region of the intensity range are stretched, giving more targeted control over the contrast.

Advantages of Histogram Equalization

- No parameter tuning required
- Effective in images with narrow range of intensity values

Disadvantages of Histogram Equalization

- Non linear changes may lead to over-enhancement or loss of detail in some region
- Not suitable for images where the contrast needs to be adjusted in specific regions

Advantages of Contrast Stretching

- Allows user control over how the contrast is enhanced in different parts of the image
- Can preserve important details by limiting stretching to specific intensity ranges

Disadvantages of Contrast Stretching

- Required defining breakpoints and slopes, so it's less automatic
- May not work as well as histogram equalization for images where the overall contrast needs to be enhanced uniformly

4. Appendix

4.1. Applying Histogram Equalization to each channel in RGB Image

We applied histogram equalization to each channel of our day rgb image and have shown the results at 21.

Using this approach we get red artifacts behind the UAE flag. We believe this is a side effect of applying histogram equalization to the red channel. This example showcases why we should not apply histogram equalization to individual rgb channels and should only apply them to intensity channels in Y'UV or HSI color spaces.

The code for this can be found attached below

Figure 17. Histogram Equalization applied to 1b

Figure 18. Contrast stretching applied to 1b

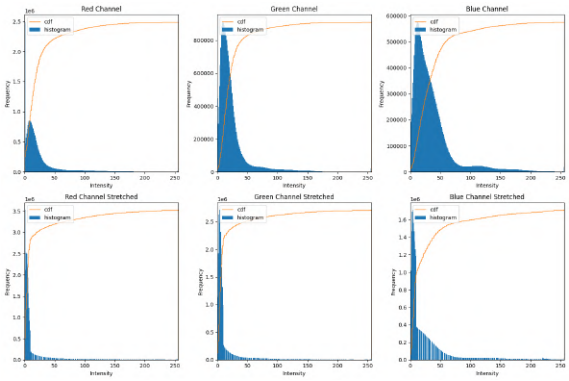


Figure 19. Histograms of the three channels of the original image 1b and after contrast stretching

```
def histogram_equalization_rgb(img:np.ndarray):
    img[:,:,0] =
    ↪  histogram_equalization_channel(img[:,:,0])
    img[:,:,1] =
    ↪  histogram_equalization_channel(img[:,:,1])
    img[:,:,2] =
    ↪  histogram_equalization_channel(img[:,:,2])

    return img
```

Figure 20. Contrast stretching applied to 1a by converting to Y'UV color space with $\alpha_1 = 30$, $\alpha_2 = 180$, $\beta_1 = 10$, and $\beta_2 = 220$.

Figure 21. Histogram Equalization applied to individual RGB channels

References

[1] Stackoverflow: Opencv python equalizehist colored image. https://stackoverflow.com/questions/31998428/opencv-python-equalizehist-colored-image. 4

[2] Tutorial: Histogram equalization of a color image with opencv. https://www.opencv-srf.com/2018/02/histogram-equalization.html. 4

[3] Yuv color space. https://en.wikipedia.org/wiki/Y%E2%80%B2UV. 4

[4] Tutorial: Contrast stretching, 2019. https://theailearner.com/2019/01/30/contrast-stretching/. 6

[5] Ana Huamán. Histogram equalization. https://docs.opencv.org/4.x/d4/d1b/tutorial_histogram_equalization.html. 6