

# Image Processing Lab

All of the programming assignments are to be done in Python using additional libraries specified in the assignments. There are many libraries available, some of which we will be using, and you are welcome to use them with one exception: if the library or a function within it performs the specific function you are asked to code, you may not use that other than perhaps as a reference to compare against. All of the code you submit must be your own. You are welcome to turn in a completed jupyter notebook.

The following code will load an image you can use for this lab. If needed make sure to install PIL using *pip install PIL* or *conda install PIL*.

## Pre-lab Notes

Since it is our first time working with color images, we need to be careful of a couple of things. Most RGB color images store each RGB value as an 8-bit number. This is fine for displaying images as shown below:

```
In [ ]: from scipy.ndimage import imread
import matplotlib.pyplot as plt
import numpy as np

raccoon = imread('raccoon.jpg')
plt.imshow(raccoon); plt.show()
```

This is fine for displaying, but becomes a problem once we try to do any image manipulations. We have to beware of overflow. For example, lets say we just want to add 20 to all of the RGB values.

```
In [ ]: plt.imshow(raccoon + 20)
plt.show()
```

Notice what happens near the white areas of the image. We only had 8-bits to represent each RGB value (0 to 255). White areas will have values near 255. So when we add 20, the colors go crazy because values have overflowed (ex.  $240 + 20 = 4$ ).

You maybe tempted to try something like this:

```
In [ ]: plt.imshow(np.minimum(raccoon + 20, 255))
plt.show()
```

Notice that this still doesn't work because the overflow occurs before the maximum check.

The way to beat this problem is to convert the image into a higher bit representations, do the manipulations, then convert it back down to the 8-bit representation.

```
In [ ]: racoon_32 = np.array(racoon, dtype=np.int32)
        racoon_32 = np.minimum(racoon_32 + 20, 255)
        racoon_8 = np.array(racoon_32, dtype=np.uint8)

        plt.imshow(racoon_8)
        plt.show()
```

For convenience, we will leave all data in int32 representation. Then, we will simply define a function that converts the image back to 8-bit representation before plotting.

```
In [ ]: def plotImage(image, title=""):
        im = np.array(image, dtype=np.uint8)
        plt.imshow(im, vmin = 0, vmax = 255)
        plt.title(title)
        plt.show()

        racoon = imread('racoon.jpg')
        racoon = np.array(racoon, dtype=np.int32)
        plotImage(racoon)
```

You are welcome to use the function above for plotting your own color images.

In this lab, you will also need to be able to convert between RGB values and HSB values. We have provided function that allow you to easily go back and forth while staying in the 0-255 representation for images. You are welcome to use the functions below in this lab.

```
In [1]: def toHSB(image):
        from matplotlib import colors
        temp = 255*colors.rgb_to_hsv(image/255.0)
        return temp.astype(np.int32)

        def toRGB(image):
        from matplotlib import colors
        temp = 255*colors.hsv_to_rgb(image/255.0)
        return temp.astype(np.int32)
```

Now that you have that understanding, you are ready to start the lab.

Implement each of the following functions. Use the provided test cases to test your functions.

## Function 1: Convert to grayscale

Takes in a color image and returns a grayscale image using the following formula: Gray = 0.299 Red + 0.587 Green + 0.114 Blue

```
In [ ]: def toGrayScale(image):
        return None
```

```
In [ ]: # Test Case
        gray_racoon = toGrayScale(racoon)
        plt.imshow(gray_racoon, cmap="Greys_r", vmin=0, vmax=255)
        plt.title("Grayscale Racoon")
        plt.show()
```

## Function 2: Brightness Adjustment

Takes in a color image and returns the brightened version of that image according to a passed in parameter. Use a max image value of 255.

```
In [ ]: def brightAdjust(image,c):
        return None
```

```
In [ ]: # Test Case
        bright_racoon = brightAdjust(racoon,100)
        plotImage(bright_racoon, "Bright Racoon")
        dark_racoon = brightAdjust(racoon,-100)
        plotImage(dark_racoon, "Dark Racoon")
```

## Function 3: Contrast Adjustment

Takes in a color image and returns the contrasted version of that image according to a passed in parameter. Use a max image value of 255.

Also, rather than a straight linear operation, we will use a mapping similar to what Photoshop does. In particular, the contrast will be in the range [-100,100] where 0 denotes no change, -100 denotes complete loss of contrast, and 100 denotes maximum enhancement (8x multiplier). If  $c$  is the contrast parameter, then the level operation applied is:

$$s = \left( \frac{c + 100}{100} \right)^4 (r - 128) + 128$$

Make sure you work in floating point, not integers. Integer division would not be very accurate.

```
In [ ]: def contrastAdjust(image, c):
        return None
```

```
In [ ]: contrast_raccoon = contrastAdjust(raccoon,30)
        plotImage(contrast_raccoon, "High Contrast Raccoon")
```

## Function 4: Image Blending

Takes in 2 color images of the same size. Given an alpha value it returns the blended image according to the alpha value. Note that your alpha value can be a single number or a mask image of the same size. The alpha values will be between 0 and 1.

```
In [ ]: def alphaBlend(image1, image2, alpha = .5):
        return None
```

```
In [ ]: # Test Cases
man = imread("man.jpg")
city = imread("city.jpg")
blended = alphaBlend(man, city, .7)
plotImage(blended, "Alpha Blend with Single Value")

mask1 = imread("alphamask1.jpg")/255.0
blended1 = alphaBlend(man, city, mask1)
plotImage(blended1, "Alpha Blend with Mask 1")

beach = imread("beach.jpg")
boat = imread("boat.jpg")
mask2 = imread("alphamask2.jpg")/255.0
blended2 = alphaBlend(boat, beach, mask2)
plotImage(blended2, "Alpha Blend with Mask 2")
```

## Function 5: Cross Dissolve

Takes in 2 color images of the same size. Returns an array of alpha blend of those two images, where the first picture is an alpha value of 1, the last picture is an alpha value of 0, and the middle pictures slowly decrease until reaching zero. Allow the user to specify the number of steps in the cross dissolve. You can then feed this array into our animation function to view the cross dissolve.

```
In [ ]: def crossDissolve(image1, image2, numsteps = 10):
        return None
```

```
In [ ]: #Test Case
import matplotlib.animation as animation
%matplotlib notebook

beach = imread("beach.jpg")
boat = imread("boat.jpg")
dis = crossDissolve(beach, boat)

fig = plt.figure()
ims = []
for im in dis:
    im = np.array(im, dtype=np.uint8)
    result = plt.imshow(im, vmin=0, vmax=255, animated=True)
    ims.append([result])

ani = animation.ArtistAnimation(fig, ims, interval=500, blit=True)
plt.show()
```

Because we are working in a notebook, this may not display properly. If necessary, plot the individual pictures to verify that the cross dissolve is working. Also, run the following line of code once you are done to return back to normal plotting functions.

```
In [1]: %matplotlib inline
```

## Function 6: Uniform Blurring

Takes in a grayscale image and returns a corresponding result that has been blurred (spatially filtered) using a uniform averaging. Allow the user to specify the size of the kernel (ex. size=3 would give a 3x3 kernel). You can ignore the edge pixels. (Hint: np.sum() may be useful)

```
In [ ]: def blur(image,size=3):
        return None
```

```
In [ ]: # Test Cases
gray_raccoon = toGrayScale(raccoon)
blur_raccoon = blur(gray_raccoon)
plt.imshow(blur_raccoon,cmap="Greys_r",vmin=0,vmax=255); plt.title("Uniform Blurring")
plt.show()

blur_raccoon2 = blur(gray_raccoon, 7)
plt.imshow(blur_raccoon2,cmap="Greys_r",vmin=0,vmax=255); plt.title("Uniform Blurring (7x7)")
plt.show()
```

## Function 7: Median Filter

Takes in a grayscale image and returns a corresponding result that has been median filtered. Allow the user to specify the size of the kernel (ex. size=3 would give a 3x3 kernel). You can ignore the edge pixels.

```
In [ ]: def medianFilter(image,size=3):
        return None
```

```
In [ ]: # Test Cases
gray_raccoon = toGrayScale(raccoon)
median_raccoon = medianFilter(gray_raccoon)
plt.imshow(median_raccoon,cmap="Greys_r",vmin=0,vmax=255); plt.title("Median Blurring")
plt.show()

median_raccoon2 = medianFilter(gray_raccoon, 7)
plt.imshow(median_raccoon2,cmap="Greys_r",vmin=0,vmax=255); plt.title("Median Blurring (7x7)")
plt.show()
```

## Function 8: General Convolution

Now that you have written a couple of different kernels, write a general convolution function that takes in an image and kernel (stored as a numpy matrix), and performs the appropriate convolution. You can assume the kernel is 3x3 if you would like, but it is not much harder to do a general size kernel as well.

```
In [ ]: def convolution(image,kernel):
        return None
```

To make sure your general convolution is working, compare the following test case with your original blur results.

```
In [ ]: # Test Cases
gray_raccoon = toGrayScale(raccoon)

blur_kernel = np.matrix([[1, 1, 1],
                          [1, 1, 1],
                          [1, 1, 1]])

median_raccoon = convolution(gray_raccoon, blur_kernel)/9.0
plt.imshow(median_raccoon,cmap="Greys_r",vmin=0,vmax=255); plt.title("Uniform Blurring")
plt.show()
```

## Function 9: Sharpening

Takes in a grayscale image and returns a corresponding result that has been sharpened using an unsharp masking kernel that has a 6 in the middle and -1s for the four-connected neighbors. You can use your general convolution function. You can ignore the edge pixels. **Don't forget to normalize your results.**

```
In [ ]: def sharpen(image):  
        return None
```

```
In [ ]: # Test Cases  
gray_raccoon = toGrayScale(raccoon)  
sharpen_raccoon = sharpen(gray_raccoon)  
plt.imshow(sharpen_raccoon, cmap="Greys_r", vmin=0, vmax=255); plt.title("Sharpened")  
plt.show()
```

## Function 10: Edge Detection

Takes in a grayscale image and returns a corresponding result that shows the gradient magnitude of the input. Use a Sobel kernel. You may afterward scale the result for visibility if you wish when you demonstrate the function. You can use your general convolution function. You can ignore the edge pixels.

```
In [ ]: def edgeDetect(image):  
        return None
```

```
In [ ]: # Test Cases  
gray_raccoon = toGrayScale(raccoon)  
edge_raccoon = edgeDetect(gray_raccoon)  
plt.imshow(edge_raccoon, cmap="Greys_r", vmin=0, vmax=255); plt.title("Edge Detection")  
plt.show()
```

```
In [ ]:
```