8

SMOOTHING AND BLURRING

I'm pretty sure we all know what blurring is. It's what happens when your camera takes a picture out of focus. Sharper regions in the image lose their detail, normally as a disc/circular shape.

Practically, this means that each pixel in the image is mixed in with its surrounding pixel intensities. This "mixture" of pixels in a neighborhood becomes our blurred pixel.

While this effect is usually unwanted in our photographs, it's actually quite helpful when performing image processing tasks.

In fact, many image processing and computer vision functions, such as thresholding and edge detection, perform better if the image is first smoothed or blurred.

In order to explore different types of blurring methods, let's start with a baseline of our original T-Rex image in Figure 8.1.

Listing 8.1: blurring.py

Figure 8.1: Our original T-Rex image before applying any blurring effects.

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
```

In order to perform image blurring, we first need to import our packages and parse our arguments (**Lines 1-8**). We then load our image and show it as a baseline to compare our blurring methods to on **Line 10 and 11**.

Now that our image is loaded, we can start blurring our images.

8.1 AVERAGING

The first blurring method we are going to explore is averaging.

As the name suggests, we are going to define a $k \times k$ sliding window on top of our image, where k is always an odd number. This window is going to slide from left-to-right and from top-to-bottom. The pixel at the center of this matrix (and hence why we have to use an odd number, otherwise there would not be a true "center") is then set to be the *average* of all other pixels surrounding it.

We call this sliding window a "convolution kernel" or just a "kernel". We'll continue to use this terminology throughout this chapter.

As we will see, as the size of the kernel increases, the more blurred our image will become.

Let's check out some code to perform average blurring:

Listing 8.2: blurring.py

```
12 blurred = np.hstack([
13     cv2.blur(image, (3, 3)),
14     cv2.blur(image, (5, 5)),
15     cv2.blur(image, (7, 7))])
16 cv2.imshow("Averaged", blurred)
17 cv2.waitKey(0)
```

Figure 8.2: Performing averaging blurring with a 3×3 kernel (*left*), 5×5 kernel (*middle*), and 7×7 kernel (*right*).

In order to average blur an image, we use the `cv2.blur` function. This function requires two arguments: the image we want to blur and the size of the kernel. As lines **13-15** show, we blur our image with increasing sizes kernels. The larger our kernel becomes, the more blurred our image will appear.

We make use of the `np.hstack` function to stack our output images together. This method "horizontally stacks" our three images into a row. This is useful since we don't want to create three separate windows using the `cv2.imshow` function.

The output of our averaged blur can be seen in Figure 8.2. The image on the left is barely blurred, but by the time we reach a kernel of size 7×7, we see that our T-Rex is very blurry indeed. Perhaps he was running at a high speed and chasing a jeep?

Figure 8.3: Performing Gaussian blurring with a 3×3 kernel (*left*), 5×5 kernel (*middle*), and 7×7 kernel (*right*). Again, our image becomes more blurred as the kernel size increases, but is less blurred than the average method in Figure 8.2.

8.2 GAUSSIAN

Next up, we are going to review Gaussian blurring. Gaussian blurring is similar to average blurring, but instead of using a simple mean, we are now using a weighted mean, where neighborhood pixels that are closer to the central pixel contribute more "weight" to the average.

The end result is that our image is less blurred, but more naturally blurred, than using the average method discussed in the previous section.

Let's look at some code to perform Gaussian blurring:

Listing 8.3: blurring.py

```
18 blurred = np.hstack([
19     cv2.GaussianBlur(image, (3, 3), 0),
```

```
20      cv2.GaussianBlur(image, (5, 5), 0),
21      cv2.GaussianBlur(image, (7, 7), 0)])
22  cv2.imshow("Gaussian", blurred)
23  cv2.waitKey(0)
```

Here you can see that we are making use of the `cv2.GaussianBlur` function on **Lines 19-21**. The first argument to the function the image we want to blur. Then, similar to `cv2.blur`, we provide a tuple representing our kernel size. Again, we start with a small kernel size of 3×3 and start to increase it.

The last parameter is our σ, the standard deviation in the x-axis direction. By setting this value to 0, we are instructing OpenCV to automatically compute them based on our kernel size.

We can see the output of our Gaussian blur in Figure 8.3. Our images have less of a blur effect than when using the averaging method in Figure 8.2; however, the blur itself is more natural, due to the computation of the weighted mean, rather than allowing all pixels in the kernel neighborhood to have equal weight.

8.3 MEDIAN

Traditionally, the median blur method has been most effective when removing salt-and-pepper noise. This type of noise is exactly what it sounds like: imagine taking a photograph, putting it on your dining room table, and sprinkling salt and pepper on top of it. Using the median blur method, you could remove the salt and pepper from your image.

When applying a median blur, we first define our kernel size k. Then, as in the averaging blurring method, we consider all pixels in the neighborhood of size $k \times k$. But, unlike the averaging method, instead of replacing the central pixel with the average of the neighborhood, we instead replace the central pixel with the median of the neighborhood.

The reason median blurring is more effective at removing salt-and-pepper style noise from an image is that each central pixel is always replaced with a pixel intensity that exists in the image.

Methods such as averaging and Gaussian compute means or weighted means for the neighborhood – this average pixel intensity may or may not be present in the neighborhood. But by definition, the median pixel *must* exist in our neighborhood. By replacing our central pixel with a median rather than an average, we can substantially reduce noise.

Now, it's time to apply our median blur:

Listing 8.4: blurring.py

```
24 blurred = np.hstack([
25     cv2.medianBlur(image, 3),
26     cv2.medianBlur(image, 5),
27     cv2.medianBlur(image, 7)])
28 cv2.imshow("Median", blurred)
29 cv2.waitKey(0)
```

Applying a median blur is accomplished by making a call to the `cv2.medianBlur` function. This method takes two parameters: the image we want to blur and the size of our kernel. On **Lines 25-27**, we start off with a kernel size of

Figure 8.4: Applying the median blur method to our T-Rex image with increasing kernel sizes of 3 (*left*), 5 (*middle*), and 7 (*right*), respectively. Notice that we are no longer creating a "motion blur".

3, then increase it to 5 and 7. The resulting blurred images are then stacked and displayed to us.

Our median blurred images can be seen in Figure 8.4. Notice that we are no longer creating a "motion blur" effect like in averaging and Gaussian blurring – instead, we are removing detail and noise.

For example, take a look at the color of the scales of the T-Rex. As our kernel size increases, the scales become less pronounced. The black and brown stripes on the legs and tail of the T-Rex especially lose their detail, all without creating a motion blur.

8.4 BILATERAL

The last method we are going to explore is bilateral blurring.

Thus far, the intention of our blurring methods have been to reduce noise and detail in an image; however, we tend to lose edges in the image.

In order to reduce noise while still maintaining edges, we can use bilateral blurring. Bilateral blurring accomplishes this by introducing two Gaussian distributions.

The first Gaussian function only considers spatial neighbors, that is, pixels that appear close together in the (x, y) coordinate space of the image. The second Gaussian then models the pixel intensity of the neighborhood, ensuring that only pixels with similar intensity are included in the actual computation of the blur.

Overall, this method is able to preserve edges of an image, while still reducing noise. The largest downside to this method is that it is considerably slower than its averaging, Gaussian, and median blurring counterparts.

Let's look at some code:

Listing 8.5: blurring.py

```
30 blurred = np.hstack([
31     cv2.bilateralFilter(image, 5, 21, 21),
32     cv2.bilateralFilter(image, 7, 31, 31),
33     cv2.bilateralFilter(image, 9, 41, 41)])
34 cv2.imshow("Bilateral", blurred)
35 cv2.waitKey(0)
```

Figure 8.5: Applying Bilateral blurring to our beach image. As the diameter of the neighborhood, color σ, and space σ increases (from left to right), our image has noised removed, yet still retains edges and does not appear to be "motion blurred".

We apply bilateral blurring by call the `cv2.bilateralFilter` function on **Lines 31-33**. The first parameter we supply is the image we want to blur. Then, we need to define the diameter of our pixel neighborhood. The third argument is our color σ. A larger value for color σ means that more colors in the neighborhood will be considered when computing the blur. Finally, we need to supply the space σ. A larger value of space σ means that pixels farther out from the central pixel will influence the blurring calculation, provided that their colors are similar enough.

We obtain results using for increasing neighborhood sizes, color σ, and space σ. These results can be seen in Figure 8.5. As the size of our parameters increases, our image has noise removed, yet the edges still remain.

Now that we know how to blur our images, we can move on to thresholding in the next chapter. You can be sure that we'll make use of blurring throughout the rest of this book!

9

THRESHOLDING

Thresholding is the binarization of an image. In general, we seek to convert a grayscale image to a binary image, where the pixels are either 0 or 255.

A simple thresholding example would be selecting a pixel value p, and then setting all pixel intensities less than p to zero, and all pixel values greater than p to 255. In this way, we are able to create a binary representation of the image.

Normally, we use thresholding to focus on objects or areas of particular interest in an image. In the examples in the sections below, we will empty out our pockets and look at our spare change. Using thresholding methods, we'll be able to find the coins in an image.

9.1 SIMPLE THRESHOLDING

Applying simple thresholding methods requires human intervention. We must specify a threshold value T. All pixel intensities below T are set to 0. And all pixel intensities greater than T are set to 255.

We could also apply the inverse of this binarization by setting all pixels below T to 255 and all pixel intensities greater than T to 0.

Let's explore some code to apply simple thresholding methods:

Listing 9.1: simple_thresholding.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 blurred = cv2.GaussianBlur(image, (5, 5), 0)
13 cv2.imshow("Image", image)
```

On **Lines 1-10** we import our packages, parse our arguments, and load our image. From there, we convert the image from the RGB color space to grayscale on **Line 11**.

At this point, we apply Gaussian blurring on **Line 12** with a $\sigma = 5$ radius. Applying Gaussian blurring helps remove some of the high frequency edges in the image that we are not concerned with.

Listing 9.2: simple_thresholding.py

```
14 (T, thresh) = cv2.threshold(blurred, 155, 255, cv2.THRESH_BINARY)
15 cv2.imshow("Threshold Binary", thresh)
16
17 (T, threshInv) = cv2.threshold(blurred, 155, 255, cv2.
       THRESH_BINARY_INV)
```

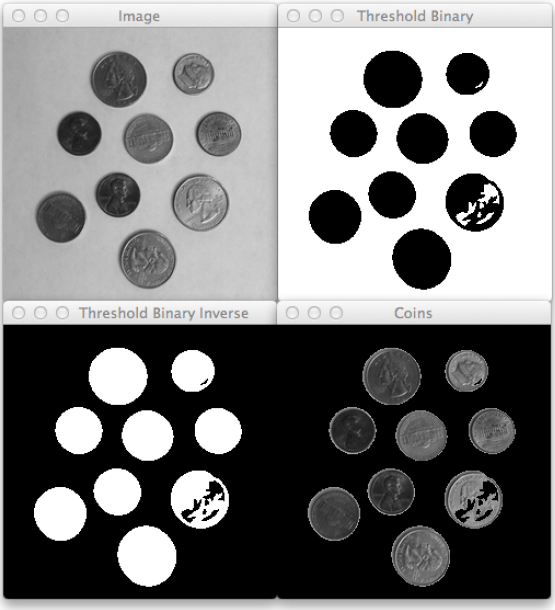

Figure 9.1: *Top-Left:* The original coins image in grayscale. *Top-Right:* Applying simple binary thresholding. The coins are shown in black and the background in white. *Bottom-Left:* Applying inverse binary thresholding. The coins are now white and the background is black. *Bottom-Right:* Applying the inverse binary threshold as a mask to the grayscale image. We are now focused on only the coins in the image.

```
18 cv2.imshow("Threshold Binary Inverse", threshInv)
19
20 cv2.imshow("Coins", cv2.bitwise_and(image, image, mask =
       threshInv))
21 cv2.waitKey(0)
```

After the image is blurred, we compute the thresholded image on **Line 14** using the `cv2.threshold` function. This method requires four arguments. The first is the grayscale image that we wish to threshold. We supply our blurred image here.

Then, we manually supply our T threshold value. We use a value of $T = 155$.

Our third argument is our maximum value applied during thresholding. Any pixel intensity p that is greater than T, is set to this value. In our example, any pixel value that is greater than 155 is set to 255. Any value that is less than 155 is set to zero.

Finally, we must provide a thresholding method. We use the `cv2.THRESH_BINARY` method, which indicates that pixel values p greater than T are set to the maximum value (the third argument).

The `cv2.threshold` function returns two values. The first is T, the value we manually specified for thresholding. The second is our actual thresholded image.

We then show our thresholded image in Figure 9.1, *Top-Right*. We can see that our coins are now black pixels and the white pixels are the background.

On **Line 17** we apply inverse thresholding rather than normal thresholding by using `cv2.THRESH_BINARY_INV` as our thresholding method. As we can see in Figure 9.1, *Bottom-Left*, our coins are now white and the background is black. This is convenient as we will see in a second.

The last task we are going to perform is to reveal the coins in the image and hide everything else.

Remember when we discussed masking? That will come in handy here.

On **Line 20** we perform masking by using the `cv2.bitwise_and` function. We supply our original coin image as the first two arguments, and then our inverted thresholded image as our mask. Remember, a mask only consider pixels in the original image where the mask is greater than zero. Since our inverted thresholded image on **Line 17** does a good job at approximating the areas coins are contained in, we can use this inverted thresholded image as our mask.

Figure 9.1 *Bottom-Right* shows the result of applying our mask – the coins are clearly revealed while the rest of the image is hidden.

9.2 ADAPTIVE THRESHOLDING

One of the downsides of using simple thresholding methods is that we need to manually supply our threshold value T. Not only does finding a good value of T require a lot of manual experiments and parameter tunings, it's not very

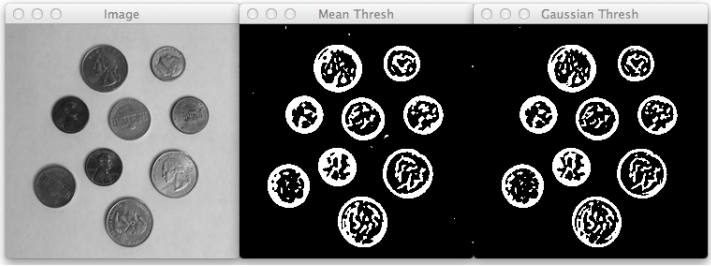

Figure 9.2: *Left:* The grayscale coins image. *Middle:* Applying adaptive thresholding using mean neighborhood values. *Right:* Applying adaptive thresholding using Gaussian neighborhood values.

helpful if the image exhibits a lot of range in pixel intensities.

Simply put, having just one value of T might not suffice.

In order to overcome this problem, we can use adaptive thresholding, which considers small neighbors of pixels and then finds an optimal threshold value T for each neighbor. This method allows us to handle cases where there may be dramatic ranges of pixel intensities and the optimal value of T may change for different parts of the image.

Let's go ahead and jump into some code that applies adaptive thresholding:

Listing 9.3: adaptive_thresholding.py

```
1  import numpy as np
2  import argparse
3  import cv2
4
5  ap = argparse.ArgumentParser()
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 blurred = cv2.GaussianBlur(image, (5, 5), 0)
13 cv2.imshow("Image", image)
14
15 thresh = cv2.adaptiveThreshold(blurred, 255,
16     cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 11, 4)
17 cv2.imshow("Mean Thresh", thresh)
18
19 thresh = cv2.adaptiveThreshold(blurred, 255,
20     cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 15, 3)
21 cv2.imshow("Gaussian Thresh", thresh)
22 cv2.waitKey(0)
```

Lines 1-10 once again handle setting up our example. We import our packages, construct our argument parser, and load the image. Just as in our simple thresholding example above, we then convert the image to grayscale and blur it slightly on **Lines 11 and 12**.

We then apply adaptive thresholding to our blurred image using the `cv2.adaptiveThreshold` function on **Line 15**. The first parameter we supply is the image we want to threshold. Then, we supply our maximum value of 255, similar to simple thresholding mentioned above.

The third argument is our method to compute the threshold for the current neighborhood of pixels. By supplying `cv2.ADAPTIVE_THRESH_MEAN_C` we indicate that we want to compute the mean of the neighborhood of pixels and treat

it as our T value.

Fourthly, we need our thresholding method. Again, the description of this parameter is identical to the simple thresholding method mentioned above. We use `cv2.THRESH_BINARY_INV` to indicate that any pixel intensity greater than T in the neighborhood should be set to 255, otherwise it should be set to 0.

The next parameter is our neighborhood size. This integer value must be odd and indicates how large our neighborhood of pixels is going to be. We supply a value of 11, indicating that we are going to examine 11×11 pixel regions of the image, instead of trying to threshold the image globally, as in simple thresholding methods.

Finally, we supply a parameter simply called C. This value is an integer that is subtracted from the mean, allowing us to fine tune our thresholding. We use $C = 4$ in this example.

The results of applying mean weighted adaptive thresholding can be seen in the *middle* image of Figure 9.2.

Besides applying standard mean thresholding, we can also apply Gaussian (weighted mean) thresholding, as we do on **Line 19**. The order of the parameters are identical to **Line 15**, but now we are tuning a few of the values.

Instead of supplying a value of `cv2.ADAPTIVE_THRESH_MEAN_C`, we instead use `cv2.ADAPTIVE_THRESH_GAUSSIAN_C` to indicate we want to use the weighted mean. We are also using a 15×15 pixel neighborhood size rather than

an 11×11 neighborhood size as in the previous example. We also alter our C value (the value we subtract from the mean) slightly and use 3 rather than 4.

The results of applying Gaussian adaptive thresholding can be seen in the *right* image of Figure 9.2. There is little difference between the two images.

In general, choosing between mean adaptive thresholding and Gaussian adaptive thresholding requires a few experiments on your end. The most important parameters to vary are the neighborhood size and C, the value you subtract from the mean. By experimenting with this values, you will be able to dramatically change the results of your thresholding.

9.3 OTSU AND RIDDLER-CALVARD

Another way we can automatically compute the threshold value of T is to use Otsu's method.

Otsu's method assumes there are two peaks in the grayscale histogram of the image. In then tries to find an optimal value to separate these two peaks – thus our value of T.

While OpenCV provides support for Otsu's method, I prefer the implementation by Luis Pedro Coelho in the `mahotas` package since it is more Pythonic.

Let's jump into some example code:

Listing 9.4: otsu_and_riddler.py

```
1 import numpy as np
2 import argparse
3 import mahotas
4 import cv2
5
6 ap = argparse.ArgumentParser()
7 ap.add_argument("-i", "--image", required = True,
8     help = "Path to the image")
9 args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
13 blurred = cv2.GaussianBlur(image, (5, 5), 0)
14 cv2.imshow("Image", image)
15
16 T = mahotas.thresholding.otsu(blurred)
17 print "Otsu's threshold: %d" % (T)
```

On **Lines 1-4** we import the packages we will utilize. We have seen `numpy`, `argparse`, and `cv2` before. We are now introducing `mahotas`, another image processing package.

Lines 6-11 then handle our standard practice of parsing arguments and loading our image.

As in previous thresholding examples, we convert the image to grayscale and then blur it slightly.

To compute our optimal value of T, we use the `otsu` function in the `mahotas.thresholding` package. As our output will later show us, Otsu's method finds a value of $T = 137$ that we will use for thresholding.

Listing 9.5: otsu_and_riddler.py

```
18 thresh = image.copy()
19 thresh[thresh > T] = 255
20 thresh[thresh < 255] = 0
```

```
21 thresh = cv2.bitwise_not(thresh)
22 cv2.imshow("Otsu", thresh)
23
24 T = mahotas.thresholding.rc(blurred)
25 print "Riddler-Calvard: %d" % (T)
26 thresh = image.copy()
27 thresh[thresh > T] = 255
28 thresh[thresh < 255] = 0
29 thresh = cv2.bitwise_not(thresh)
30 cv2.imshow("Riddler-Calvard", thresh)
31 cv2.waitKey(0)
```

Applying the thresholding is accomplished on **Lines 18-21**. First, we make a copy of our grayscale image so that we have an image to threshold. **Line 19** then makes any values greater than T white, whereas **Line 20** makes all remaining pixels that are not white into black pixels. We then invert our threshold by using `cv2.bitwise_not`. This is equivalent to applying a `cv2.THRESH_BINARY_INV` thresholding type as in previous examples in this chapter.

The results of Otsu's method can be seen in the *middle* image of Figure 9.3. We can clearly see that the coins in the image have been highlighted.

Another method to keep in mind when finding optimal values for T is the Riddler-Calvard method. Just as in Otsu's method, the Riddler-Calvard method also computes an optimal value of 137 for T. We apply this method on **Line 24** using the `rc` function in `mahotas.thresholding`. Finally, the actual thresholding of the image takes place on **Lines 26-29**, as in the previous example. Given that the values of T are identical for Otsu and Riddler-Calvard, the thresholded image in Figure 9.3 (*right*) is identical to the thresholded image in the center.

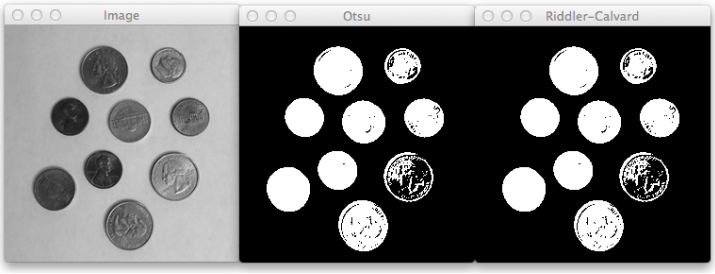

Figure 9.3: *Left:* The original grayscale coins image. *Middle:* Applying Otsu's method to find an optimal value of T. *Right:* Applying the Riddler-Calvard method to find an optimal value of T.

Listing 9.6: otsu_and_riddler.py

```
Otsu's threshold: 137
Riddler-Calvard: 137
```

Now that we have explored thresholding, we will move on to another powerful image processing technique – edge detection.

10

GRADIENTS AND EDGE DETECTION

This chapter is primarily concerned with gradients and edge detection. Formally, edge detection embodies mathematical methods to find points in an image where the brightness of pixel intensities changes distinctly.

The first thing we are going to do is find the "gradient" of the grayscale image, allowing us to find edge like regions in the x and y direction.

We'll then apply Canny edge detection, a multi-stage process of noise reduction (blurring), finding the gradient of the image (utilizing the Sobel kernel in both the horizontal and vertical direction), non-maximum suppression, and hysteresis thresholding.

If that sounds like a mouthful, it's because it is. Again, we won't jump too far into the details since this book is concerned with practical examples of computer vision; however, if you are interested in the mathematics behind gradients and edge detection, I encourage you to read up on the algorithms. Overall, they are not complicated and can be

Figure 10.1: *Left:* The original coins image. *Right:* Applying the Laplacian method to obtain the gradient of the image.

insightful to the behind the scenes action of OpenCV.

10.1 LAPLACIAN AND SOBEL

Let's go ahead and explore some code:

Listing 10.1: sobel_and_laplacian.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 cv2.imshow("Original", image)
```

```
13
14 lap = cv2.Laplacian(image, cv2.CV_64F)
15 lap = np.uint8(np.absolute(lap))
16 cv2.imshow("Laplacian", lap)
17 cv2.waitKey(0)
```

Lines 1-8 import our packages and setup our argument parser. From, there we load our image and convert it to grayscale on **Lines 10 and 11**. When computing gradients and edges we (normally) compute them on a single channel – in this case, we are using the grayscale image; however, we could also compute gradients for each channel of the RGB image. For the sake of simplicity, let's stick with the grayscale image since that is what you will use in most cases.

On **Line 14** we use the Laplacian method to compute the gradient magnitude image by calling the `cv2.Laplacian` function. The first argument is our grayscale image – the image we want to compute the gradient magnitude representation for. The second argument is our data type for the output image.

Throughout this book, we have mainly used 8-bit unsigned integers. Why are we using a 64-bit float now?

The reason involves the transition of black-to-white and white-to-black in the image.

Transitioning from black-to-white is considered a positive slope, whereas a transition from white-to-black is a negative slope. If you remember back to our discussion of image arithmetic in Chapter 6, you'll know that an 8-bit unsigned integer does not represent negative values. Either it will be clipped to zero if you are using OpenCV or a mod-

ulus operation will be performed using NumPy.

The short answer here is that if you don't use a floating point data type when computing the gradient magnitude image, you will miss edges, specifically the white-to-black transitions.

In order to ensure you catch all edges, use a floating point data type, then take the absolute value of the gradient image and convert it back to an 8-bit unsigned integer, as in **Line 15**. This is definitely an important technique to take note of – otherwise you'll be missing edges in your image!

To see the results of our gradient processing, take a look at Figure 10.1.

Let's move on to computing the Sobel gradient representation:

Listing 10.2: sobel_and_laplacian.py

```
18 sobelX = cv2.Sobel(image, cv2.CV_64F, 1, 0)
19 sobelY = cv2.Sobel(image, cv2.CV_64F, 0, 1)
20
21 sobelX = np.uint8(np.absolute(sobelX))
22 sobelY = np.uint8(np.absolute(sobelY))
23
24 sobelCombined = cv2.bitwise_or(sobelX, sobelY)
25
26 cv2.imshow("Sobel X", sobelX)
27 cv2.imshow("Sobel Y", sobelY)
28 cv2.imshow("Sobel Combined", sobelCombined)
```

Using the Sobel operator, we can compute gradient magnitude representations along the x and y axis, allowing us to find both horizontal and vertical edge-like regions.

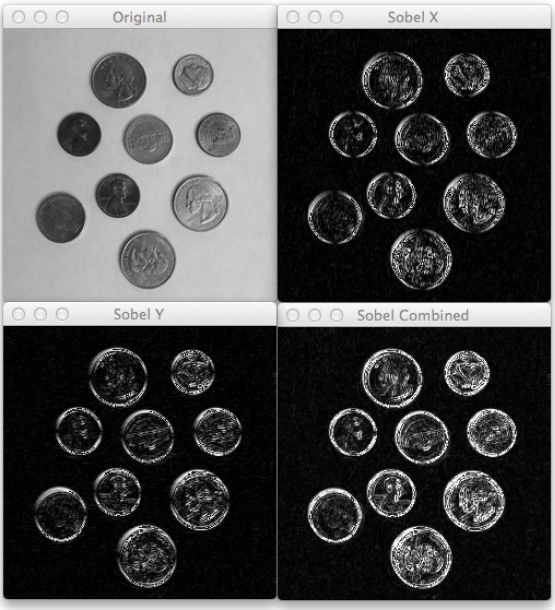

Figure 10.2: *Top-Left:* The original coins image. *Top-Right:* Computing the Sobel gradient magnitude along the x-axis (finding vertical edges). *Bottom-Left:* Computing the Sobel gradient along the y-axis (finding horizontal edges). *Bottom-Right:* Applying a bitwise OR to combine the two Sobel representations.

In fact, that's exactly what **Lines 18 and 19** do by using the `cv2.Sobel` method. The first argument to the Sobel operator is the image we want to compute the gradient representation for. Then, just like in the Laplacian example above, we use a floating point data type. The last two arguments are the order of the derivatives in the x and y direction, respectively. Specify a value of 1 and 0 to find vertical edge-like regions and 0 and 1 to find horizontal edge-like regions

On **Lines 21 and 22** we then ensure we find all edges by taking the absolute value of the floating point image and then converting it to an 8-bit unsigned integer.

In order to combine the gradient images in both the x and y direction, we can apply a bitwise OR. Remember, an OR operation is true when *either* pixel is greater than zero. Therefore, a given pixel will be `True` if either a horizontal or vertical edge is present.

Finally, we show our gradient images on **Lines 26-28**.

You can see the result of our work in Figure 10.2. We start with our original image *Top-Left* and then find vertical edges *Top-Right* and horizontal edges *Bottom-Left*. Finally, we compute a bitwise OR to combine the two directions into a single image *Bottom-Right*.

One thing you'll notice is that the edges are very "noisy". They are not clean and crisp. We'll remedy that by using the Canny edge detector in the next section.

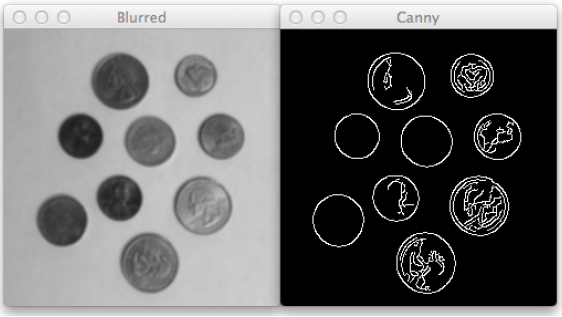

Figure 10.3: *Left:* Our coins image in grayscale and blurred slightly. *Right:* Applying the Canny edge detector to the blurred image to find edges. Notice how our edges more "crisp" and the outlines of the coins are found.

10.2 CANNY EDGE DETECTOR

The Canny edge detector is a multi-step process. It involves blurring the image to remove noise, computing Sobel gradient images in the x and y direction, suppression of edges, and finally a hysteresis thresholding stage that determines if a pixel is "edge-like" or not.

We won't get into all these steps in detail. Instead, we'll just look at some code and show how it's done:

Listing 10.3: canny.py

```
1 import numpy as np
2 import argparse
```

```
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 image = cv2.GaussianBlur(image, (5, 5), 0)
13 cv2.imshow("Blurred", image)
14
15 canny = cv2.Canny(image, 30, 150)
16 cv2.imshow("Canny", canny)
17 cv2.waitKey(0)
```

The first thing we do is import our packages and parse our arguments. We then load our image, convert it to grayscale, and blur it using the Gaussian blurring method. By applying a blur prior to edge detection, we will help remove "noisey" edges in the image that are not of interest to us. Our goal here is to find *only* the outlines of the coins.

Applying the Canny edge detector is performed on **Line 15** using the `cv2.Canny` function. The first argument we supply is our blurred, grayscale image. Then, we need to provide two values: `threshold1` and `threshold2`.

Any gradient value larger than `threshold2` are considered to be an edge. Any value below `threshold1` are considered not to be an edge. Values in between `threshold1` and `threshold2` are either classified as edges or non-edges based on how their intensities are "connected". In this case, any gradient values below 30 are considered non-edges whereas any value above 150 are considered edges.

We then show the results of our edge detection on **Line 16**.

Figure 10.3 shows the results of the Canny edge detector. The image on the *left* is our grayscale, blurred image that we pass into the Canny operator. The image on the *right* is the result of applying the Canny operator.

Notice how the edges are more "crisp". We have substantially less noise than we used the Laplacian or Sobel gradient images. Furthermore, the outline of our coins are clearly revealed.

In the next chapter we'll continue to make use of the Canny edge detector and use it to count the number of coins in our image.