Color Image Processing

By,
Navya Sushma Tummala,
Wahab Nadir Kadiwar,
Jaivardhan Singh Shekhawat,
Fabian Mendez,
Sikender Shahid,
Tyler Driver,
Israel Perez.

Objective:

- To develop a Tkinter based application to perform color image processing operations on an given input image.
- The color image processing operations involve:
    - Color Image Transformations
        - Rgb to HSI (Hue Saturation Intensity)
        - Rgb to CMYK (Cyan, Magenta, Yellow and Black)
    - Color/Intensity Slicing
    - Image smoothening
    - Image Sharpening

Roles and Responsibilities:

Section	Implemented by
Color Image Transformations	Navya Sushma Tummala
Smoothing	Wahab Nadir Kadiwar
Sharpening	Jaivardhan Singh Shekhawat
Intensity/Color Slicing	Sikender Shahid, Israel Perez
GUI	Fabian Mendez
Controller	Tyler Driver

Introduction:

The use of color in image processing is primarily motivated by two major factors:

•Humans can perceive thousands of shades of color as opposed to only about two dozen shades of gray

•Color is a powerful descriptor that greatly simplifies object segmentation and identification

Human vision is more sensitive to color than gray levels. Therefore, color image processing is important, although it requires more memory to store and longer execution times to process. There are different color models, and each one is suitable for some application.

Color Image Transformations:

It is useful to think of a color image as a vector valued image, where each pixel has associated with it, as vector of three values. For instance, there are different models like RGB(Red Green Blue), HIS(Hue Saturation Intensity) and CMYK(Cyan Magenta Yellow Black). In the RGB model, a color image is expressed in terms of the intensities of its red, green, and blue components. In the HSI model, the intensity component is separated from the color components. This model can use the algorithms for gray level images. CMYK uses the colors Cyan Magenta and Yellow, which ideally should result in dark black after combining but, it gives a muddy black color in reality. So other component K(dark black) has been added to CMY components. Some of the processing are based on those of gray level images, and some are exclusive to color images.

I. RGB to HSI:

Step 1: Read the RGB image provided as input to the function

Step 2: Convert the image range from 255 to [0, 1].

Step 3: Use the following formulas for calculating HSI components.

$$H = \begin{cases} \cos^{-1}\left(1/2 \cdot \frac{(R-G)+(R-B)}{\sqrt{(R-G)(R-G)+(R-B)(G-B)}}\right), & \text{if } G \geq B; \\ 360^\circ - \cos^{-1}\left(1/2 \cdot \frac{(R-G)+(R-B)}{\sqrt{(R-G)(R-G)+(R-B)(G-B)}}\right), & \text{Otherwise} \end{cases}$$

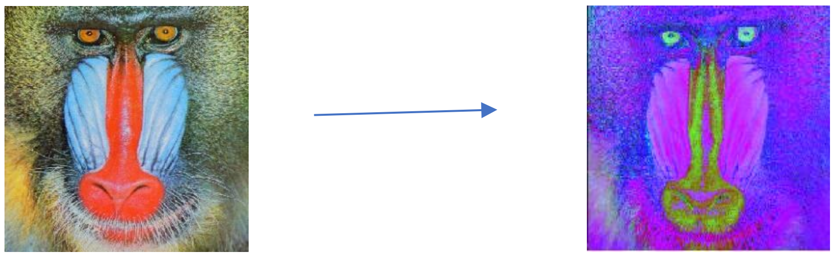
$$H \in [0, 360[$$

$$S = 1 - 3 \cdot \frac{\min\{R, G, B\}}{R+G+B} \quad S \in [0, 1]$$

$$I = \frac{R+G+B}{3} \quad I \in [0, 255]$$

Step 4: Merge the HIS components.

Results:



The left side image is the input image in RGB space and right side image is the resultant output image after merging the channels.

Observations:

(1) Saturation channel resulted in some null values which caused a different output in the initial stage, but adding a very small random integer the output was as expected.

(2) The colors were totally represented differently in the HIS space when compared to RGB. For instance red color in RGB space would be seen as 0^0 of Hue, 100% saturation and 100% intensity.

Difficulties faced:

(1) Calculation of hue resulted in a run time warning "invalid value encountered in double scalars". Fixed this using, np.seterr statement just next to import section of python file.

(2) Played around with different random values for saturation to come up with a optimal one.

(3) Had some issues with the image types that tkinter is able to output. Since it cannot properly display HSL images, we have to display them in a separate window using cv2.

II. RGB to CMYK:

Step 1: Read the RGB image provided as input to the function

Step 2: Convert the image range from 255 to [0, 1].

Step 3: Use the following formulas for calculating CMYK components.

Step 4: Use the below mentioned formula to come up with CMY components.

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Step 5: Calculate the minimum of CMY components.

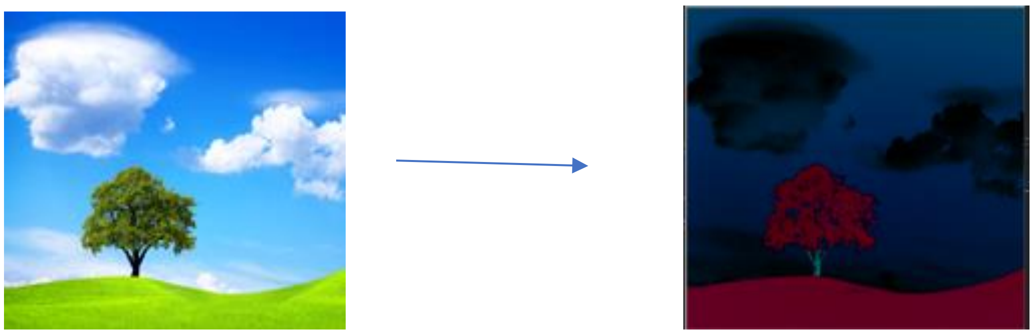
Step 6: If this minimum not equals to 1 then calculate CMYK components using the formula mentioned below.

$$\text{Black (K)} = \text{minimum of } C,M,Y$$
$$\text{Cyan}_{\text{CMYK}} = (C - K)/(1 - K)$$
$$\text{Magenta}_{\text{CMYK}} = (M - K)/(1 - K)$$
$$\text{Yellow}_{\text{CMYK}} = (Y - K)/(1 - K)$$

Step 7: If minimum of CMY is equal to 1, then assign C=0,M=0,Y=0,K=1.

Step 8: Merge the channels to come up with CMYK image.

Results:



The left side image depicts the input and the right side image is the resultant after merging CMYK channels.

Observations:

(1) The combination of all the CMYK channel resulted in a pure black value.
(2) The colors in the left side appear to be more darker than right side.

Difficulties Faced:

(1) Tried to calculate the values of CMYK directly without putting in a loop; this resulted in an error that Boolean values are changing, resolved this by putting them in a loop.
(2) Without multiplication with range factor 100, the output is a bit different.

Image Smoothing

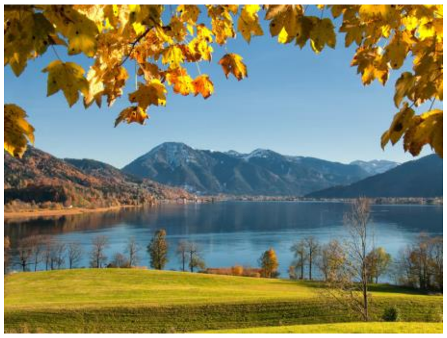
Step 1: Create a 5*5 filter as follows:

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

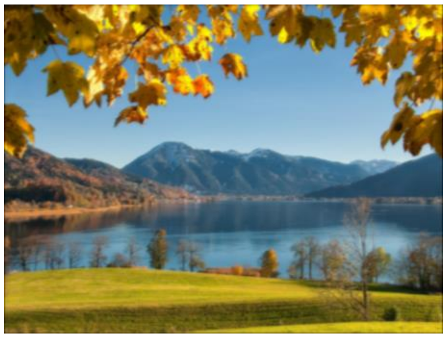
Step 2: Add 4 rows and columns padding to accommodate the 5*5 filter
Step 3: Traverse the filter through the image to obtain the dot product
Step 4: For each of the RGB components in the input image calculate the dot product
Step 5: After getting the dot product evaluate the average value for each component by dividing by 25
Step 6: Restore the image to original size

Results:

Original Image



Smoothened Image



Observations:

1. Patterns can be easily identified in the smoothened image
2. Output image is blurred compared to the input image

Challenges faced:

1. Computing the dot product of a 5*5 matrix was a little challenging
2. Reducing the time and space complexity of the algorithm was also difficult

Image Sharpening

Step 1: Selected the following laplacian filter (3*3):

0	1	0
1	-4	1
0	1	0

Step 2: Added 1 row and column padding on each side to accommodate the 3*3 laplacian filter.
Step 3: Got filtered image by computing the dot product of laplacian filter and original image.
Step 4: Multiplied the filtered image with (-1) to get the final filtered image. Since, we have used this (middle value of filter is negative) type of filter, so need to multiply with (-1).
Step 5: Finally, added the derived filtered image to the original image, to get the sharpened image.
Step 6: Saved the final sharpened image.

The formula used to get the sharpened image is:

Sharpened Image = Input Image + c (∇^2 . Input Image)

where, c = -1

The Laplacian of Vector **c** :

$$\nabla^2\left[\mathbf{c}(x,y)\right] = \begin{bmatrix} \nabla^2 R(x,y) \\ \nabla^2 G(x,y) \\ \nabla^2 B(x,y) \end{bmatrix}$$

Observations:

1. Apparent image quality is improved in sharpened image
2. Blurring is overcomed in sharpened image
3. Legibility is increased in sharpened image

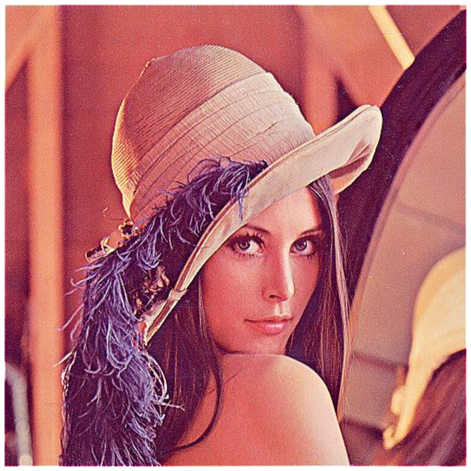
Challenges faced:

1. Saved image under ‘uint8’ format. Saving the image in other formats gave black image.
2. Performing the dot product of laplacian filter and padded original image.

Results:

Original Image

Sharpened Image



Intensity/Color Slicing

Intensity Slicing:

Step 1: Read an image for input.

Step 3: Determine type of slicing (linear or constant)

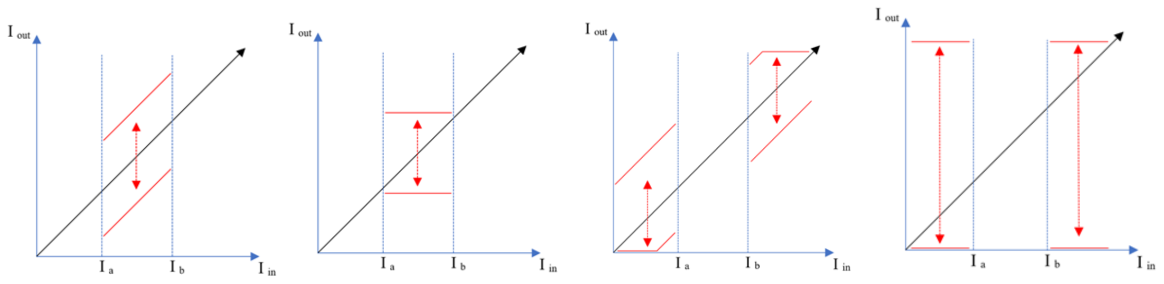
Step 2: Select from one of the given channels. Example: Red, Green or Blue for RGB.

Step 3: Select intensity boundary values [I_a, I_b].

Step 4: Determine whether to do slicing within range [I_a, I_b] or the inverted range [0, I_a] ∪ [I_b, 255].

Step 5: Determine a value for the Gain variable.

The following show the different types of slicing that can occur and the possible ranges the transformed intensities could be. The order is linear slice, constant slice, inverted linear slice and inverted constant slice



Step 6: For each pixel in the image, determine if their intensity value is within the given range and if they are then, based on the type of slicing, do the following:

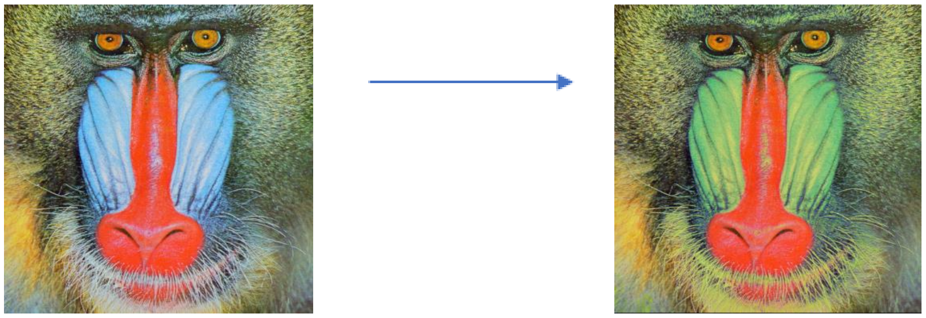
(1) Gain * $I_{i,j}$ (linear slicing)
(2) Gain * Median(I_a, I_b) (constant slicing)

Otherwise keep the intensity value as is.

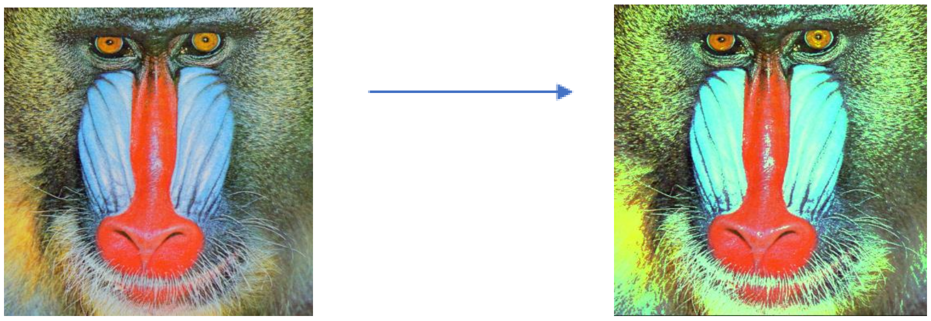
Results

The following are some results that were produced. There are several combinations that can be done and applied to images.

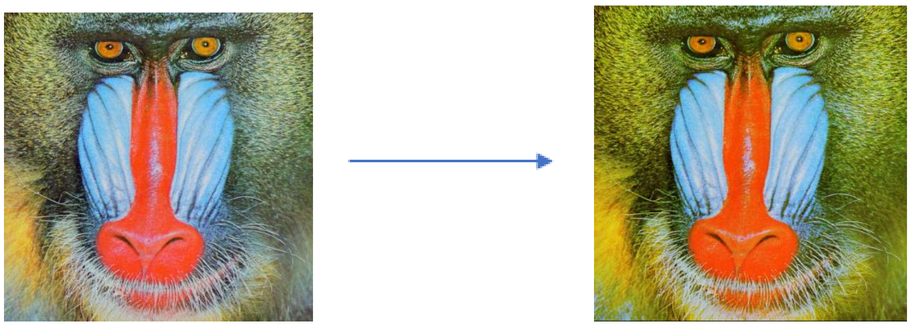
(1) Linear slicing on Blue channel in range [150, 255] with Gain = 0.5



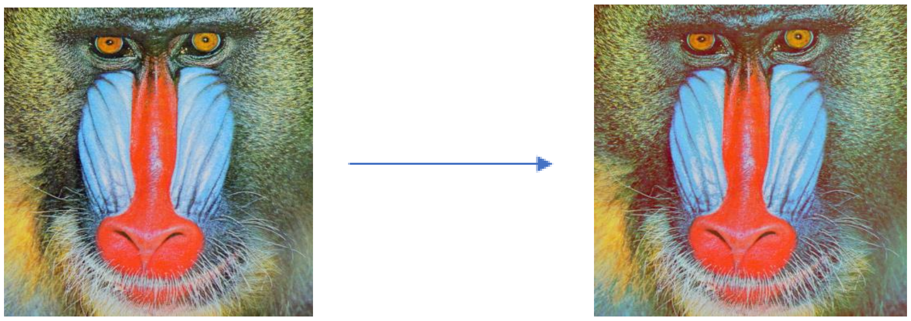
(2) Constant slicing in Green channel on range [150, 255] with Gain = 2



(3) Inverted linear slicing in Blue channel on range [0, 100] and [200, 255] with Gain = 0.5



(4) Inverted constant slicing in Red channel on range [0, 100] and [200, 255] with Gain = 1.5



Observations:

(1) Depending on the Gain value, some colors are more noticeable than others after slicing.

Difficulties faced:

(1) For the inverted slices, the slicing must be done in a specific order. In this implementation, slicing was done for $[0, I_a]$ first then $[I_b, 255]$.
(2) Some test cases were written to make sure that each of different slices were being done as expected.

Gray Level to Color Transformation

As discussed before, humans are more sensitive to colored images than grayscale images. Because of this, there are methods to transform a grayscale image to a color image. These transformations take a grayscale image, in the form of a NxMx1 matrix, to a color image, usually in the RGB model of NxMx3 matrix. An image that undergoes this transformation will be shown in a certain color defined by some weights that are used to convert from grayscale to color. An application to this transformation is that multiple transformations can be applied on an image over multiple intensity ranges to create multiple distinct slices to portray various aspects of an image.

Gray Level to Color Transformation:

For these types of transformations, a grayscale image is transformed to an image of a single color.

Step 1: Read grayscale image as input image I with N rows and M columns.

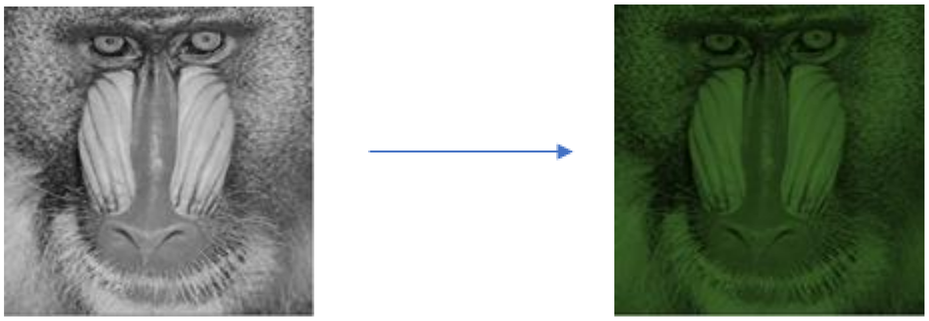
Step 2: Provide weights W to be used for the transformation.

Step 3: Create a NxMx3 matrix T to represent the transformed image.

Step 4: For each pixel in the input image, calculate its RGB value based on the weights given before and store transformed values to corresponding position in matrix T.

$$[T_{R\ i,j}, T_{G\ i,j}, T_{B\ i,j}] = [W_R, W_G, W_B]I_{i,j}$$

Results



For this transformation, the weights are [0.3, 0.5, 0.2].

Observations:

(1) The resulting image contains shadings of the colors it was transformed into like how there are several shadings of gray in the input image.
(2) The value of the weights can be used to transform the input images to colors that are not red, green or blue.

Difficulties Faced:

(1) There are many combinations that can be used for weights.

Density Slicing:

Step 1: Read grayscale image as input I with N rows and M columns.

Step 2: Provide some range [a, b] and a color in RGB.

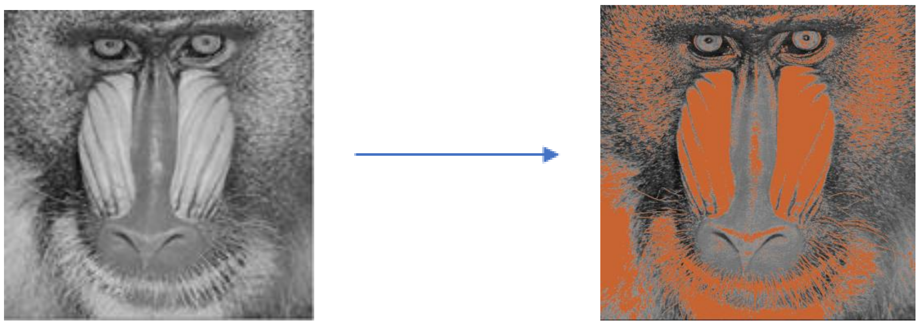
Step 3: Create a NxMx3 matrix T to represent the transformed image.

Step 4: Go through each pixel in the image and determine if their intensity value is within the range [a,b].

Step 5: If the intensity value at pixel i, j is within [a,b], make pixel at that position the color given in Step 2. If the value is not within range, do the following

$$[T_{R\ i,j}, T_{G\ i,j}, T_{B\ i,j}] = [1, 1, 1]I_{i,j}$$

Results:



This result was done using the color [200, 100, 50] at the intensity range [150, 255]. Density slicing can also be applied again to the output image at a different range.

Observations:

(1) This technique shows the locations of where intensities within a range are located within an image.

Difficulties faced:

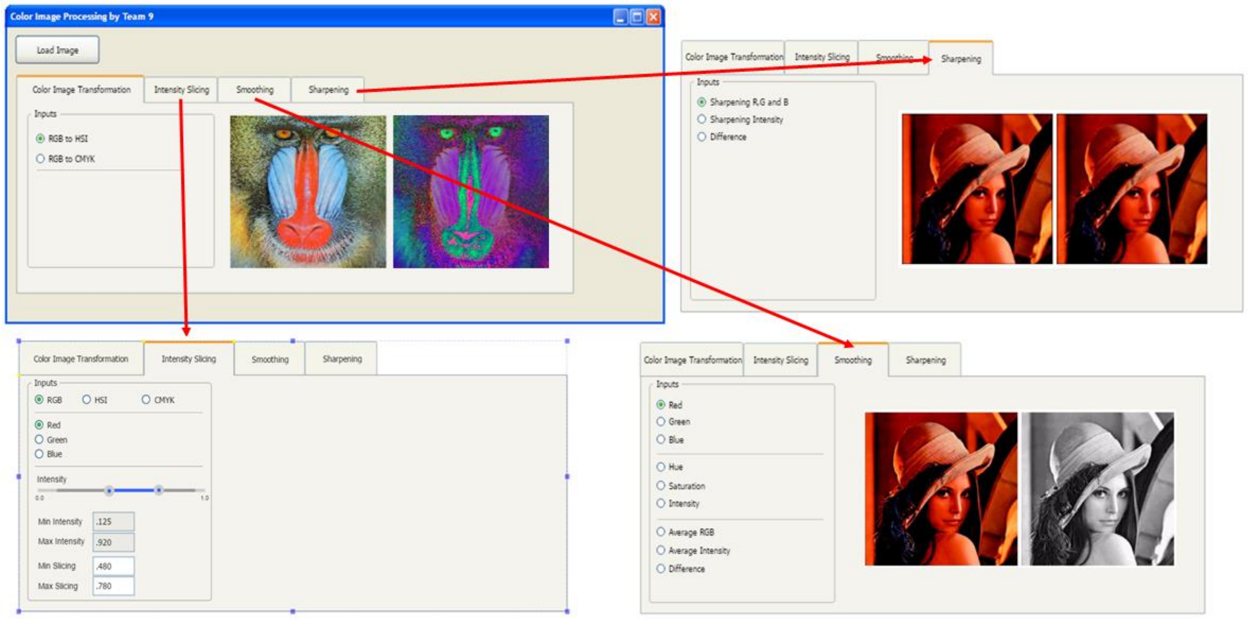
(1) Performance for density slicing can sometimes be slower than regular gray to color transformation.

GUI

The choice of GUI Framework and design were made with the objective of our application and the target audience in mind. Since the Color Image Processing application was made for academic purposes, we decided it is best to build a self contained desktop application, as opposed to a web based application. There are several advantages to taking this approach, the main one being that desktop applications have better performance. This was particularly beneficial since, by avoiding any unnecessary processing overhead, it relieved some of the pressure of having highly efficient implementations for the color transformations.

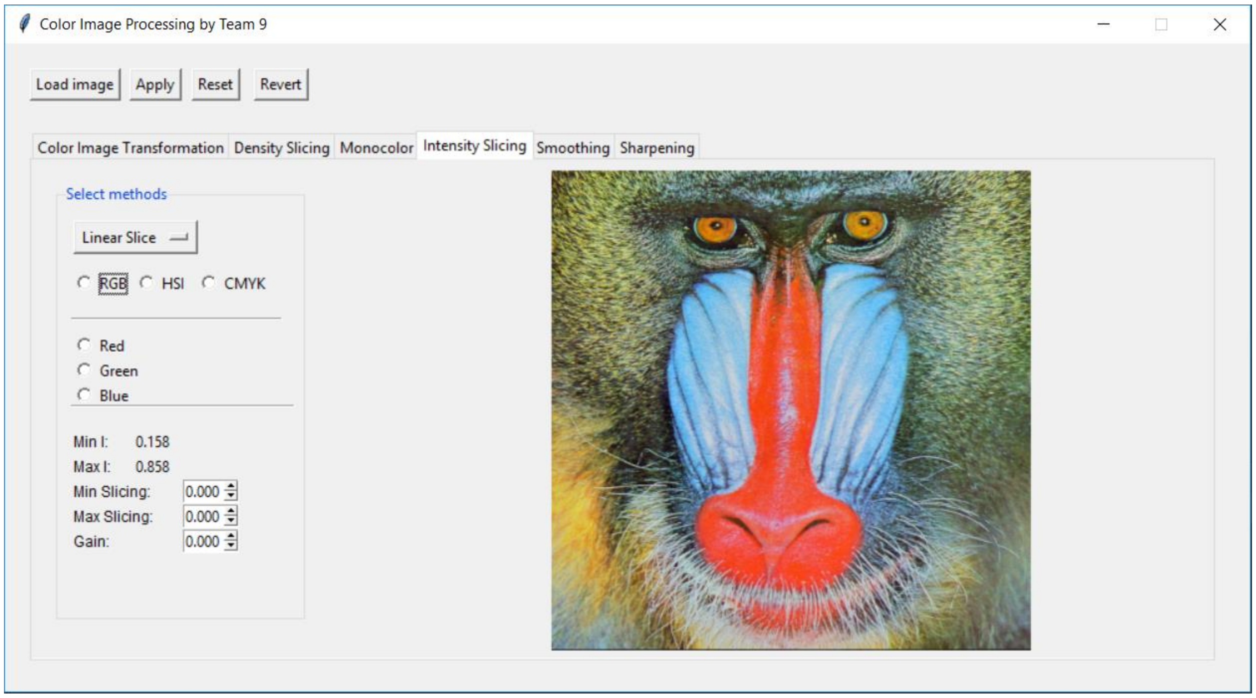
Once we settled on a desktop GUI Framework, the choice was narrowed considerably. Some of the main aspects we looked into was integration with backend, good support for image handling, availability of resources, and level of experience within our team. We first looked into Tkinter because is the standard Python interface and it made common sense to develop both frontend and backend in Python. Since all the modules pertaining to it are native to Python, we knew that all team members already could run it on their machines. It is a widely used framework therefore there is a large amount of resources on how to implement it.

We wanted to present a simple, intuitive interface that could be quickly learned by the user. Therefore, we opted for a minimalistic style, with simply a “Load Image” button at the top and tabs which would present the user with only the options pertaining to the current transformation.



Min Intensity .125
Max Intensity .920
Min Slicing .480
Max Slicing .780

Figure 1. Initial GUI



Min I: 0.158
Max I: 0.858
Min Slicing: 0.000
Max Slicing: 0.000
Gain: 0.000

Figure 2. Final Product

Initially we wanted to have two images side by side, one displaying the original and the second with the processed image. However, due to time constraints we were unable to implement this.

Building the interface and integrating with the backend was fairly straightforward. Tkinter callback functions are simple to use, with variables that can be traced to execute the callback function every time the user changes them. However, some issues were found when attempting to load HSL images into the GUI. This is because tkinter uses PIL images, which does not support HSL color format. Because of this, whenever an image is converted into HSL, it is displayed in a cv2 window rather than in the GUI.