Part 1

**Q1**

**resize()**

This function resizes an image to a specified size.

| **#** | **Parameters** | **Type** | **Explanation** |
| --- | --- | --- | --- |
|  | image | ndarray | Input image |
|  | output\_shape | size | Size of the generated output image - rows, cols, dim |
|  | order | int, optional | The order of the interpolation [0-5]. The default will be 0 if the image dtype is bool and 1 otherwise. |
|  | mode | {‘\*\*\*’}, optional | Fills in points outside the input's boundaries |
|  | cval | float, optional | Used id the mode is ‘constant’ and it is the value outside the image boundaries. |
|  | clip | bool, optional | Whether to clip the output to the range of values of the input image. The default is true because higher order interpolation may produce values outside the given input range |
|  | preserve\_range | float, optional | Whether to keep the original range of values. Otherwise, the image will be converted to float. |
|  | anti\_aliasing | bool, optional | Whether to apply a Gaussian filter to smooth the image prior to downsampling. It is crucial to filter when downsampling the image to avoid aliasing artifacts. If not specified, it is set to True when downsampling an image whose data type is not bool. |
|  | anti\_aliasing\_sigma | float \ tuple of floats, optional | Standard deviation for Gaussian filtering used when anti-aliasing. By default, this value is chosen as (s - 1) / 2 where s is the downsampling factor, where s > 1. For the up-size case, s < 1, no anti-aliasing is performed prior to rescaling. |

**rescale()**

Rescaling an image according to a certain factor.

This function gets an image and wanted scale and returns the scale version of the input image.

| **#** | **Parameters** | **Type** | **Explanation** |
| --- | --- | --- | --- |
|  | image | ndarray | Input image |
|  | scale | float \ tuple of floats | Scale factor |
|  | order | int, optional | The order of the interpolation [0-5]. The default will be 0 if the image dtype is bool and 1 otherwise. |
|  | mode | {‘\*\*\*’}, optional | Fills in points outside the input's boundaries |
|  | cval | float, optional | Used id the mode is ‘constant’ and it is the value outside the image boundaries. |
|  | clip | bool, optional | Whether to clip the output to the range of values of the input image. The default is true because higher order interpolation may produce values outside the given input range |
|  | preserve\_range | float, optional | Whether to keep the original range of values. Otherwise, the image will be converted to float. |
|  | anti\_aliasing | bool, optional | Whether to apply a Gaussian filter to smooth the image prior to downsampling. It is crucial to filter when downsampling the image to avoid aliasing artifacts. If not specified, it is set to True when downsampling an image whose data type is not bool. |
|  | anti\_aliasing\_sigma | float \ tuple of floats, optional | Standard deviation for Gaussian filtering used when anti-aliasing. By default, this value is chosen as (s - 1) / 2 where s is the downsampling factor, where s > 1. For the up-size case, s < 1, no anti-aliasing is performed prior to rescaling. |
|  | channel\_axis | int \ none, optional | If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels. |

**rotate()**

Rotate an input image by a certain angle around its center.

| **#** | **Parameters** | **Type** | **Explanation** |
| --- | --- | --- | --- |
|  | image | ndarray | Input image |
|  | angle | float | Clockwise direction rotation angle degrees |
|  | resize | bool, optional | Determine whether the shape of the output image will be automatically calculated, so the complete rotated image exactly fits. **The default is False**. |
|  | center | iterable of length 2 | The rotation center. If None, the image is rotated around its center, i.e. center=(cols / 2 - 0.5, rows / 2 - 0.5). Please note that this parameter is (cols, rows), contrary to normal skimage ordering. |
|  | order | int, optional | The order of the interpolation [0-5]. The default will be 0 if the image dtype is bool and 1 otherwise. |
|  | mode | {‘\*\*\*’}, optional | Fills in points outside the input's boundaries |
|  | cval | float, optional | Used id the mode is ‘constant’ and it is the value outside the image boundaries. |
|  | clip | bool, optional | Whether to clip the output to the range of values of the input image. The default is true because higher order interpolation may produce values outside the given input range |
|  | preserve\_range | float, optional | Whether to keep the original range of values. Otherwise, the image will be converted to float. |

**Q2**

In the rotation algorithm we are mapping each pixel (x,y) to a new pixel (x’, y’) with the following transformation:

x' = x \* cos(theta) - y \* sin(theta)

y'= x \* sin(theta) + y \* cos(theta)

This transformation results in non-integers values, and this is why we need the interpolation, to approximate this non-integer point using the points nearby.

In addition, there are built-in options to use in the rotate function, like wrap mode and resize that use the interpolation approximation also.

Interpolation is a procedure that estimates an unknown value based on other close and known values, depending on the order of the interpolation.

Bilinear interpolation and considers the closest 2x2 neighborhood of known pixel values surrounding the unknown pixel. It then takes a weighted average of these 4 pixels to arrive at its final interpolated value. This results in much smoother-looking images than the nearest neighbor, which takes a constant value from known pixels.

**Q3**

The main problem that may occur is that the form of the shape will not be preserved, since we are cramming all the information into fewer pixels. In other words, it means that we are decreasing the sampling rate during this process which destroys some elements. In order to overcome this issue, resize() function suggests applying a Gaussian filter (low pass filter) to smooth the image prior to the downsampling, and in this way, we will remove the high frequencies instead of destroying them.

Part 2

**Q1**

Histogram is a bar graph showing how often each different value in a set of data occurs. A histogram is the most commonly used graph to show frequency distributions. In our field of image processing, we can use this histogram to analyze an image - brightness, contrast, equalization, thresholding, etc.

**Q2**

Histogram Equalization is a computer image processing technique used to improve contrast in images. It accomplishes this by effectively spreading out the most frequent intensity values, i.e. stretching out the intensity range of the image

**Q3**

There are two most commonly used methods to convert RGB image to grayscale image:

Average Method

This method is simple but not working well, since the human eyes react differently to RGB - most sensitive to green light, less to the red light and the least to the blues light. Therefore, the three colors should have different weights in the distribution.

Weighted (luminosity) Method

While the weights correspond to the color wavelength, for example.

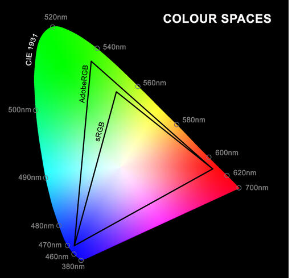
**Q4**

RBG - Red Blue Green

The RBG is a common color space based on 3 primary colors - Red, Blue & Green - and represented by 3 color channels as we studied in Lab #1. While each channel defines its relevant intensity of the pixel. We can say that those 3 primary colors define a color triangle that only colors within this triangle can be reproduced by mixing those 3 primary colors.

Regarding the letter order of the RGB representation, we found that the reason some of the libraries use RGB (i.e. matplotlib) and some use BGR is that in history, there were cameras that worked with different colors representing.

Figure 1 - RGB color spaces on CIE Chromatic Diagram

****

CMYK - Cyan, Magenta, Yellow, Black

Those quartet primary colors are subtractive colors, so we can look at them like filter colors.

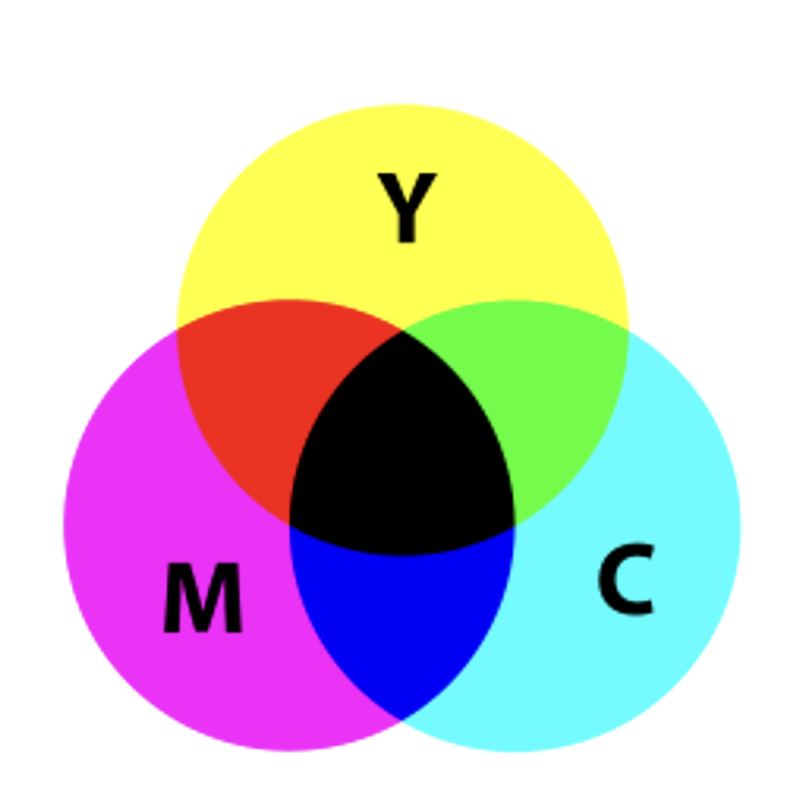
As we can see in Fig.2 there is no color in the background, but as we increase the intensity of each color, we will get the color itself, but where they are mixed and the intensity is high, we will get white color.

Figure 2 - RGB Diagram

****

On the other hand in Fig.3 we see that the background is lighted (white) but when we start to increase the CMY colors, the white light is swallowed, and theoretically, when we mixed those colors, we can even get black. Physically, it is hard to reach the black color by mixing those 3 colors, so we are adding the black color itself and getting the CMYK color space. A known example of this method we can find in printers.

Figure 3 - CMY Diagram

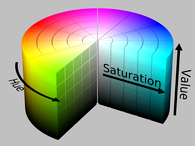
****

HSV - Hue, Saturation, Lightness

HSV is another method to represent colors, by 3 different parameters as you can see in Fig.4.

**Hue** constitutes the color and accept values between 0-360 while 0 is Red, 120 is Green and 240 is Blue. **Saturation** represents the intensity of each color in a range of 0%-100% which means 0% for no color and 100% is the full intensity of the color. The **Value** is the color’s brightness which is also between 0%-100% while the 0% is always Black and 100% could be white or saturated color according to the **S** channel.

Figure 4 - HSV Diagram

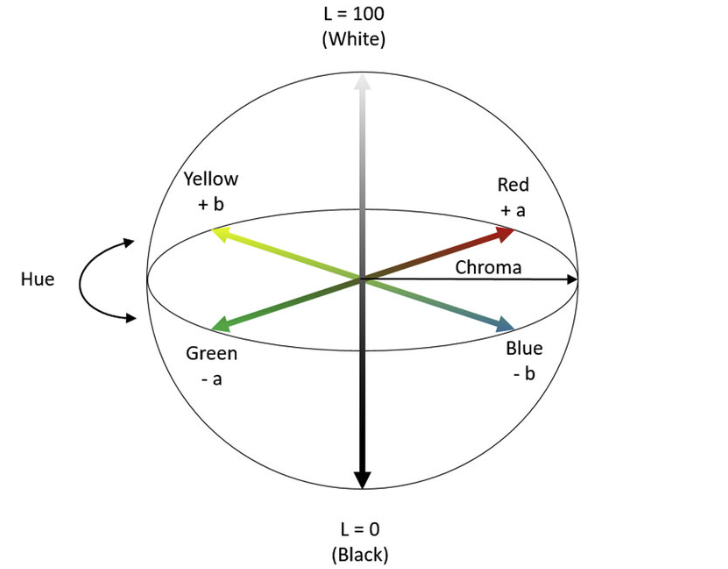


LAB - Lightness, Red/Green Values, Blue/Yellow Values

In this method the color is defined by the following 3 parameters that are represented as axes.

**L\*** - Lightness, that will be between 0% (Black) to 100% (White). While it is represented on the vertical axis. **a\*** value indicates the Red-Green component of a color, and the yellow and blue components are represented on the **b\*** axis. Denote that each of the horizontal axes accepts values between -128 to +128. For example, in a\* -128 for Green and +128 for Red.

Figure 5 - CIELAB Color Space



**Q5**

Mapping the following colors according to RGB and HSV color spaces.

| Color | Name | R, G, B | H, S, V |
| --- | --- | --- | --- |
|  | Black | 0, 0, 0 | 0, 0, 0 |
|  | White | 255, 255, 255 | 0, 0, 100 |
|  | Gray | 127, 127, 127 | 0, 0, 50 |
|  | Red | 255, 0, 0 | 0, 100, 100 |
|  | Green | 0, 255, 0 | 120, 100, 100 |
|  | Blue | 0, 0, 255 | 240, 100, 100 |
|  | Cyan | 0, 255, 255 | 180, 100, 100 |
|  | Magenta | 255, 0, 255 | 300, 100, 100 |
|  | Yellow | 255, 255, 0 | 60, 100, 100 |

**Q6**

From the table above, it is easy to see that the relationship of the comprehensive colors are:

Red ←→ Cyan

Green ←→ Magenta

Blue ←→ Yellow

Note that in the RGB model 2 comprehensive colors summation will give us White representation. While in the HSV model we should add 180 degrees to the Hue channel.

**Q7**

1. hsv\_Image = cv2.cvtColor(img, cv2.COLOR\_BGR2HSV)
2. bgr\_Image = cv2.cvtColor(img, cv2.COLOR\_HSV2BGR\_FULL)

**Q8**

1. h, s, v = cv2.split(hsv\_image)  
   hsv\_image = cv2.merge([h, s, v])