CHAPTER 8

Basic Image Processing with NumPy and Matplotlib

In the last chapter, we have learned how to modify the default settings of visualizations of matplotlib. We also have studied multiline and single-line plots, in detail. We saw how to visualize NumPy data with matplotlib. We, also, have experimented with colors, line styles, and markers.

Now, we know NumPy and Matplotlib and can comfortably work with them. We have covered all the required topics needed to get started with image processing. So, in this chapter, we will learn the basic image processing concepts and implement them with NumPy and matplotlib.

8.1 Image Datasets

As, we are going to learn the topic of **image processing**, we will need a lot of test images. You can use any of the image. You can, even, capture them with digital camera or scan the printed film photographs from your family archives. However, the best option is to use the standard sets of photographs, used by the worldwide community of image processing researchers. You can find such sets at the following URLs,

http://sipi.usc.edu/database/

http://www.imageprocessingplace.com/root_files_V3/image_databases.htm

Also, University of Tsukuba (http://www.tsukuba.ac.jp/en/) has got excellent set of images for advanced image processing and computer vision operations. Following is the URL for their stereo dataset,

http://www.cvlab.cs.tsukuba.ac.jp/dataset/tsukubastereo.php

You can, also, find other datasets there.

So, download the test images from these URLs and store them in a local directory of your computer or Pi. We will be using them throughout the chapter and the rest of the book as test images.

8.2 Installing Pillow

We need **Python Imaging Library (PIL)** for reading images that are not in PNG format, using matplotlib (we're going to see it next). However, its development appears to be discontinued and there is a newer version of it known as **pillow,** which is under active development. We can check more details about it, at https://pillow.readthedocs.io/en/stable/. We run the following command to install it on Windows,

```
pip3 install pillow
```

We run the following command on terminal to install it on Raspberry Pi,

```
sudo pip3 install pillow
```

8.3 Reading and saving images

Let's start with reading, displaying, and saving images with matplotlib. First, import all the required libraries and enable matplotlib visualization with the magic command, as follows,

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

We can read image into a variable, as follows,

```
img1 = plt.imread('/home/pi/Dataset/4.1.06.tiff')
```

Note, that, the above code is for Linux and Raspberry Pi. For Windows OS, the code will be, as follows,

```
img1 = plt.imread('D:\\Dataset\\4.1.06.tiff')
```

I will, mostly, be using a Windows computer for the programming. However, the code, when we make appropriate changes for Linux platform, works well with Raspberry Pi Raspbian OS well.

We have to use `plt.imshow()` function, followed by `plt.show()`, to visualize the image,

```
plt.imshow(img1)
```



```
plt.axis('off')
plt.title('Tree')
plt.show()
```

This will show a color image, as it is. However, when it comes to the greyscale images then it's a bit tricky. Greyscale images are shown, with a default colormap. Following is an example code that reads and displays a grayscale image,

```
img1 = plt.imread('D:\\Dataset\\5.3.01.tiff')
plt.imshow(img1)
plt.axis('off')
plt.show()
```

The output is as follows, *(figure 8.1)*

Figure 8.1 A greyscale image rendered with default colormap

As you can see that the greyscale image is tinted. We can avoid this by applying grey colormap, as follows,

```
img1 = plt.imread('D:\\Dataset\\5.3.01.tiff')
plt.imshow(img1, cmap='gray')
plt.axis('off')
plt.show()
```

Run the above program and you will see the greyscale image without tinting.

We can save the image to a location on a disk, as follows,

```
plt.imsave('/home/pi/output.png', img1)
```

For windows, it will be,

```
plt.imsave('D:\\output.png', img1)
```

8.4 NumPy for Images

When, we use Python programming for image processing operations, all the images that's been used, are read and stored in memory (RAM), as NumPy ndarrays. We have already seen ndarrays and their properties. These properties have meaning when we work with ndarrays representing images. Let's see the example. Following are the properties of a color image,

```
img1 = plt.imread('D:\\Dataset\\4.1.06.tiff')
print(type(img1))
print(img1.shape)
print(img1.ndim)
print(img1.size)
print(img1.dtype)
print(img1.nbytes)
```

Following is the output,

```
<class 'numpy.ndarray'>
(256, 256, 3)
3
196608
uint8
196608
```

Let's, analyze it, one by one. The first output confirms that the image is indeed represented as a NumPy array. The property `shape` returns the shape of image. In the returned tuple, first two numbers represent width and height, respectively, (i.e. resolution). The last number in the tuple represents the number of color channels. For all the color image, it will mostly be 3 or 4. The property `ndim` represents the number of dimensions. It is 3, for a color image, and 2, for a greyscale image. The property `size` is the size of the image, which means how many data points or numbers it takes to represent the image. If you multiply all the numbers in the tuple returned by the property `shape`, it is exactly the same number.

The property `dtype` tells us how we are representing each data point for the image. Here, we are using `uint8` which means unsigned integer of 8 bits (1 byte). Finally, `nbytes` tell us how many bytes are required to store it in memory (RAM). It is the multiplication of the size and the memory required for the given datatype, in which, image data points are represented. Here, `uint8` takes 1 byte each and the size is 196608, thus, the number of bytes required are 196608 x 1 = 196608.

Let's do the same for a greyscale image,

```
img2 = plt.imread('D:\\Dataset\\5.3.01.tiff')
print(type(img2))
print(img2.shape)
print(img2.ndim)
print(img2.size)
print(img2.dtype)
print(img2.nbytes)
```

Following is, the output,

```
<class 'numpy.ndarray'>
(1024, 1024)
2
1048576
uint8
1048576
```

The property `shape` will return a tuple with two numbers that refers to resolution of the image. There is only one channel in any greyscale image that stores the intensity values of the grey color, for all the pixels, in the image. Thus, the number of dimensions, the property `ndim`, is 2. The image data points are still represented with `uint8`. Rest of the two properties are dependent on these three properties and can be deduced, once we have these three values.

We can access value for individual channel in a color image, as follow,

```
print(img1[10, 10, 0])
print(img1[10, 10, 1])
print(img1[10, 10, 2])
```

The code above returns the values for channel 0, channel 1, and channel 2 at pixel 10, 10, as follows,

```
199
216
220
```

We can even use the slicing operation, as follows,

```
print(img1[10, 10, :])
```

Output is, as follows,

```
[199 216 220]
```

For a greyscale image, we can know the value of individual pixel, as follows,

```
print(img2[10, 10])
```

The output is,

```
71
```

8.5 Image Statistics

We can retrieve the image statistics with the NumPy ndarray properties, as follows,

```
img1 = plt.imread('D:\\Dataset\\4.1.06.tiff')
print(img1.min())
print(img1.max())
print(img1.mean())
```

Run the above code and check the output. **NumPy** has some statistical functions to retrieve statistics of a NumPy array. Following are those functions,

```
print(np.median(img1))
print(np.average(img1))
print(np.mean(img1))
print(np.std(img1))
print(np.var(img1))
```

Run the above code and check the output.

8.6 Image Masks

We can create the **masks** for the images to cover the images with pixels of a particular color. Following is the code sample,


```
img1 = plt.imread('D:\\Dataset\\4.1.06.tiff')
nrows, ncols, channels = img1.shape
row, col = np.ogrid[:nrows, :ncols]
cnt_row, cnt_col = nrows/2, ncols/2
outer_disk_mask = ((row - cnt_row) ** 2 + (col - cnt_
col) ** 2 > (nrows/2) ** 2)
img1.setflags(write=1)
img1[outer_disk_mask]=0
plt.imshow(img1)
plt.axis('off')
plt.title('Masked Image')
plt.show()
```

In the above code, initially, we are creating an outer disk mask and, then, setting all the pixels in the mask to the value 0 (0 means black). The function `setflags(write=1)` allows us to modify the pixels in the image stored in memory (**RAM**). Following is the output, *(figure 8.2)*

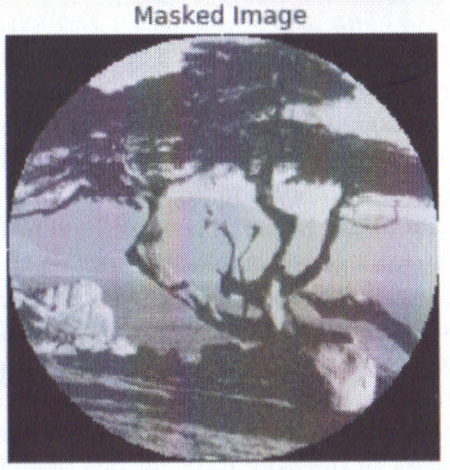

Figure 8.2 Masked image

8.7 Image Channels

We know that, for colored images, the information to the colors is saved in separate channels. There are many schemes to do that. When `plt.imread()` reads a color image, it stores the color information in NumPy array, such that, the information for colors red, green, and blue is stored in

color channels separately. It can be retrieved by slicing the NumPy array that stores the color image, as follows,

```
img1 = plt.imread('D:\\Dataset\\4.1.04.tiff')


r = img1[:, :, 0]
g = img1[:, :, 1]
b = img1[:, :, 2]

output = [img1, r, g, b]

titles = ['Image', 'Red', 'Green', 'Blue']

for i in range(4):
    plt.subplot(2, 2, i+1)

    plt.axis('off')
    plt.title(titles[i])
    if i == 0:
        plt.imshow(output[i])
    else:
        plt.imshow(output[i], cmap='gray')

plt.show()
```

In the above code, we are retrieving the information for color channels in the separate two-dimensional ndarray for each color. So, we have 3 different two-dimensional ndarrays with datatype `uint8`. We know that an unsigned integer of 8 bit can store 256 values ranging from 0 to 255. Each value represents an intensity level for that particular color. A **pixel** is made of information related to 3 color channels. Thus, a pixel can have any one of the possible 256 x 256 x 256 = 16777216 color values. That is, around 16 and half million colors. Human eye can distinguish around 10 million colors. So, this 3-byte representation of a color pixel is adequate for humans. The output of the code above is as follows,

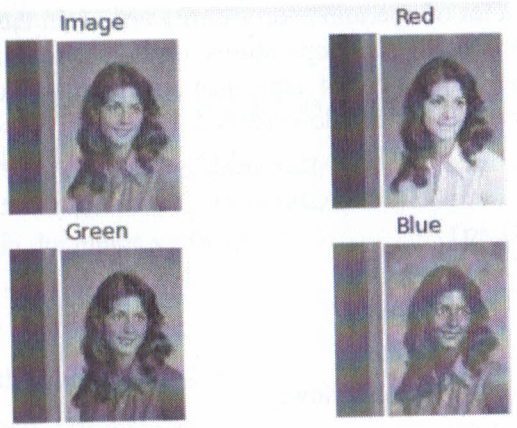

Figure 8.3 Separated color channels

As you can see, we are able to separate and display the color channels. *(figure 8.3)*

Also, if you have noticed, we are displaying multiple images in matplotlib output. This is possible due to `subplot()` function. This function allows us to treat the output, as a grid and set the output images in that grid. The first two arguments passed to `subplot()` to convey size of the grid. In this example, it is 2 x 2. The last argument passed conveys the position of the image in this grid. The top left position is **1**, the horizontally adjacent position is **2**, and so on. This is very useful, as we will be using the same template of code to display multiple images in a single output, throughout the book. We can even show multiple plots separately this way. We will see that too in this book.

We can even recombine them to form original image, as follows,

```
output = np.dstack((r, g, b))
plt.imshow(output)
plt.show()
```

Run the above code and verify, if the output is the original image.

8.8 Arithmetic Operations on Images

We can perform a lot of arithmetic operations on the images. For this section, I am not showing the output, in the book. Instead, I want you to run all the code examples below and check the outputs.

We have seen a lot of operations on NumPy arrays, in the earlier section. We are going to see the same operations, in this section, in the context of images. Choose two images of same size resolution and dimension from the image dataset that's been downloaded. Following is the sample code,

```
img1 = plt.imread('D:\\Dataset\\4.1.06.tiff')
img2 = plt.imread('D:\\Dataset\\4.1.04.tiff')
plt.imshow(img1)
plt.show()
plt.imshow(img2)
plt.show()
```

We can add two images, as follows,

```
plt.imshow(img1 + img2)
plt.show()
```

We know that the addition operation over numbers is commutative. Changing position of operands does not change the final result. So, following is the equivalent of the above,

```
plt.imshow(img1 + img2)
plt.show()
```

We can subtract one image from the other, as follows,

```
plt.imshow(img1 - img2)
plt.show()
```

We, also, know that the subtraction operation is not commutative. So, the following code produces different output,

```
plt.imshow(img2 - img1)
plt.show()
```

We can flip the image, as follows,

```
plt.imshow(np.flip(img1, 0))
plt.show()
```

We can even change the axis of flip, as follows,

```
plt.imshow(np.flip(img1, 1))
plt.show()
```

We can roll the image, as follows,


```
plt.imshow(np.roll(img1, 2048))
plt.show()
```

We can flip the image from left to right, as follows,

```
plt.imshow(np.fliplr(img1))
plt.show()
```

We can even flip the image vertically,

```
plt.imshow(np.flipud(img1))
plt.show()
```

We can rotate the image, as follows,

```
plt.imshow(np.rot90(img1))
plt.show()
```

8.9 Bitwise Logical Operations

We can use NumPy bitwise logical operation functions on images. We know that all the bitwise logical operations are commutative. We will use same images that we used in the earlier section for the demonstration of logical operations. Following code demonstrates **bitwise logical AND** between images,

```
plt.imshow(np.bitwise_and(img1, img2))
plt.show()
```

Following is the code for bitwise logical OR operation,

```
plt.imshow(np.bitwise_or(img1, img2))
plt.show()
```

Bitwise logical XOR can be achieved, as follows,

```
plt.imshow(np.bitwise_xor(img2, img1))
plt.show()
```

Bitwise logical NOT is nothing but negative of an image, as follows,

```
plt.subplot(1, 2, 1)
plt.imshow(img1)
plt.subplot(1, 2, 2)
plt.imshow(np.bitwise_not(img1))
plt.show()
```

8.10 Image Histograms with NumPy and Matplotlib

In the last chapter, we saw the code for computing histogram of a NumPy ndarray. However, we did not discuss the concept in detail. I was saving it for this occasion when I can explain it in visual way. A **histogram** is a visual representation of distribution of data in a dataset. In simple words, it is visualization of frequency distribution table of a dataset. A **distribution table** is nothing, but a table of values in a dataset versus the number of occurrences of those values in that dataset. The **dataset**, here, is an image represented by NumPy ndarray. Let's create and visualize the chnnelwise histograms of a color image. The code is as follows,

```
img1 = plt.imread('D:\\Dataset\\4.1.01.tiff')


r = img1[:, :, 0]
g = img1[:, :, 1]
b = img1[:, :, 2]
```

In the above code, we are separating the channels, by slicing the NumPy array, that holds the image. Let's adjust the space between the subplots with the following code,

```
plt.subplots_adjust(hspace=0.5, wspace=0.5)
```

Now let's plot the original image,

```
plt.subplot(2, 2, 1)
plt.title('Original Image')
plt.imshow(img1)
```

Now, let's compute the histogram for the red channel,

```
hist, bins = np.histogram(r.ravel(), bins=256,
range=(0, 256))
```

The red channel is a two dimensional NumPy ndarray. First, we are flattening it and passing it, as an argument, to `np.histogram()` function. We are, also, mentioning the number of bins and range of the values for which the histogram is to be computed. Finally, we will use `plt.bar()` to show the histogram visually,

```
plt.subplot(2, 2, 2)
plt.title('Red Histogram')
plt.bar(bins[:-1], hist)
```


Same way, we can compute and visualize the histograms for other channels as follows,

```
hist, bins = np.histogram(g.ravel(), bins=256,
range=(0, 256))
plt.subplot(2, 2, 3)
plt.title('Green Histogram')
plt.bar(bins[:-1], hist)

hist, bins = np.histogram(b.ravel(), bins=256,
range=(0, 256))
plt.subplot(2, 2, 4)
plt.title('Blue Histogram')
plt.bar(bins[:-1], hist)

plt.show()
```

The output is as follows, *(figure 8.4)*

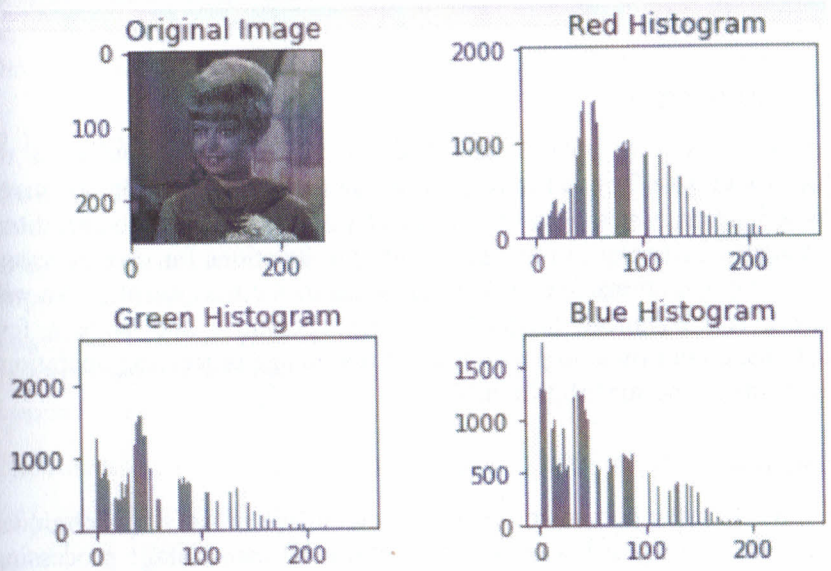

Figure 8.4 Channelwise histograms for a color image

We can directly use `plt.hist()` function in matplotlib to compute and visualize histogram, as follows,

```
plt.subplots_adjust(hspace=0.5, wspace=0.5)
plt.subplot(2, 2, 1)
plt.title('Original Image')
plt.imshow(img1)
plt.subplot(2, 2, 2)
plt.title('Red Histogram')
plt.hist(r.ravel(), bins=256, range=(0, 256))
plt.subplot(2, 2, 3)
plt.title('Green Histogram')
plt.hist(g.ravel(), bins=256, range=(0, 256))
plt.subplot(2, 2, 4)
plt.title('Blue Histogram')
plt.hist(b.ravel(), bins=256, range=(0, 256))
plt.show()
```

The output of the code, above, will exactly be the same as the output of the earlier code.

This is how we can compute and visualize the histogram of any image or NumPy ndarray.

8.11 Summary

In this chapter, we have studied the image processing operations on images with **NumPy** and **matplotlib**. We have not worked with any image processing library yet. **NumPy** itself can be used for most basic operations on images. We can even implement our own functions for various image processing operations. We will do the same in the next chapter. We will study a few advanced image processing operations and write a few functions, on our own, to perform those few image processing operations with **NumPy** and **matplotlib** only.

Exercise

We have seen the demonstration of various image processing techniques for the color images. I want you to perform all these image processing operations for greyscale images. It will be an interesting exercise.

CHAPTER 9

Advanced Image Processing with NumPy and Matplotlib

In the last chapter, we have got started with the image processing programming. We have learned how to implement basic image processing operations with **NumPy** and **matplotlib**. We have studied and implemented very basic operations, without using any dedicated library for image processing.

We will continue with the image processing operations with **NumPy** and **matplotlib**. We will study a bit more advanced operations and we will write the functions for them, ourselves. We will study the concepts like thresholding, color to greyscale, normalization, and other operations.

9.1 Color to Greyscale Conversion

We know that a color image has 3 channels and it is represented with **RGB** colorspace. We can convert a color image to a greyscale image. We need to convert values represented by 3 channels to a single channel. First, we import all the required libraries, following is the code:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

The custom function to **convert** color image to greyscale image, as follows:

```
def rgb2gray(img):
    r = img[:, :, 0]
    g = img[:, :, 1]
    b = img[:, :, 2]
    return (0.2989 * r
```