CHAPTER 5

Advanced Operations on Images

The last chapter explored the arithmetic and logical operations on images with Pillow. There is more to the world of image processing than that. Pillow has a lot more functionality to offer. You can enhance and filter images. You can also calculate histogram of image and its channels. You will learn all these advanced operations with Pillow in this chapter.

The ImageFilter Module

You can use the `ImageFilter` module in Pillow to perform a variety of filtering operations on images. You can use filters to remove noise, to add blur effects, and to sharpen and smooth your images. Listing 5-1 shows a simple image-filtering program using the `ImageFilter` module.

Listing 5-1. prog01.py

```
from PIL import Image, ImageFilter

im1 = Image.open("/home/pi/DIP/Dataset/4.1.08.tiff")

im2 = im1.filter(ImageFilter.BLUR)
im2.show()
```


A. Pajankar, *Raspberry Pi Image Processing Programming*,
DOI 10.1007/978-1-4842-2731-2_5

Figure 5-1 shows the output of this program.

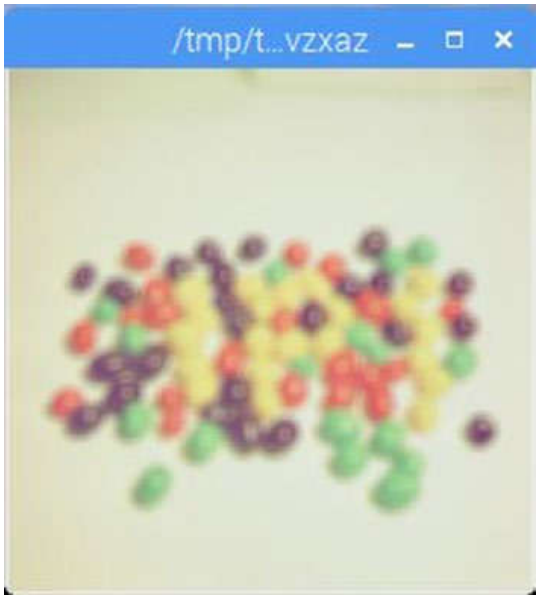

Figure 5-1. *The blur operation*

Apart from BLUR, Pillow has the following filters. Modify the previous code and try all the filters.

```
CONTOUR
DETAIL
EDGE_ENHANCE
EDGE_ENHANCE_MORE
EMBOSS
FIND_EDGES
SMOOTH
SMOOTH_MORE
SHARPEN
```

You can also define custom filters. Listing 5-2 shows how to define a custom filter.

Listing 5-2. prog02.py

```
from PIL import Image, ImageFilter

im1 = Image.open("/home/pi/DIP/Dataset/4.1.08.tiff")

custom_filter = ImageFilter.GaussianBlur(radius=float(5))
im2 = im1.filter(custom_filter)
im2.show()
```

The code in Listing 5-2 uses the GaussianBlur() method for a custom filter. The radius of the blur is 5. You can change the radius. Let's use Tkinter to modify the code and make the blur radius dynamic by using the slider in Tkinter (see Listing 5-3).

Listing 5-3. prog03.py

```
from PIL import Image, ImageTk, ImageFilter
import tkinter as tk

def show_value_1(blur_radius):
    print('Gaussian Blur Radius: ', blur_radius)

    custom_filter = ImageFilter.GaussianBlur(radius=float(blur_radius))
    img = im1.filter(custom_filter)
    photo = ImageTk.PhotoImage(img)
    l['image'] = photo
    l.photo = photo

root = tk.Tk()
root.title('Gaussian Blur Filter Demo')

im1 = Image.open("/home/pi/DIP/Dataset/4.1.04.tiff")

photo = ImageTk.PhotoImage(im1)

l = tk.Label(root, image=photo)
l.pack()
l.photo = photo

w1 = (tk.Scale(root, label="Blur Radius", from_=0, to=10,
      resolution=0.2, command=show_value_1, orient=tk.HORIZONTAL))
w1.pack()

root.mainloop()
```

The output of Listing 5-3 is shown in Figure 5-2.

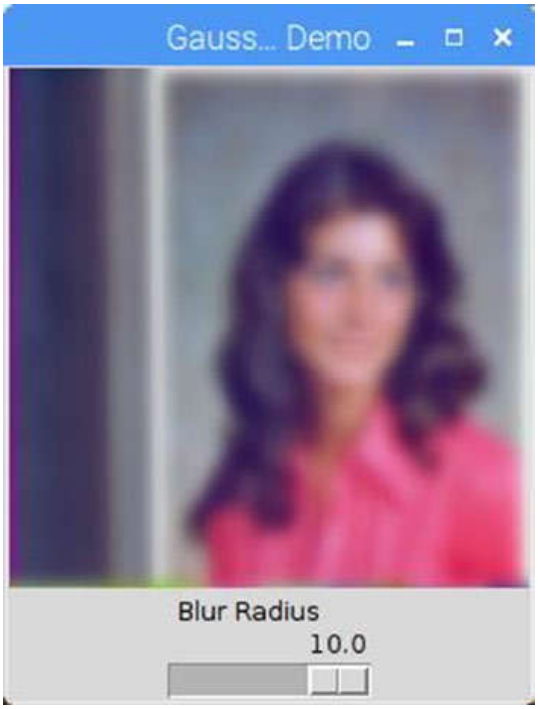

Figure 5-2. *Gaussian blur demo*

You can see the `GaussianBlur()` has been applied with 10 as the radius of blur in the figure.

The next filter you will explore is the convolution filter. You need to understand the concept of *kernels* for this example. Kernels are the square matrices used in image processing operations. You mostly use kernels for filtering. Kernels produce a variety of effects, including blurs, smoothing, and noise reduction. They are especially used to remove high-frequency components from an image. This process is also called low-pass filtering.

Figure 5-3 shows an example kernel.

```
1   1   1
1  -4   1
1   1   1
```

Figure 5-3. *An example kernel*

Listing 5-4 shows a code example of the kernel used for image convolution.

Listing 5-4. prog04.py

```
from PIL import Image, ImageTk, ImageFilter
import tkinter as tk

root = tk.Tk()
root.title('Convolution Kernel Demo')

im1 = Image.open("/home/pi/DIP/Dataset/4.1.05.tiff")

custom_filter = ImageFilter.Kernel((3, 3), [1, 1, 1, 1, -4, 1, 1, 1, 1])

img = im1.filter(custom_filter)

photo = ImageTk.PhotoImage(img)

l = tk.Label(root, image=photo)
l.pack()
l.photo = photo

root.mainloop()
```

Run the program in Listing 5-4 and check out the output. As of now, the `ImageFilter.Kernel()` method supports **(3,3)** and (**5,5**) kernels only.

You can use the Digital Unsharp Mask filter, as shown in Listing 5-5.

Listing 5-5. prog05.py

```
from PIL import Image, ImageTk, ImageFilter
import tkinter as tk

def show_value_1(blur_radius):
    print('Unsharp Blur Radius: ', blur_radius)

    custom_filter = ImageFilter.UnsharpMask(radius=float(blur_radius))
    img = im1.filter(custom_filter)
    photo = ImageTk.PhotoImage(img)
    l['image'] = photo
    l.photo = photo

root = tk.Tk()
root.title('Digital Unsharp Mask Demo')

im1 = Image.open("/home/pi/DIP/Dataset/4.1.04.tiff")

photo = ImageTk.PhotoImage(im1)
```

```
l = tk.Label(root, image=photo)
l.pack()
l.photo = photo

w1 = (tk.Scale(root, label="Blur Radius", from_=0, to=10,
      resolution=0.2, command=show_value_1, orient=tk.HORIZONTAL))
w1.pack()

root.mainloop()
```

The digital unsharping mask sharpens the image. Figure 5-4 shows the output with a mask radius of size 10.

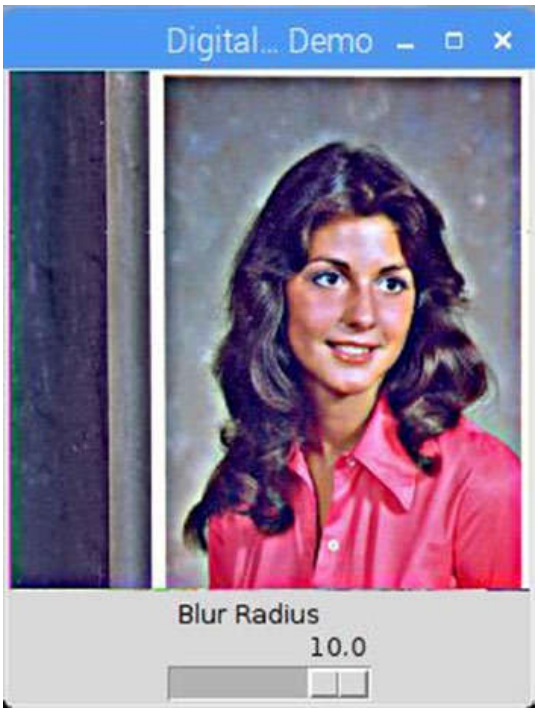

Figure 5-4. *Digital unsharp demo*

The radius of size 10 creates a highly sharpened image.

Let's study the median, min, and max filters in Pillow. All these filters only accept an odd number as the window size. The code in Listing 5-6 demonstrates a median filter.

Listing 5-6. prog06.py

```
from PIL import Image, ImageTk, ImageFilter
import tkinter as tk

def show_value_1(window_size):
    print('Window Size: ', window_size)

    if (int(window_size) % 2 == 0):
```

```
        print("Invalid Window Size...\nWindow Size must be an odd number...")
    else:
        custom_filter = ImageFilter.MedianFilter(size=int(window_size))
        img = im1.filter(custom_filter)
        photo = ImageTk.PhotoImage(img)
        l['image'] = photo
        l.photo = photo

root = tk.Tk()
root.title('Median Filter Demo')

im1 = Image.open("/home/pi/DIP/Dataset/4.1.04.tiff")

photo = ImageTk.PhotoImage(im1)

l = tk.Label(root, image=photo)
l.pack()
l.photo = photo

w1 = (tk.Scale(root, label="Window Size", from_=1, to=19,
      resolution=1, command=show_value_1, orient=tk.HORIZONTAL))
w1.pack()

root.mainloop()
```

The output shown in Figure 5-5 has a window size of 19.

Figure 5-5. *A median filter*

If you change the filter to a min filter using the following code line:

```
custom_filter = ImageFilter.MinFilter(size=int(window_size))
```

You'll get the output shown in Figure 5-6.

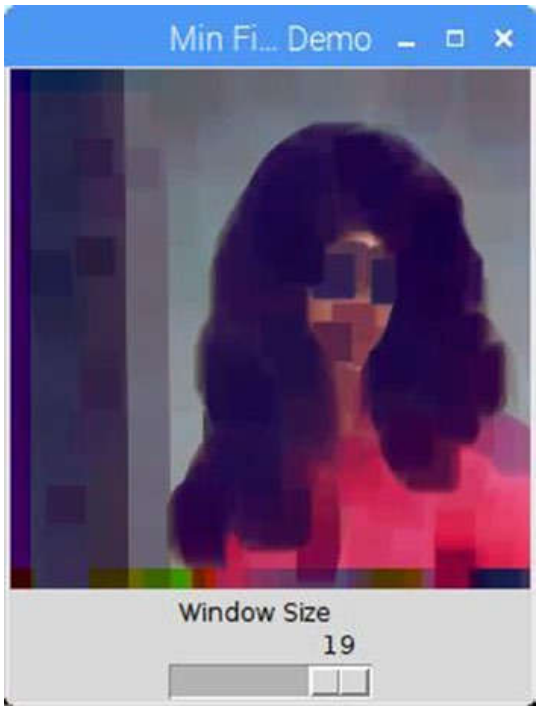

Figure 5-6. *A min filter*

Here's an example of the max filter:

```
custom_filter = ImageFilter.MaxFilter(size=int(window_size))
```

The output for the max filer is shown in Figure 5-7.

Figure 5-7. *The max filter*

Next, you'll see an example of a mode filter. The mode filter works with even and odd window sizes. Listing 5-7 shows a code example of the mode filter.

Listing 5-7. prog09.py

```
from PIL import Image, ImageTk, ImageFilter
import tkinter as tk

def show_value_1(window_size):
    print('Window Size: ', window_size)

    custom_filter = ImageFilter.ModeFilter(size=int(window_size))
    img = im1.filter(custom_filter)
    photo = ImageTk.PhotoImage(img)
    l['image'] = photo
    l.photo = photo

root = tk.Tk()
root.title('Mode Filter Demo')

im1 = Image.open("/home/pi/DIP/Dataset/4.1.04.tiff")

photo = ImageTk.PhotoImage(im1)
```

```
l = tk.Label(root, image=photo)
l.pack()
l.photo = photo

w1 = (tk.Scale(root, label="Window Size", from_=1, to=19,
      resolution=1, command=show_value_1, orient=tk.HORIZONTAL))
w1.pack()

root.mainloop()
```

The output with a window size of 19 is shown in Figure 5-8.

Figure 5-8. *The mode filter*

The ImageEnhance Module

You can use the ImageEnhance module in Pillow to adjust the contrast, color, sharpness, and brightness of an image just like you used to do it in old analog television sets.

Listing 5-8 shows the code for color adjustment.

Listing 5-8. prog10.py

```
from PIL import Image, ImageTk, ImageEnhance
import tkinter as tk

def show_value_1(factor):
    print('Color Factor: ', factor)
```

```
    enhancer = ImageEnhance.Color(im1)
    img = enhancer.enhance(float(factor))
    photo = ImageTk.PhotoImage(img)
    l['image'] = photo
    l.photo = photo

root = tk.Tk()
root.title('Color Adjustment Demo')

im1 = Image.open("/home/pi/DIP/Dataset/4.1.04.tiff")

photo = ImageTk.PhotoImage(im1)

l = tk.Label(root, image=photo)
l.pack()
l.photo = photo

w1 = (tk.Scale(root, label="Color Factor", from_=0, to=2,
      resolution=0.1, command=show_value_1, orient=tk.HORIZONTAL))
w1.pack()
w1.set(1)
root.mainloop()
```

In Listing 5-8, the image processing code is as follows:

```
enhancer = ImageEnhance.Color(im1)
img = enhancer.enhance(float(factor))
```

You can follow the same style of coding for all the other image enhancement operations. First, you create an enhancer and then you apply the enhancement factor to that. You also must have observed `w1.set(1)` in the code. This sets the scale to `1` at the beginning. Changing the argument to `set()` changes the default position of the scale pointer.

Run the program in Listing 5-8 and take a look at the output.

To change the contrast, use the code in Listing 5-9.

Listing 5-9. prog11.py

```
from PIL import Image, ImageTk, ImageEnhance
import tkinter as tk

def show_value_1(factor):
    print('Contrast Factor: ', factor)

    enhancer = ImageEnhance.Contrast(im1)
    img = enhancer.enhance(float(factor))
    photo = ImageTk.PhotoImage(img)
    l['image'] = photo
    l.photo = photo
```

```
root = tk.Tk()
root.title('Contrast Adjustment Demo')

im1 = Image.open("/home/pi/DIP/Dataset/4.1.04.tiff")

photo = ImageTk.PhotoImage(im1)

l = tk.Label(root, image=photo)
l.pack()
l.photo = photo

w1 = (tk.Scale(root, label="Contrast Factor", from_=0, to=2,
      resolution=0.1, command=show_value_1, orient=tk.HORIZONTAL))
w1.pack()
w1.set(1)
root.mainloop()
```

Run the program in Listing 5-9 and take a look at the output.
The following enhancer is used to change the brightness:

```
enhancer = ImageEnhance.Brightness(im1)
```

The following enhancer is used to change the sharpness:

```
enhancer = ImageEnhance.Sharpness(im1)
```

For finer control of the sharpness, use the following code for the scale:

```
w1 = (tk.Scale(root, label="Sharpness Factor", from_=0, to=2,
      resolution=0.1, command=show_value_1, orient=tk.HORIZONTAL))
```

These were all image enhancement operations. The next section looks at more advanced image operations.

Color Quantization

Color quantization is the process of reducing the number of distinct colors in an image. The new image should be similar to the original image in appearance. Color quantization is done for a variety of purposes, including when you want to store an image in a digital medium. Real-life images have millions of colors. However, encoding them in the digital format and retaining all the color related information would result in a huge image size. If you limit the number of colors in the image, you'll need less space to store the color-related information. This is the practical application of quantization. The `Image` module has a method called `quantize()` that's used for image quantization.

The code in Listing 5-10 demonstrates image quantization in Pillow.

Listing 5-10. prog14.py

```
from PIL import Image, ImageTk
import tkinter as tk

def show_value_1(num_of_col):
    print('Number of colors: ', num_of_col)

    img = im1.quantize(int(num_of_col))
    photo = ImageTk.PhotoImage(img)
    l['image'] = photo
    l.photo = photo

root = tk.Tk()
root.title('Color Quantization Demo')

im1 = Image.open("/home/pi/DIP/Dataset/4.1.06.tiff")

photo = ImageTk.PhotoImage(im1)

l = tk.Label(root, image=photo)
l.pack()
l.photo = photo

w1 = (tk.Scale(root, label="Number of colors", from_=4, to=16,
      resolution=1, command=show_value_1, orient=tk.HORIZONTAL))
w1.pack()
w1.set(256)
root.mainloop()
```

Run the program in Listing 5-10 and take a look at the results of quantization by changing the slider.

Histograms and Equalization

You likely studied frequency tables in statistics in school. Well, the histogram is nothing but a frequency table visualized. You can calculate a histogram of any dataset represented in the form of numbers.

The Image module has a method called `histogram()` that's used to calculate the histogram of an image. An RGB image has three 8-bit channels. This means that it can have a staggering **256x256x256** number of colors. Drawing a histogram of such a dataset would be very difficult. So, the `histogram()` method calculates the histogram of individual channels in an image. Each channel has 256 distinct intensities. The histogram is a list of values for each intensity level of a channel.

The histogram for each channel has 256 numbers in the list. Suppose the histogram for the Red channel has the values (4, 8, 0, 19, ..., 90). This means that four pixels have the red intensity of 0, eight pixels have the red intensity of 1, no pixel has red intensity of 2, 19 pixels have the red intensity of three, and so on, until the last value, which indicates that 90 pixels have the red intensity of 255.

When you combine the histogram of all three channels, you get a list of 768 numbers. In this chapter, we will just compute the histogram. We will not show it visually. When you learn about the advanced image processing library scipy.ndimage, you will then learn how to represent histograms for each channel individually.

The code in Listing 5-11 calculates and stores the histograms of an image and its individual channels.

Listing 5-11. prog15.py

```
from PIL import Image

im1 = Image.open("/home/pi/DIP/Dataset/4.2.07.tiff")

print(len(im1.histogram()))

r, g, b = im1.split()

print(len(r.histogram()))
print(len(g.histogram()))
print(len(b.histogram()))
```

Modify this program to directly print the histograms of the image and channels.

A grayscale image (`L` mode image) will have a histogram of only 256 values because it has only a single channel.

Histogram Equalization

You can adjust the histogram to enhance the image contrast. This technique is known as *histogram equalization*. The `ImageOps.equalize()` method equalizes the histogram of the image. Listing 5-12 shows an example of this process.

Listing 5-12. prog16.py

```
from PIL import Image, ImageOps

im1 = Image.open("/home/pi/DIP/Dataset/4.2.07.tiff")

print(im1.histogram())

im2 = ImageOps.equalize(im1)

print(im2.histogram())

im2.show()
```

The program in Listing 5-12 prints the histogram of the original image after the equalization. Add the `im1.show()` statement to the program and then run it to see the difference between the images.

EXERCISE

Complete the following exercises to gain a better understanding of advanced image processing methods in Pillow.

1. Write the code to demonstrate all the predefined filters defined at the beginning of the chapter.
2. Calculate and equalize the histogram for a grayscale (`L` mode) image.
3. Modify the code for convolution using the following kernel:

```
custom_filter = ImageFilter.Kernel((5, 5), [1,1,1,1,1,
1,1,1,1,1, 1,1,-10,1,1, 1,1,1,1,1, 1,1,1,1,1])
```

Conclusion

In this chapter, you explored the Pillow library for advanced image processing. Pillow is good for the beginners who want to get started with an easy-to-program and less mathematical image-processing library. However, if you want a more mathematical and scientific approach, then Pillow might not be your best choice. In the following chapters, you will learn about a more powerful library for image processing, scipy.ndimage. It's widely used by scientific community all over the world for image processing. You will also learn the basics of the NumPy and matplotlib libraries, which come in handy when processing and displaying images.