CHAPTER 8

Filters and Their Application

In the last chapter, you learned about the scipy.ndimage module of the SciPy library. You learned how to apply transformations like `shift()` and `zoom()` on an image. You also learned how to obtain statistical information about an image. You saw how to compute and plot the histogram for an image.

In Chapter 5, you studied image filters using the Pillow library. In this chapter, you will study the theory behind the filters in detail. You will also see the types of filters, kernels, and the applications of the filters in image processing.

Filters

Image filters are used to modify and/or enhance an image. All the image-filtering techniques you will study in this chapter are frequently used in image processing software programs like GIMP and Photoshop. Filtering is mostly a neighborhood operation. It is also called *local filtering*. For this, you need a matrix known as a *kernel*. A kernel is a two-dimensional matrix used to perform image processing. It is also known as a *filter*.

Let's look at a simple local filtering operation, the *convolution*. In the convolution operation, the filter (or the kernel) is applied to each pixel of the image in such a way that the center of the kernel is multiplied with the pixel. All the neighboring pixels of the pixel being processed are also multiplied by the corresponding element of the matrix. All the values are then added together to give an intensity value of the output pixel.

Note A kernel is always an odd matrix. For example, a size of a kernel could be (3, 3) or (5, 5). Bigger kernels need more time to process the image due to the number of computations involved.

Take a look at the simple example in Listing 8-1 for the convolution.

Listing 8-1. prog01.py

```
import scipy.misc as misc
import scipy.ndimage as ndi
import numpy as np
```


A. Pajankar, *Raspberry Pi Image Processing Programming*,
DOI 10.1007/978-1-4842-2731-2_8

```
import matplotlib.pyplot as plt

img = misc.lena()

k = np.array([[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]])

output = ndi.convolve(img, k)

plt.imshow(output, cmap='gray')
plt.title('Convolution')
plt.axis('off')
plt.show()
```

Run the program for yourself and take a look at the output. You will find that the output image is blurry. Convolution kernels can produce a variety of outputs and it is a vast topic in and of itself. Try to modify the values in the kernel and see the results for yourself.

■ **Note** Visit http://setosa.io/ev/image-kernels/ and https://docs.gimp.org/en/plug-in-convmatrix.html for more and detailed information on kernels.

Low-Pass Filters

Low-pass filters filter out high-frequency information. In other words, they allow information with a frequency lower than the cutoff frequency to pass through. They are usually called smoothing, blurring, or averaging filters in the image processing domain, as they are used to blur images or remove noise. A convolution filter is a basic low-pass filter. In this section, you will study low-pass filters and their applications.

Low-Pass Filters for Blurring

This section discusses how to use the low-pass filters for blurring images. You will use Gaussian and uniform filters for this purpose. The example in Listing 8-2 shows a Gaussian low-pass filter used to blur an image.

Listing 8-2. prog02.py

```
import scipy.misc as misc
import scipy.ndimage as ndi
import matplotlib.pyplot as plt
```

```
img = misc.lena()

output1 = ndi.gaussian_filter(img, sigma=3)
output2 = ndi.gaussian_filter(img, sigma=5)
output3 = ndi.gaussian_filter(img, sigma=7)

output = [output1, output2, output3]
titles = ['Sigma = 3', 'Sigma = 5', 'Sigma = 7']

for i in range(3):
        plt.subplot(1, 3, i+1)
        plt.imshow(output[i], cmap='gray')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

In the code in Listing 8-2, you are applying a Gaussian filter with standard deviation (sigma) values of 3, 5, and 7, respectively. You'll see a progressively blurred image, as shown in Figure 8-1.

Figure 8-1. *Gaussian filter*

You can also use the uniform filter to get this blurring/smoothing effect. Listing 8-3 shows an example of this effect.

Listing 8-3. prog03.py

```
import scipy.misc as misc
import scipy.ndimage as ndi
import matplotlib.pyplot as plt

img = misc.face()

output1 = ndi.uniform_filter(img, size=19)
output2 = ndi.uniform_filter(img, size=25)
```

```
output3 = ndi.uniform_filter(img, size=31)

output = [output1, output2, output3]
titles = ['Size = 19', 'Size = 25', 'Size = 31']

for i in range(3):
        plt.subplot(1, 3, i+1)
        plt.imshow(output[i], cmap='gray')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

In the code in Listing 8-3, you are applying the uniform filter of various sizes to the test image. The result is shown in Figure 8-2.

Figure 8-2. *Uniform filter*

Using Low-Pass Filters for Noise Removal

Another application of a low-pass filter is to remove noise from an image. You can define *noise* as any unwanted interference in a signal. All electronic devices are prone to noise. A small amount of noise is always present in all electronic signals, including electronically captured images. Noise is usually in the form of sharp and high-frequency data, which can be reduced by using low-pass filters like Gaussian and median filters. The example in Listing 8-4 demonstrates a Gaussian filter used to remove noise.

Listing 8-4. prog04.py

```
import scipy.misc as misc
import scipy.ndimage as ndi
import numpy as np
import matplotlib.pyplot as plt

img = misc.lena()

noisy = img + 0.8 * img.std() * np.random.random(img.shape)

output1 = ndi.gaussian_filter(noisy, sigma=1)
```

```
output2 = ndi.gaussian_filter(noisy, sigma=3)
output3 = ndi.gaussian_filter(noisy, sigma=5)

output = [noisy, output1, output2, output3]
titles = ['Noisy', 'Sigma = 1', 'Sigma = 3', 'Sigma = 5']

for i in range(4):
        plt.subplot(2, 2, i+1)
        plt.imshow(output[i], cmap='gray')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

In the code in Listing 8-4, the following line:

```
noisy = img + 0.8 * img.std() * np.random.random(img.shape)
```

is used to create the noisy image for the sample. As you are using the `random()` method to generate noise, the output image will be different every time the code is executed. The result is shown in Figure 8-3.

Figure 8-3. *Gaussian filter for noise removal*

Listing 8-5 shows an example of a median filter used to remove noise.

Listing 8-5. prog05.py

```
import scipy.misc as misc
import scipy.ndimage as ndi
import numpy as np
import matplotlib.pyplot as plt

img = misc.lena()

noisy = img + 0.8 * img.std() * np.random.random(img.shape)

output1 = ndi.median_filter(noisy, 3)
output2 = ndi.median_filter(noisy, 7)
output3 = ndi.median_filter(noisy, 9)

output = [noisy, output1, output2, output3]
titles = ['Noisy', 'Size = 1', 'Size = 3', 'Size = 5']

for i in range(4):
        plt.subplot(2, 2, i+1)
        plt.imshow(output[i], cmap='gray')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

This example uses the median filters of the size 1, 3, and 5, respectively. The result shown in Figure 8-4 demonstrates that the median filter is better at noise removal than the Gaussian filter.

Figure 8-4. *Median filter for noise removal*

High-Pass Filters

High-pass filters allow higher frequency information to pass through. In terms of image processing, the high-frequency components include edges and boundaries in an image. Thus, applying high-pass filters results in the edge highlighting and detection in an image. Let's look at a simple high-pass filter, the *prewitt* filter. A code example is shown in Listing 8-6.

Listing 8-6. prog06.py

```
import scipy.misc as misc
import scipy.ndimage as ndi
import matplotlib.pyplot as plt

img = misc.ascent()

filtered = ndi.prewitt(img)

output = [img, filtered]
titles = ['Original', 'Filtered']
```

```
for i in range(2):
        plt.subplot(1, 2, i+1)
        plt.imshow(output[i], cmap='gray')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

The application of this filter results in highlighted edges, as shown in Figure 8-5.

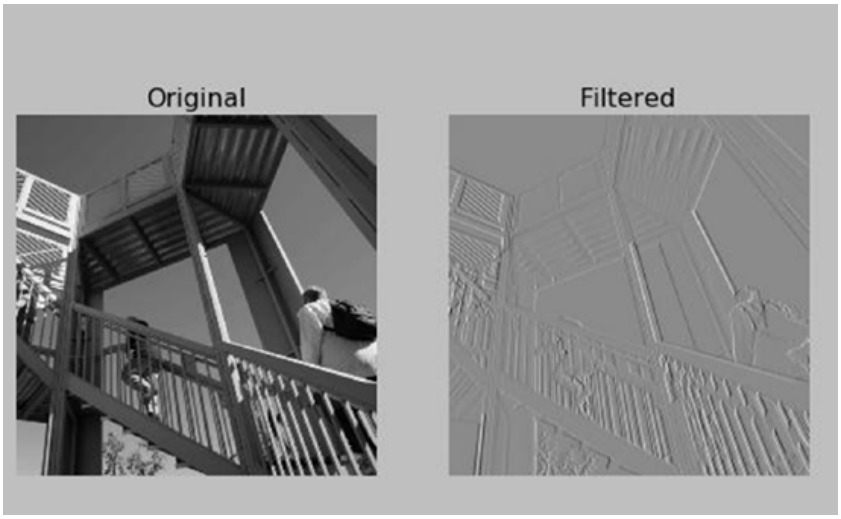

Figure 8-5. *Prewitt filter output*

There is a better high-pass filter, called the *Sobel* filter. As shown in Listing 8-7, it provides the horizontal and vertical edges separately.

Listing 8-7. prog07.py

```
import numpy as np
import scipy.ndimage as ndi
import matplotlib.pyplot as plt

img = np.zeros((516, 516))

img[128:-128, 128:-128] = 1

img = ndi.gaussian_filter(img, 8)

rotated = ndi.rotate(img, -20)

noisy = rotated + 0.09 * np.random.random(rotated.shape)

sx = ndi.sobel(noisy, axis=0)
sy = ndi.sobel(noisy, axis=1)
sob = np.hypot(sx, sy)
```

```
titles = ['Original', 'Rotated', 'Noisy',
          'Sobel (X-axis)', 'Sobel (Y-axis)', 'Sobel']

output = [img, rotated, noisy, sx, sy, sob]

for i in range(6):
        plt.subplot(2, 3, i+1)
        plt.imshow(output[i])
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

The code in Listing 8-7 generates a black square of dimensions 516x516 using the `np.zeros()` method. Then, the next line sets the intensity values of a 256x256 square within the generated square to white. Then it blurs and rotates the generated image. You then use this image as the source for applying the Sobel filters. First, you calculate the Sobel filter for the x axis, and then you calculate it for the y axis. `np.hypot()` is for calculation of the hypotenuse of a triangle. Passing the Sobel for the x and y axes to it gives you the combined results. When you calculate the Sobel for the x axis, it highlights the horizontal edges. When you calculate the Sobel for the y axis, it highlights the vertical edges. Combining the results returns the highlighted edges, as shown in Figure 8-6.

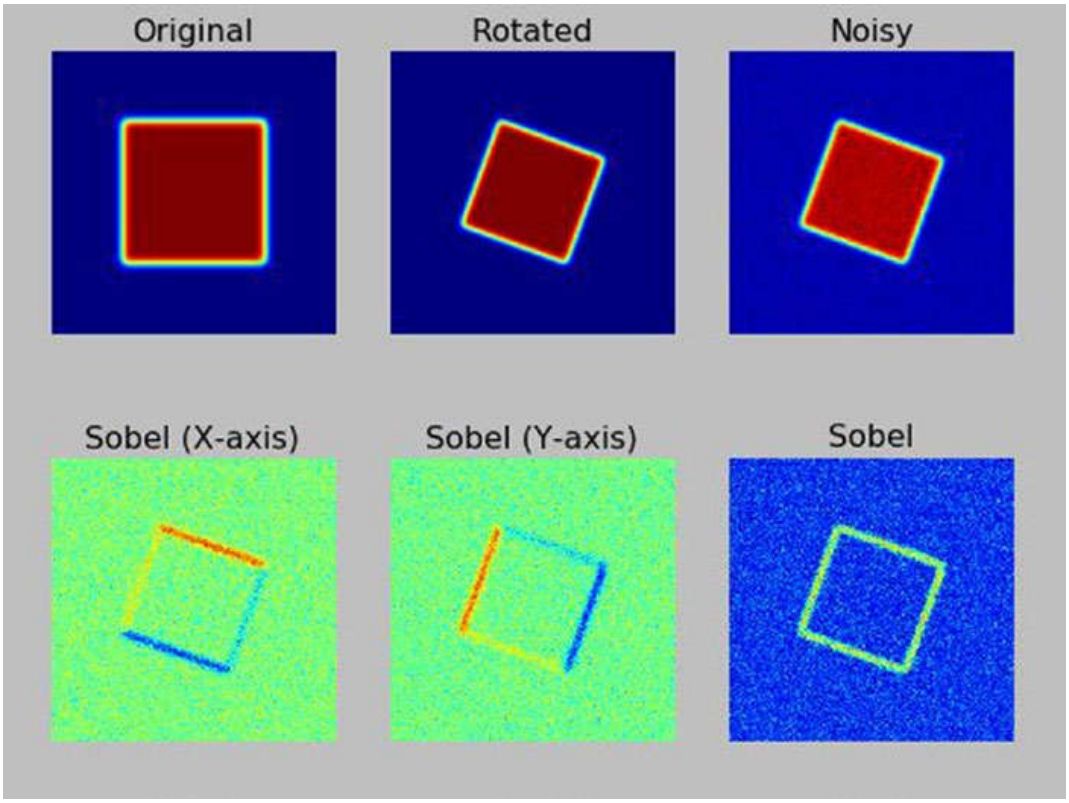

Figure 8-6. *Sobel filter demo*

Fourier Filters

The Fourier filter works on the frequency component of the signal. Fourier filters apply transformations in the frequency domain of the images. A Fourier filter first computes the Fourier transform of the image signals and then applies the required filtering function on the calculated Fourier transform of the image. Finally, it takes the inverse Fourier transform of the resultant output. Thus, it gives you the output on the frequency domain. It is used for a wide variety of image and signal processing tasks, including image filtering and reconstruction. Listing 8-8 shows an example of various Fourier transforms applied to an image.

Listing 8-8. prog08.py

```
import scipy.ndimage as ndi
import scipy.misc as misc
import matplotlib.pyplot as plt
import numpy as np

img = misc.ascent()

noisy = img + 0.09 * img.std() * np.random.random(img.shape)

fe = ndi.fourier_ellipsoid(img, 1)
fg = ndi.fourier_gaussian(img, 1)
fs = ndi.fourier_shift(img, 1)
fu = ndi.fourier_uniform(img, 1)

titles = ['Original', 'Noisy',
          'Fourier Ellipsoid', 'Fourier Gaussian',
          'Fourier Shift', 'Fourier Uniform']

output = [img, noisy, fe, fg, fs, fu]

for i in range(6):
        plt.subplot(2, 3, i+1)
        plt.imshow(np.float64(output[i]), cmap='gray')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

The result of the example in Listing 8-8 is shown in Figure 8-7.

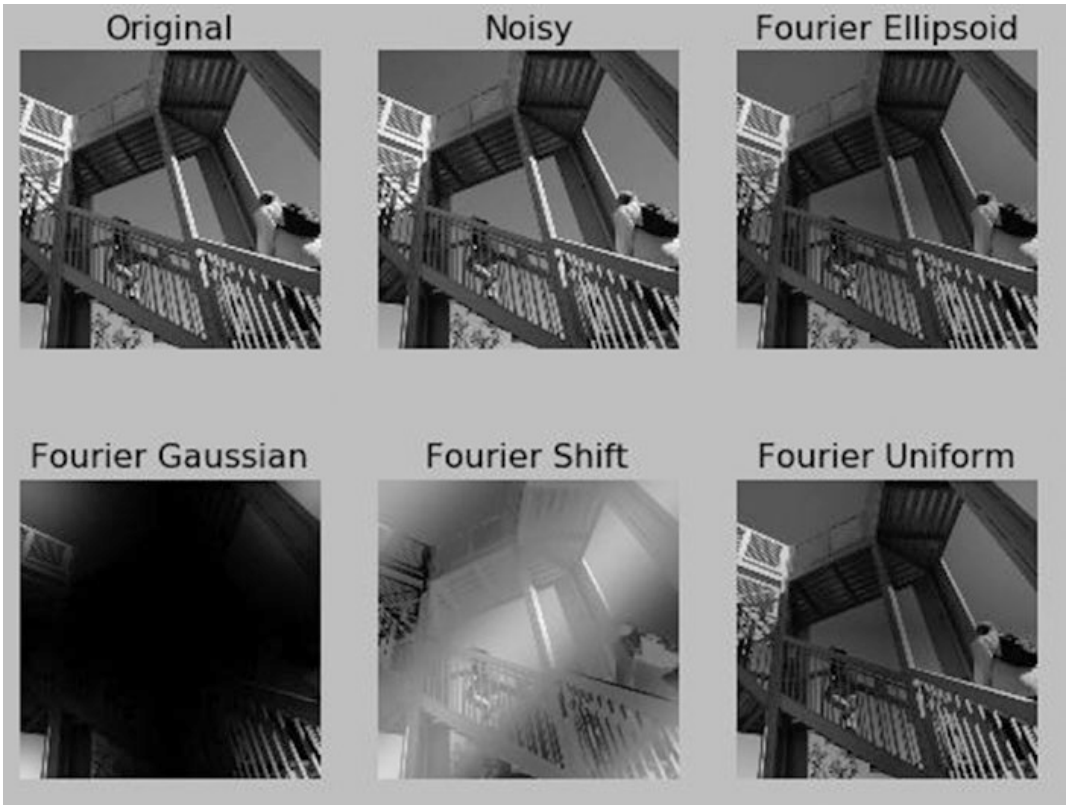

Figure 8-7. *Fourier filter demo*

■ **Note** Fourier filters are a vast area and it's difficult to cover them in a single chapter or section. You can find more information on Fourier filters at `https://terpconnect.umd.edu/~toh/spectrum/FourierFilter.html` and `http://homepages.inf.ed.ac.uk/rbf/HIPR2/fourier.htm`.

EXERCISE

This chapter introduced you to the concepts very briefly. However, image filtering and its applications are very complicated areas. Complete the following exercises to gain a better understanding of filters and their implementation in SciPy.

- Explore more filters in `ndimage`. You can find the documentation on https://docs.scipy.org/doc/scipy-0.18.1/reference/ndimage.html.
- Try different kernels on the convolution operation.
- Visit all the reference URLs mentioned in the chapter and explore more about the kernels and Fourier filters.
- Change the Sobel filter code example to show the results in grayscale using `cmap='gray'`.

Conclusion

In this chapter, you were introduced to the filters. You saw their types and looked at their applications according to the types. The image filtering area is too vast to be completely explored in a single chapter. Follow-up exercises will help you understand filtering better.

In the next chapter, we conclude the book with morphological operators, image thresholding, and basic segmentation.

CHAPTER 9

Morphology, Thresholding, and Segmentation

In the previous chapter, you studied the theory behind image filters, along with the types and practical applications of filters in enhancing images.

This is the last chapter in the book and a bit detailed one. In this chapter, you are going to study important concepts in image processing like morphology, morphological operations on images, thresholding, and segmentation. The chapter starts with the distance transform operation and then moves to the basics of morphology and structuring elements. You will see plenty of examples of morphological operations. We will then wrap up the chapter with thresholding and segmentation.

Distance Transforms

A *distance transform* is an operation on binary images. Binary images have background elements (zero value - black color) and foreground elements (white color). Distance transform replaces each foreground element with the value of the shortest distance to the background. scipy.ndimage has three methods for calculating the distance transform of a binary image. The code in Listing 9-1 illustrates how a distance transform can be used practically to generate test images.

Listing 9-1. prog01.py

```
import matplotlib.pyplot as plt
import scipy.ndimage as ndi
import numpy as np

img = np.zeros((32, 32))
img[8:-8, 8:-8] = 1

print(img)

dist1 = ndi.distance_transform_bf(img)
dist2 = ndi.distance_transform_cdt(img)
dist3 = ndi.distance_transform_edt(img)
```


A. Pajankar, *Raspberry Pi Image Processing Programming*,
DOI 10.1007/978-1-4842-2731-2_9

```
output = [img, dist1, dist2, dist3]
titles = ['Original', 'Brute Force', 'Chamfer', 'Euclidean']

for i in range(4):
        print(output[i])
        plt.subplot(2, 2, i+1)
        plt.imshow(output[i], interpolation='nearest', cmap='spectral')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

The code shown in Listing 9-1 calculates the distance transform by the brute force algorithm, Chamfer type algorithm, and Euclidean methods, respectively. You are going to use distance transforms in generating the test images in this chapter. The output is shown in Figure 9-1.

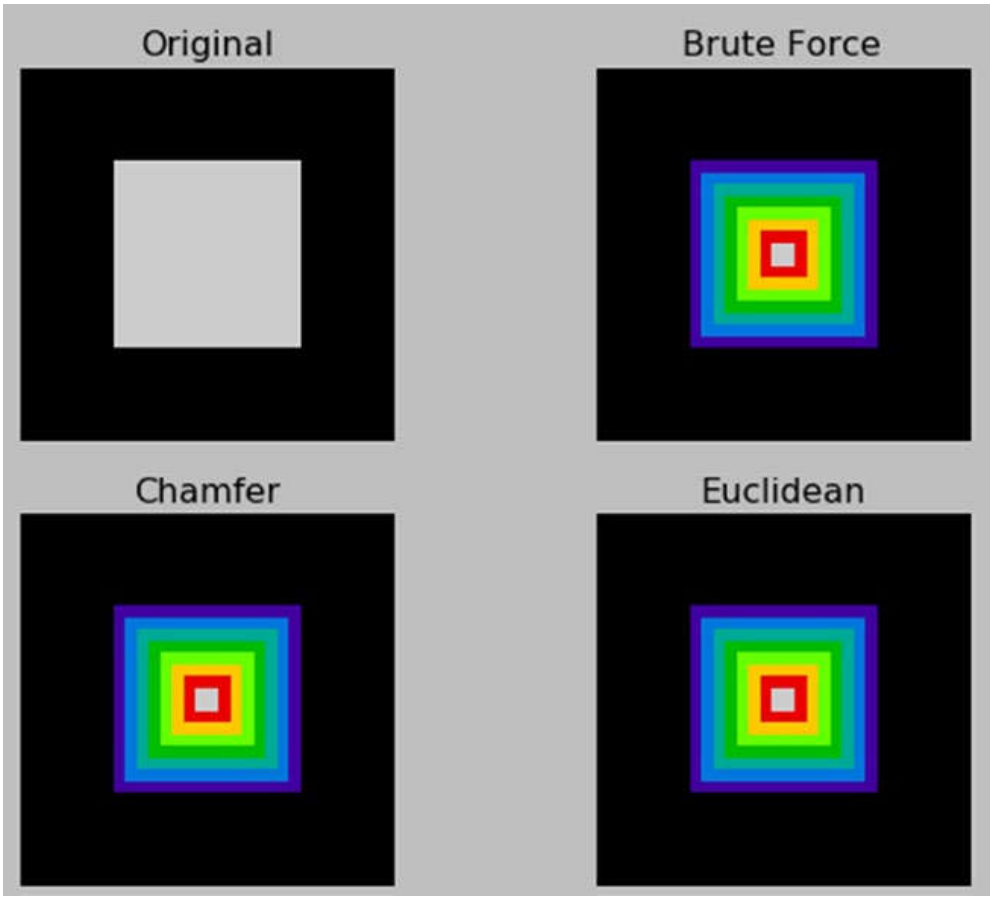

Figure 9-1. *Distance transforms demo*

Morphology and Morphological Operations

Morphology is the study of shapes and forms. The morphological study of images deals with shapes rather than the values of the pixels. Morphological operations are usually performed on binary images. Let's take a look at a few concepts related to the morphological operations.

Structuring Element

A structuring element is a matrix that's used to interact with a given binary image. It comes in various shapes, like a ball or a ring or a line. It can come in many shapes, like a 3x3 or 7x7 matrix. Bigger size structuring elements take more time for computation. A simple structuring element can be defined as a unity matrix of odd sizes. `np.ones((3,3))` is an example of this.

Various Morphological Operations

Let's briefly look at various morphological operations. Dilation causes the expansion of the shapes in an input image. Erosion causes the shrinkage in the shape in an input image. Opening is dilation of erosion. Closing is erosion of dilation.

It is difficult to understand these concepts just by reading about them. Take a look at the example in Listing 9-2, which demonstrates these concepts.

Listing 9-2. prog02.py

```
import matplotlib.pyplot as plt
import scipy.ndimage as ndi
import numpy as np

img = np.zeros((16, 16))
img[4:-4, 4:-4] = 1

print(img)

erosion = ndi.binary_erosion(img).astype(img.dtype)
dilation = ndi.binary_dilation(img).astype(img.dtype)
opening = ndi.binary_opening(img).astype(img.dtype)
closing = ndi.binary_closing(img).astype(img.dtype)

output = [img, erosion, dilation, opening, closing]
titles = ['Original', 'Erosion', 'Dilation', 'Opening', 'Closing']
```

```
for i in range(5):
        print(output[i])
        plt.subplot(1, 5, i+1)
        plt.imshow(output[i], interpolation='nearest', cmap='spectral')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

The code example in Listing 9-2 generates a binary image and applies all the binary morphological operations to it. The output is shown in Figure 9-2.

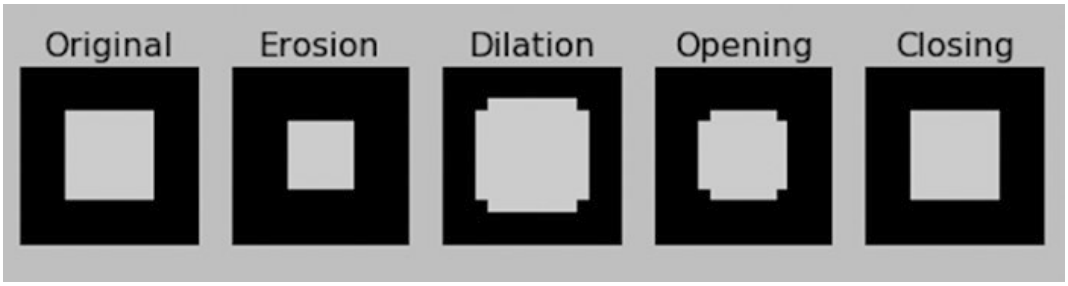

Figure 9-2. *Morphological operations demo*

Another important operation is `binary_fill_holes()`. It is used to fill the gaps in the binary image, as shown in Listing 9-3.

Listing 9-3. prog03.py

```
import matplotlib.pyplot as plt
import scipy.ndimage as ndi
import numpy as np

img = np.ones((32, 32))
x, y = (32*np.random.random((2, 20))).astype(np.int)
img[x, y] = 0

noise_removed = ndi.binary_fill_holes(img).astype(int)

output = [img, noise_removed]
titles = ['Original', 'Noise Removed']

for i in range(2):
        print(output[i])
        plt.subplot(1, 2, i+1)
        plt.imshow(output[i], interpolation='nearest', cmap='spectral')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

The code in Listing 9-3 generates a 32x32 square matrix image with all the values as `1` (white value). Then it randomly sets a few pixels to `0` (the dark value). The dark pixels can be considered holes in the image, which can then be removed using `binary_fill_holes()`.

The output is shown in Figure 9-3.

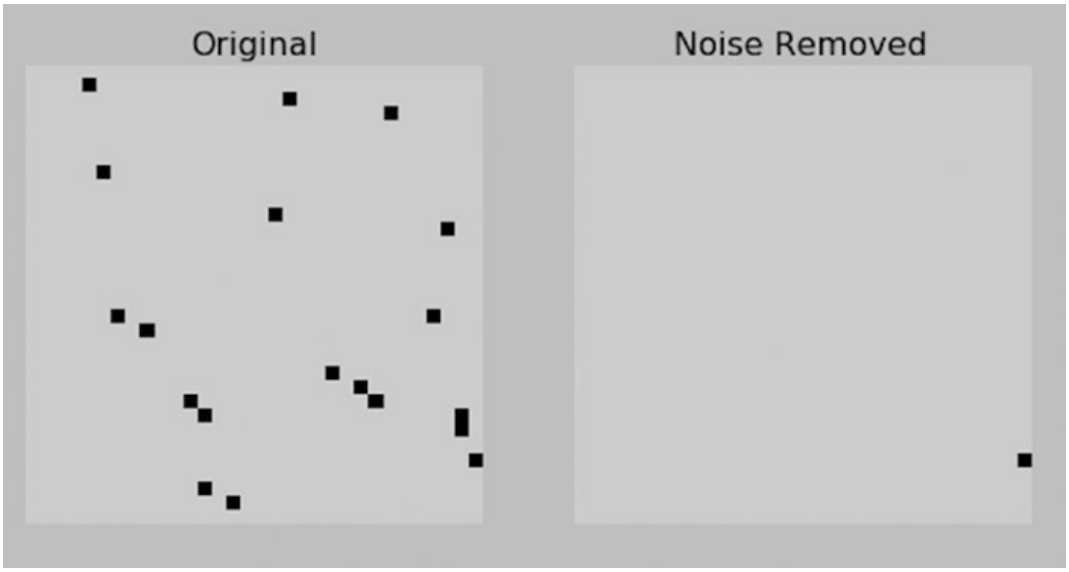

Figure 9-3. *The binary_fill_holes() demo*

Grayscale Morphological Operations

There is a set of grayscale morphological operations. The code shown in Listing 9-4 generates eight random values and assigns them to pixels on a completely dark (zero value) background. Then it uses the `grey_dilation()` method to dilate them.

Listing 9-4. prog04.py

```
import matplotlib.pyplot as plt
import scipy.ndimage as ndi
import numpy as np

img = np.zeros((64, 64))
x, y = (63*np.random.random((2, 8))).astype(np.int)
img[x, y] = np.arange(8)

dilation = ndi.grey_dilation(img, size=(5, 5),
                             structure=np.ones((5, 5)))

output = [img, dilation]
titles = ['Original', 'Dilation']
```

```
for i in range(2):
        print(output[i])
        plt.subplot(1, 2, i+1)
        plt.imshow(output[i], interpolation='nearest', cmap='spectral')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

The output is shown in Figure 9-4.

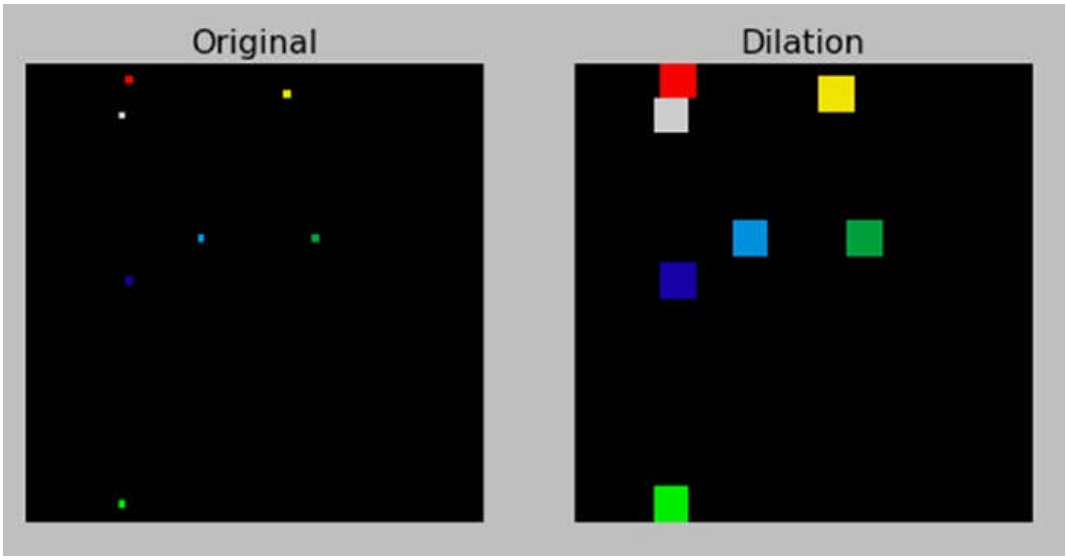

Figure 9-4. *Gray dilation demo*

You are using a structuring element that's 5x5 here.

■ **Note** This example uses the `random()` method. When you execute the code in the bundle, the output won't be exactly the same.

The code shown in Listing 9-5 applies gray dilation and gray erosion operations to a distance transform.

Listing 9-5. prog05.py

```
import matplotlib.pyplot as plt
import scipy.ndimage as ndi
import numpy as np

img = np.zeros((16, 16))
img[4:-4, 4:-4] = 1
```

```
img = ndi.distance_transform_bf(img)

dilation = ndi.grey_dilation(img, size=(3, 3),
                             structure=np.ones((3, 3)))

erosion = ndi.grey_erosion(img, size=(3, 3),
                           structure=np.ones((3, 3)))

output = [img, dilation, erosion]
titles = ['Original', 'Dilation', 'Erosion']

for i in range(3):
        print(output[i])
        plt.subplot(1, 3, i+1)
        plt.imshow(output[i], interpolation='nearest', cmap='spectral')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

The code in Listing 9-5 uses a structuring element that's 3x3 for both operations. The output is shown in Figure 9-5.

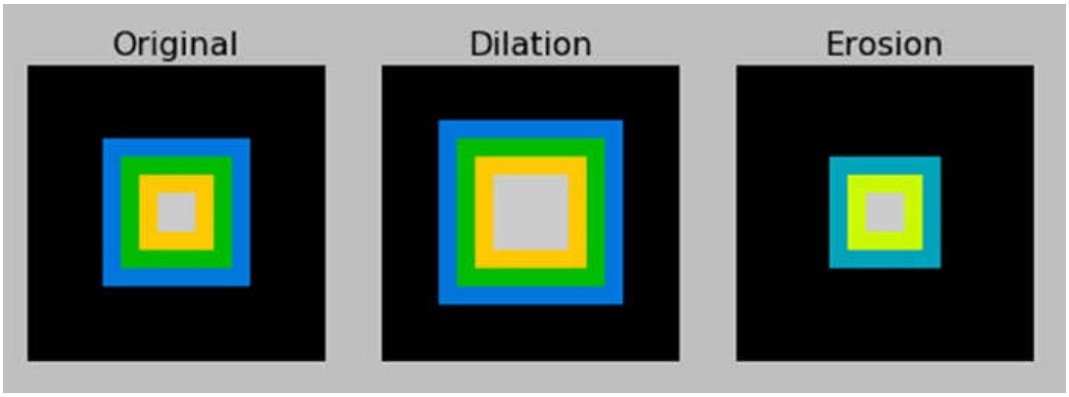

Figure 9-5. *Dilation and erosion on a distance transform*

Thresholding and Segmentation

This is the final part of the book and it deals with one of the most important applications of image processing, *segmentation*. In thresholding operations, you convert grayscale images to binary (black-and-white) images based on the threshold value. The pixels with intensity values greater than the threshold are assigned white and the pixels with an intensity value lower than the threshold are assigned a dark value. This is known as *binary thresholding* and it's the most basic form of thresholding and segmentation. An example is shown in Listing 9-6.

Listing 9-6. prog06.py

```
import matplotlib.pyplot as plt
import scipy.misc as misc

img = misc.ascent()

thresh = img > 127

output = [img, thresh]
titles = ['Original', 'Thresholding']

for i in range(2):
        plt.subplot(1, 2, i+1)
        plt.imshow(output[i], cmap='gray')
        plt.title(titles[i])
        plt.axis('off')

plt.show()
```

The code in Listing 9-6 sets the threshold to 127. At the grayscale, a pixel value of 127 corresponds to the gray color. The resultant thresholded image is shown in Figure 9-6.

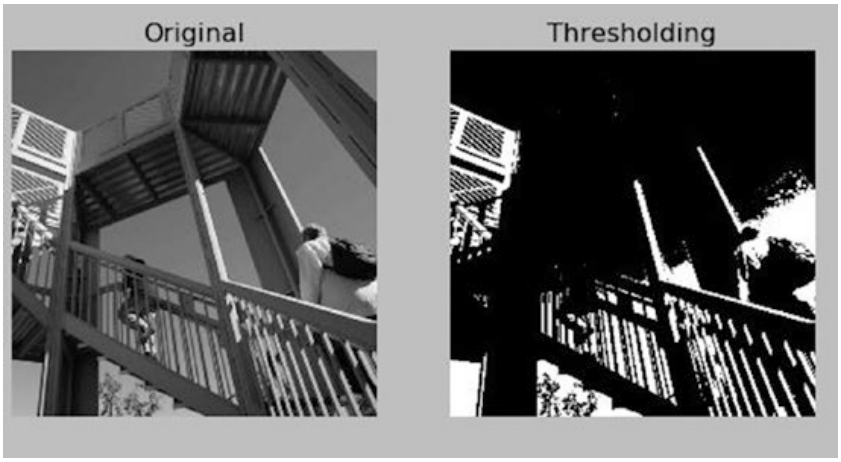

Figure 9-6. *Binary thresholding*

Thresholding is the most basic type of image segmentation. Image segmentation refers to dividing an image into many regions based on some property, like colors of pixels, connectivity of the region, etc. You can get the better segments of an image by applying the morphological operations on a thresholded image (see Listing 9-7).

Listing 9-7. prog07.py

```
import matplotlib.pyplot as plt
import scipy.misc as misc
import scipy.ndimage as ndi
import numpy as np

img = misc.ascent()

thresh = img > 127

dilated = ndi.binary_dilation(thresh, structure=np.ones((9, 9))).astype(int)

eroded = ndi.binary_erosion(dilated, structure=np.ones((9, 9))).astype(int)

output = [img, thresh, dilated, eroded]
titles = ['Original', 'Thresholding', 'Dilated', 'Eroded and Segmented']

for i in range(4):
        plt.subplot(2, 2, i+1)
        plt.imshow(output[i], cmap='gray')
        plt.title(titles[i])
        plt.axis('off')

plt.show()
```

The output is shown in Figure 9-7.

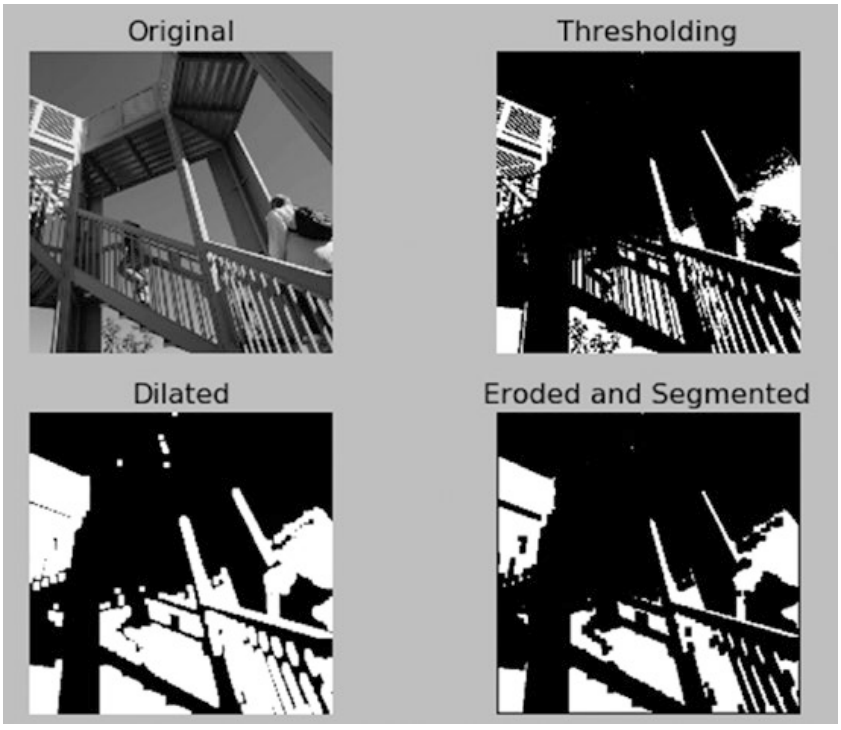

Figure 9-7. *Thresholded and segmented image*

Finally, here is yet another real-world application of segmentation, the processing of printed documents. The example in Listing 9-8 shows processed images of printed documents.

Listing 9-8. prog08.py

```
import matplotlib.pyplot as plt
import scipy.ndimage as ndi
import numpy as np

img = ndi.imread('/home/pi/DIP/Dataset/5.1.13.tiff')

thresh = img > 127

output = [img, thresh]
titles = ['Original', 'Thresholding']

for i in range(2):
        plt.subplot(1, 2, i+1)
        plt.imshow(output[i], cmap='gray')
        plt.title(titles[i])
        plt.axis('off')
plt.show()
```

The output is shown in Figure 9-8.

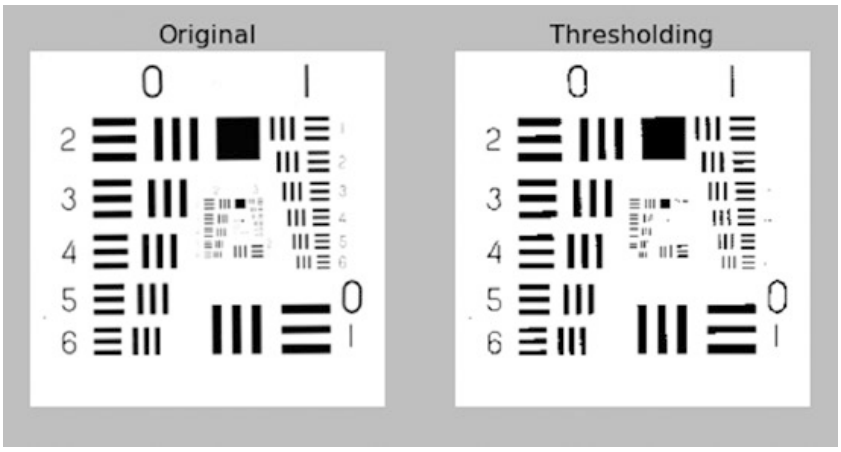

Figure 9-8. *Thresholding a printed document*

Conclusion

In this chapter, you studied distance transforms for generating test images. You then learned morphology and morphological operations on images. Morphological operations come in two varieties—binary and grayscale. You also studied thresholding, which is the simplest form of image segmentation. Finally, you studied how you can use thresholding on images in a document.

Book Summary

This is the final chapter of the book. You started with single board computers and Raspberry Pi. Then you learned how to set up the Pi and acquire images with various types of camera sensors. You took your first steps in the world of image processing with the help of Pillow. You also learned to create an interactive GUI for the image processing demos using Tkinter. Then you got started with scientific computing and the SciPy toolkit. You learned the basics of NumPy ndarrays and matplotlib for visualization. Finally, you learned to use scipy.misc and scipy.ndimage to process images.

I hope you enjoyed reading this book and following the examples as much as I enjoyed writing it and programming the examples.

What's Next

This is not an end but the mere beginning into the amazing world of image processing and computer vision. From here, you can follow various libraries for the image processing and data visualization. More advanced libraries for image processing include scikit-image, OpenCV, and Mahotas. You can explore these libraries. Consider also exploring NumPy and other aspects of scientific computing like signal processing and data visualization. The SciPy stack has suitable modules for all these applications. Happy exploring and Pythoning!