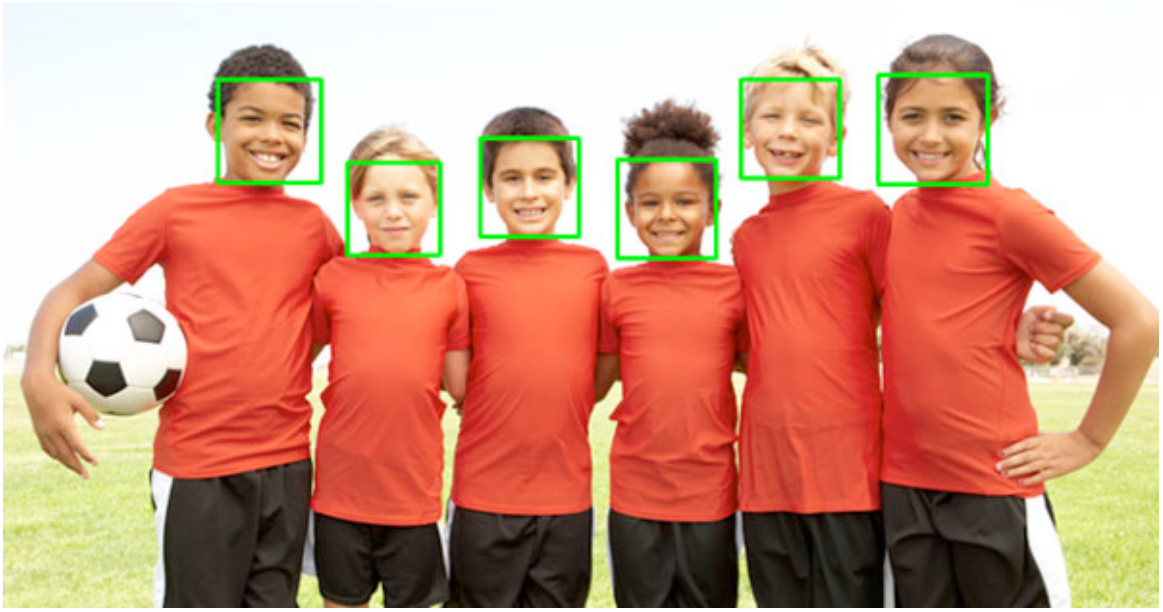
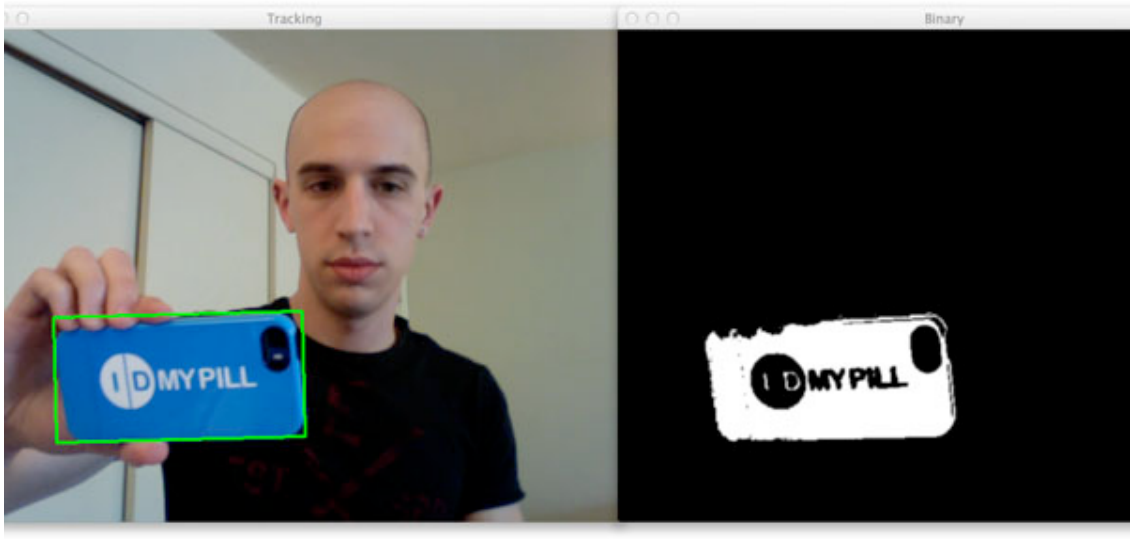
Hey, thank you for taking a look at this sample chapter, I appreciate it.

Let's start off with a preview of what you'll learn when you pick up a copy of my new book, *Practical Python and OpenCV.*

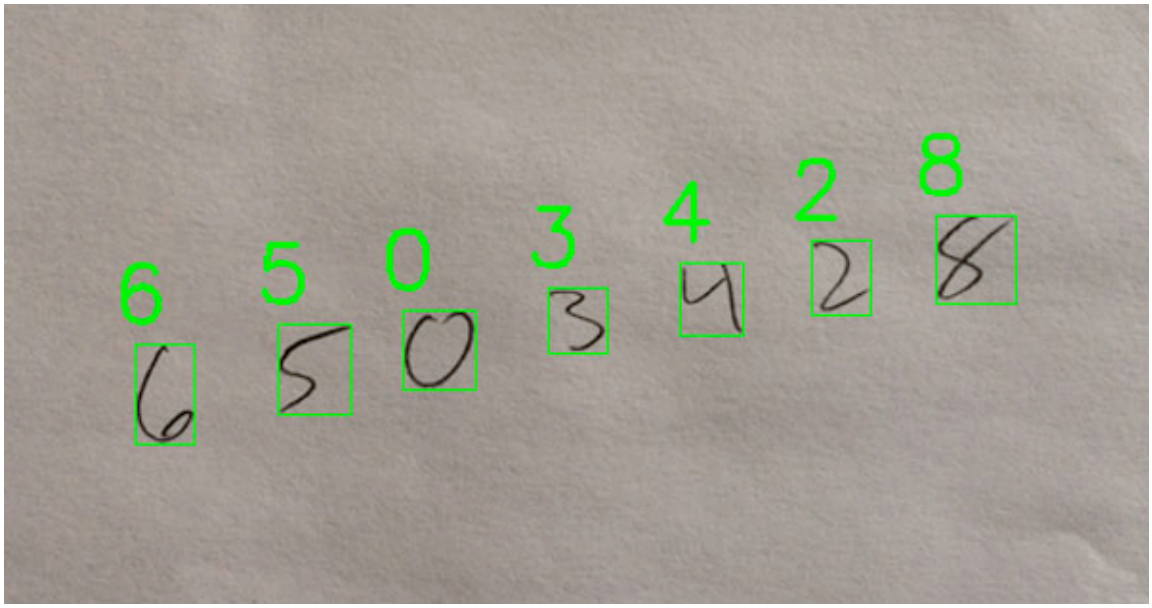
Are you curious how to *detect faces in images?*



What about *track objects in video?*



Ever wonder how computers can *recognize handwritten digits?*



Or apply *machine learning* to classify plant species?

What if you could identify book covers in a snap...*of your smartphone?*

These may seem like advanced computer vision techniques...*but I promise you can learn them too.*

Of course, don't take my word for it, just check out these great reviews by other *Practical Python and OpenCV* readers:

"Your teaching method rocks. Hands down." - Giridhur S.

"I was lost for a couple of months until I ran into the books **Practical Python and OpenCV** *and* **Case Studies.** *From that moment, I was able to face my university final project with utter confidence; now, I can automatically detect and recognize license plates in a parking lot. Adrian's writing style is clear, straightforward, and very easy to understand (even for insecure beginners), but also very close and entertaining. I'm happy I found it." - Eduardo Valenzuela*

“It is an easy read, step by step approach. Smarter than any reference manual I have read.” - Lai-Cheung-Kit Christophe

"I can guarantee you that Case Studies and Practical Python and OpenCV are the best books to teach you OpenCV and Python right now." - One June Kang

“That perfect first step if you are interested in computer vision but don't know where to start...You'll be glued to your workstation as you try out just one more example.” - Jason Brownlee

“First of all, thank you for the book. I find it really valuable and helpful. It gave me a good grasp on my path to learning computer vision/image processing. Now I have a good starting point to continue learning, exploring, and how to apply the OpenCV library in my new ideas.” - Mikko Leppänen

As you can see, I'm not making this up...

These books are your *guaranteed, quick-start guides* to learning computer vision.

The rest of this PDF includes sample chapters from *Practical Python and OpenCV* and *Case Studies*.

Continue reading to check out the sample chapters. Or, if you're ready, click here to pickup a copy.

I hope you enjoy the sample chapters! And if you have any questions, feel free to reach me at adrian@pyimagesearch.com.

Practical Python and OpenCV

An Introductory, Example Driven Guide to Image Processing and Computer Vision

Adrian Rosebrock

pyimagesearch

CONTENTS

PREFACE

When I first set out to write this book, I wanted it to be as hands-on as possible. I wanted lots of visual examples with lots of code. I wanted to write something that you could easily learn from, without all the rigor and detail of mathematics associated with college level computer vision and image processing courses.

I know that from all my years spent in the classroom that the way I learned best was from simply opening up an editor and writing some code. Sure, the theory and examples in my textbooks gave me a solid starting point. But I never really "learned" something until I did it myself. I was very hands on. And that's exactly how I wanted this book to be. Very hands on, with all the code easily modifiable and well documented so you could play with it on your own. That's why I'm giving you the full source code listings and images used in this book.

More importantly, I wanted this book to be accessible to a wide range of programmers. I remember when I first started learning computer vision – it was a daunting task. But I learned a lot. And I had a lot of fun.

I hope this book helps you in your journey into computer vision. I had a blast writing it. If you have any questions, suggestions or comments, or if you simply want to say hello, shoot me an email at adrian@pyimagesearch.com, or

you can visit my website at www.PyImageSearch.com and leave a comment. I look forward to hearing from you soon!

-Adrian Rosebrock

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.

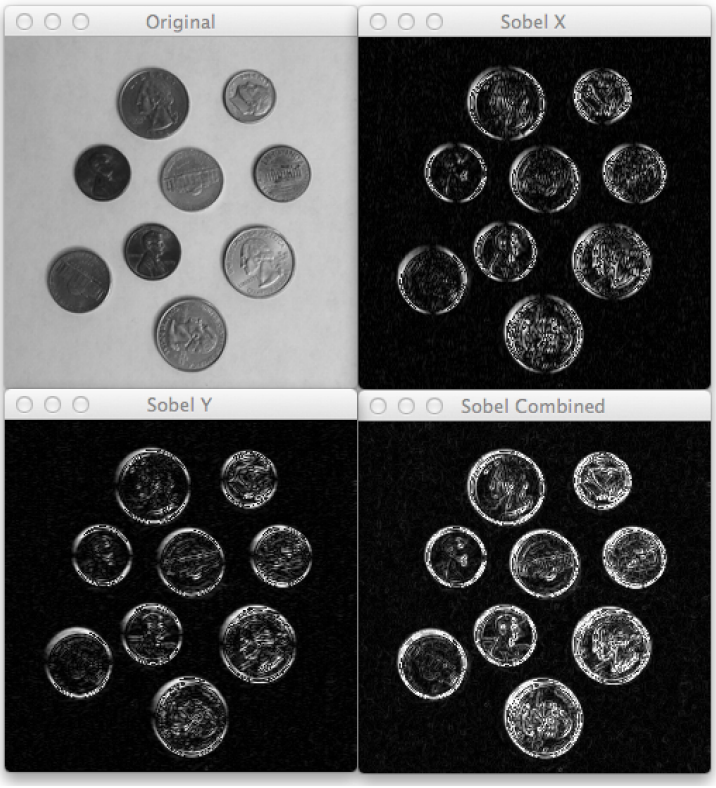


Figure 10.2: *Top-Left:* The original coins image. *Top-Right:* Computing the Sobel gradient magnitude along the x-axis (finding vertical edges). *Bottom-Left:* Computing the Sobel gradient along the y-axis (finding horizontal edges). *Bottom-Right:* Applying a bitwise OR to combine the two Sobel representations.

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.

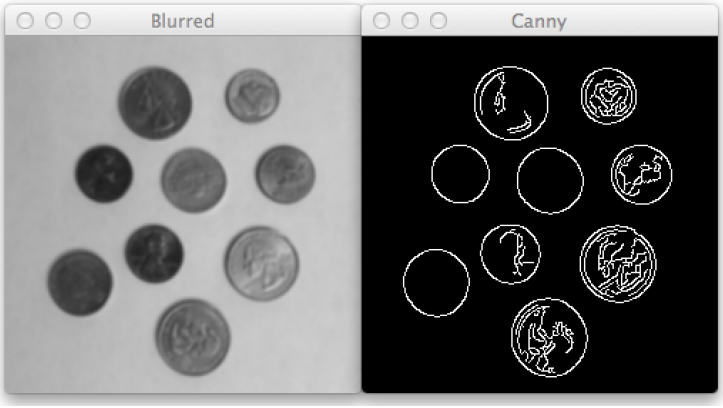


Figure 10.3: *Left:* Our coins image in grayscale and blurred slightly. *Right:* Applying the Canny edge detector to the blurred image to find edges. Notice how our edges more "crisp" and the outlines of the coins are found.

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.

As you can see, *Practical Python and OpenCV* gives you a strong computer vision and image processing foundation. By the time you finish reading *Practical Python and OpenCV*, you'll have mastered the basics.

But if you're itching to learn more and apply your skills *to solve real world problems*...keep reading.

CASE STUDIES

Practical Python and OpenCV

An Introductory, Example Driven Guide to Image Processing and Computer Vision

Adrian Rosebrock

pyimagesearch

CONTENTS

8

BUILDING AN AMAZON.COM COVER SEARCH

"*Wu-Tang Clan* has a bigger vocabulary than Shakespeare", argued Gregory, taking a draw of his third Marlboro menthol of the morning, as the dull San Francisco light struggled to penetrate the dense fog.

This was Gregory's favorite justification for his love of the famous hip-hop group. But this morning Jeff, his cofounder, wasn't up for arguing. Instead, Jeff was focused on integrating Google Analytics into their newly finished company website, which at the moment, consisted of nothing more than a logo and a short "About Us" section.

Five months ago, Gregory and Jeff quit their full time jobs and created a start up focused on visual recognition. Both of them were convinced that they had the "next big idea".

So far, they weren't making a money. And they couldn't afford an office either.

But working from a San Francisco park bench next to a taco truck that serves kimchi and craft beers definitely had

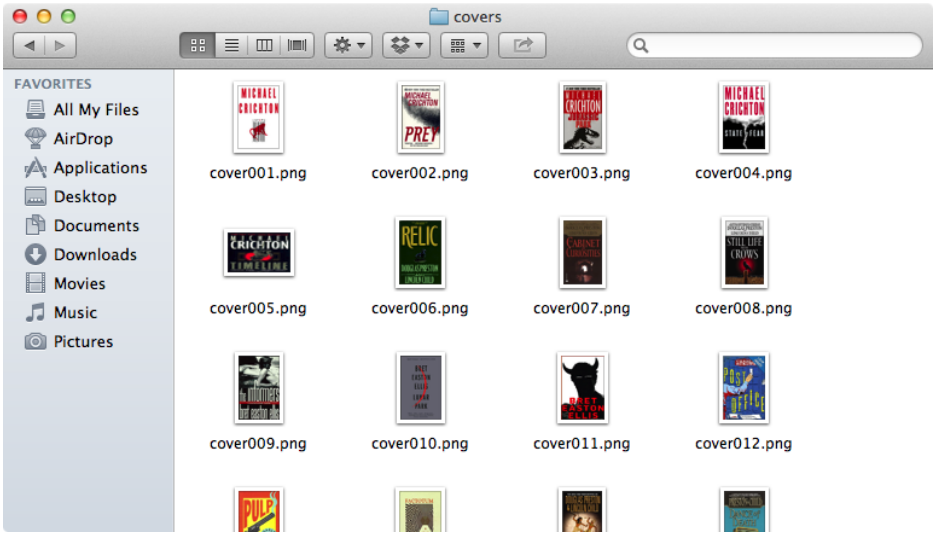


Figure 8.1: A sample of Gregory's book cover database. He has collected images for a small number of books and then plans on recognizing them using keypoint matching techniques.

its upsides.

Gregory looked longingly at the taco truck. 10am. It wasn't open yet. Which is probably a good thing. He had started to put on a few pounds after consuming a few months worth of tacos and IPAs. And he didn't have the funds to afford a gym membership to work those pesky pounds off.

See, Gregory's plan was to compete with the Amazon Flow app. Flow allows users to use their smartphone camera as a digital shopping device. By simply taking a picture of a cover of a book, DVD, or video game, Flow automati-

cally identifies the product and adds it to the user's shopping cart.

That's all fine and good. But Gregory wants to niche it down. He wants to to cater *strictly* on the bookworms and do only book cover identification – and do it better than anyone on the market, including Amazon. That would get some attention.

With the thought of beating Amazon at their own bread and butter, Gregory opens his laptop, cracks his knuckles, slowly, one at a time, stubs his cigarette out, and gets to work:

Listing 8.1: coverdescriptor.py

```
1 import numpy as np
2 import cv2
3
4 class CoverDescriptor:
5     def __init__(self, kpMethod = "SIFT", descMethod = "SIFT"):
6         self.kpMethod = kpMethod
7         self.descMethod = descMethod
8
9     def describe(self, image):
10        detector = cv2.FeatureDetector_create(self.kpMethod)
11        kps = detector.detect(image)
12
13        extractor = cv2.DescriptorExtractor_create(self.
              descMethod)
14        (kps, descs) = extractor.compute(image, kps)
15        kps = np.float32([kp.pt for kp in kps])
16
17        return (kps, descs)
```

Gregory starts by importing the packages that he needs on **Lines 1 and 2**: NumPy for numerical processing and `cv2` for OpenCV bindings.

He then defines his `CoverDescriptor` class on **Line 4** which encapsulates finding keypoints in an image and then describing the area surrounding each keypoint using local invariant descriptors.

The `__init__` constructor is defined on **Line 5**, requiring two optional parameters: `kpMethod` and `descMethod`.

The `kpMethod` is the keypoint detection method, which defaults to `SIFT`. Specifying a value of `SIFT` indicates that keypoint detection using David Lowe's Difference of Gaussian method [1] should be used. Other acceptable values include (but are not limited to) `SURF` and `ORB`.

The second parameter is the `descMethod`, which defines how the area surrounding each keypoint is described. Again, the default value is `SIFT`, but other local invariant descriptors can be supplied, such as `SURF`, `ORB`, `BRIEF`, and `FREAK`, depending on which OpenCV version you are using.

It is important to note that `SIFT` and `SURF` produce *real-valued* feature vectors whereas `ORB`, `BRIEF`, and `BRISK` produce *binary* feature vectors. When comparing real-valued descriptors, like SIFT or SURF, Gregory will likely want to use the Euclidean distance. If the descriptors are binary, then the Hamming distance should be utilized.

To extract both keypoints and descriptors from an image, Gregory defines the `describe` method on **Line 9**, which accepts a single parameter – the `image` in which keypoints

1 Lowe, David (1999). "Object recognition from local scale-invariant features". *Proceedings of the International Conference on Computer Vision*.

and descriptors should be extracted from.

Line 10 defines the keypoint `detector` using the method defined in the `CoverDescriptor` constructor and the `cv2.FeatureDetector_create` function. **Line 11** then detects the actual keypoints.

Now that Gregory has his list of keypoints, he now needs to extract descriptors from the region of the image surrounding each keypoint.

To do this, he creates his feature `extractor` on **Line 13** using the `cv2.DescriptorExtractor_create` function and the local invariant descriptor method supplied in the constructor. The descriptors are then computed for each keypoint on **Line 14**.

The list of keypoints contain multiple `KeyPoint` objects which are defined by OpenCV. These objects contain information such as the *(x, y)* location of the keypoint, the size of the keypoint, the rotation angle, amongst other attributes.

For this application, Gregory only needs the *(x, y)* coordinates of the keypoint, contained in the `pt` attribute.

He grabs the *(x, y)* coordinates for the keypoints, discarding the other attributes, and stores the points as a NumPy array on **Line 15**.

Finally, a tuple of keypoints and corresponding descriptors are returned to the calling function on **Line 17**.

At this point Gregory can extract keypoints and descriptors from the covers of books...but how is he going to compare them?

Let's check out Gregory's `CoverMatcher` class:

Listing 8.2: covermatcher.py

```
1 import numpy as np
2 import cv2
3
4 class CoverMatcher:
5     def __init__(self, descriptor, coverPaths):
6         self.descriptor = descriptor
7         self.coverPaths = coverPaths
```

Again, Gregory starts off on **Lines 1 and 2** by importing the packages he will need: NumPy for numerical processing and `cv2` for his OpenCV bindings.

The `CoverMatcher` class is defined on **Line 4** and the constructor on **Line 5**. The constructor takes two parameters, the `descriptor`, which is assumed to be an instantiation of the `CoverDescriptor` defined above, and the path to the directory where the cover images are stored.

This wasn't too exciting. Let's check out the `search` method and see how the keypoints and descriptors will be matched:

Listing 8.3: covermatcher.py

```
9     def search(self, queryKps, queryDescs):
10        results = {}
11
12        for coverPath in self.coverPaths:
13            cover = cv2.imread(coverPath)
14            gray = cv2.cvtColor(cover, cv2.COLOR_BGR2GRAY)
```

```
15            (kps, descs) = self.descriptor.describe(gray)
16
17            score = self.match(queryKps, queryDescs, kps, descs)
18            results[coverPath] = score
19
20        if len(results) > 0:
21            results = sorted([(v, k) for (k, v) in results.items()
                  if v > 0],
22                reverse = True)
23
24        return results
```

Gregory defines his `search` method on **Line 9**, requiring two arguments – the set of keypoints and descriptors extracted from the *query image*. The goal of this method is take the keypoints and descriptors from the query image and then match them against a database of keypoints and descriptors. The entry in the database with the best "match" will be chosen as the identification of the book cover.

To store his results of match accuracies, Gregory defines a dictionary of `results` on **Line 10**. The key of the dictionary will be the unique book cover filename and the value will be the matching percentage of keypoints.

Then, on **Line 12** Gregory starts looping over the list of book cover paths. The book cover is loaded from disk on **Line 13**, converted to grayscale on **Line 14**, and then keypoints and descriptors are extracted from it using the `CoverDescriptor` passed into the constructor on **Line 15**.

The number of matched keypoints is then determined using the `match` method (defined below) and the `results` dictionary updated on **Lines 17-18**.

Pages 90-97 have been omitted from the sample chapter.

placed at the top of the list.

Finally, Gregory displays the query image to the user on **Line 31**.

Now let's take a look at how Gregory is going to display the results to the user:

Listing 8.7: search.py

```
33 if len(results) == 0:
34     print "I could not find a match for that cover!"
35     cv2.waitKey(0)
36
37 else:
38     for (i, (score, coverPath)) in enumerate(results):
39         (author, title) = db[coverPath[coverPath.rfind("/") +
               1:]]
40         print "%d. %.2f%% : %s - %s" % (i + 1, score * 100,
41             author, title)
42
43         result = cv2.imread(coverPath)
44         cv2.imshow("Result", result)
45         cv2.waitKey(0)
```

First, Gregory makes a check on **Line 33** to ensure that at least one book cover match was found. If a match was not found, Gregory lets the user know.

If a match was found, he then starts to loop over the `results` on **Line 38**.

The unique filename of the book is extracted and the author and book title grabbed from the book database on **Line 39** and displayed to the user on **Line 40**.

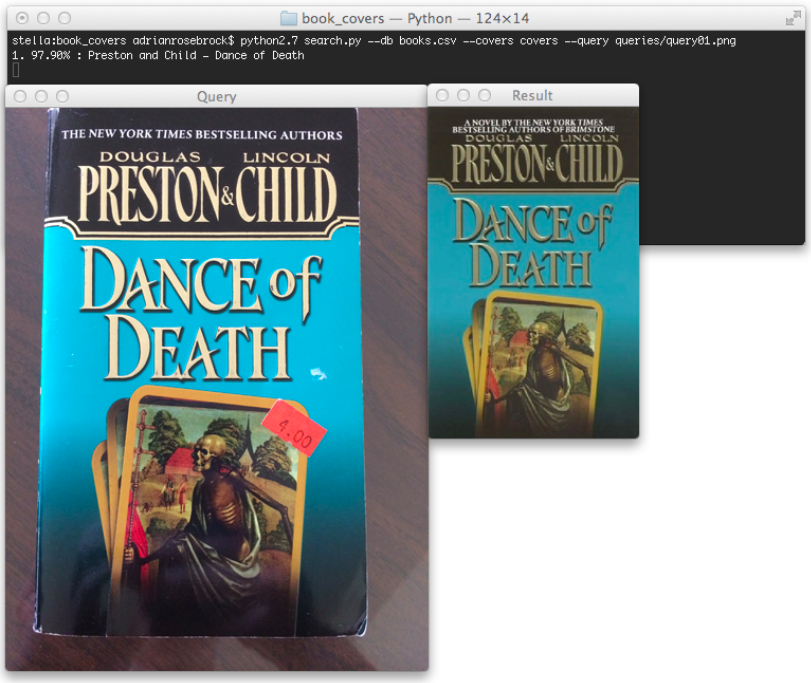


Figure 8.3: *Left:* The query image Gregory wants to match in his database. *Right:* The book cover in his database with the most matches (97.9%). The book cover has been successfully matched.

Finally, the actual book cover itself is loaded off disk and displayed to the user on **Lines 43-45**.

Gregory executes his script using the following command:

Listing 8.8: search.py

```
$ python search.py --db books.csv --covers covers --query queries
    /query01.png
```

Figure 8.4: Gregory now attempts to match Crichton's *Next*. Even despite laptop in the background and the hand covering part of the book, 87.38% of the keypoints were matched.

The results of Gregory's script can be seen in Figure 8.3, where Gregory attempts to match the cover of Preston and Child's *Dance of Death left* to the corresponding cover in his database *right*.

As Figure 8.3 demonstrates the covers were successfully matched, with over 97% of the keypoints matched as well. Indeed, out of all the 50 book covers, Gregory's code was able to correctly identify the cover of the book.

Gregory then moves on to another book cover, this time Michael Crichton's *Next* in Figure 8.4. Again, the query image is on the *left* and the successful match on the *right*. Despite the laptop in the background and the hand covering part of the book, 87.38% of the keypoints were matched. Once again, Gregory's algorithm has successfully identified the cover of the book!

Jurassic Park is Gregory's favorite book. So why not use it as a query image?

Figure 8.5 demonstrates that Gregory's keypoint matching code can successfully match the *Jurassic Park* book covers together. Yet another successful identification!

Time for a real test.

Gregory takes a photo of *State of Fear* and uses it as his query image in Figure 8.6. Notice how the book is angled and there is text at the top of the cover (i.e. "New York Times Bestseller") that does not appear in the cover database.

Regardless, his cover match algorithm is robust enough to handle this situation and a successful match is found.

Finally, Gregory attempts to match another Lincoln and Child book, *The Book of the Dead* in Figure 8.7. Despite the 30% off stick in the upper right hand corner and substantially different lighting conditions, the two covers are matched without an issue.

Figure 8.5: *Jurassic Park* is Gregory's favorite book. Sure enough, his algorithm is able to match the cover to his book database without an issue.

Figure 8.6: Notice how the book is not only angled in the query image, but there is also text on the cover that doesn't exist in the image database. None-theless, Gregory's algorithm is still easily able to identify the book.

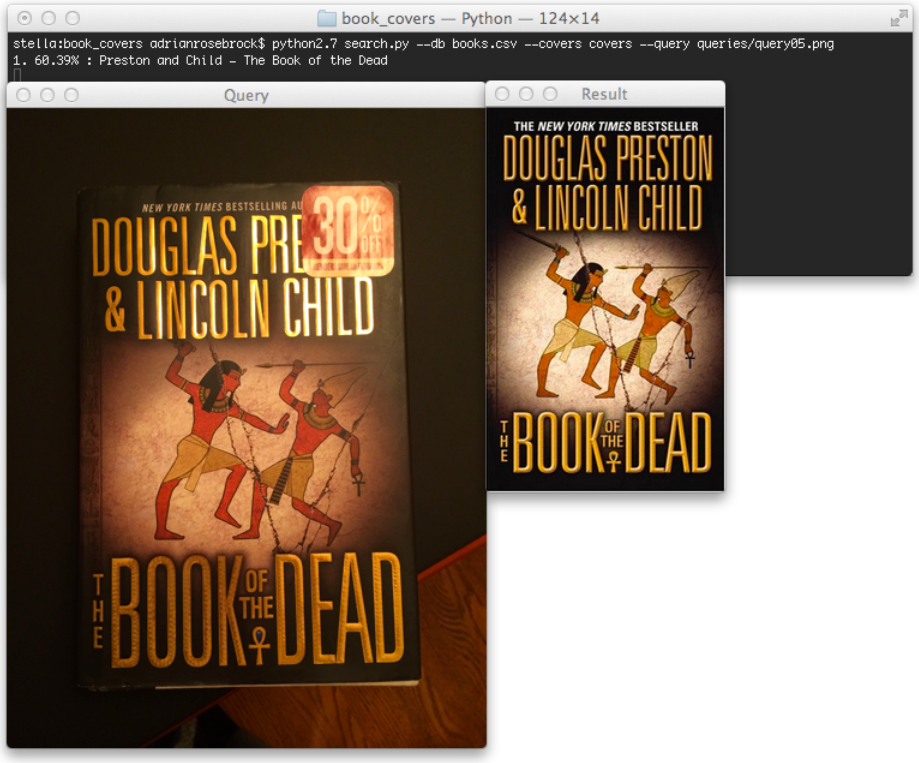


Figure 8.7: Despite the 30% off sticker and the substantially different lighting conditions, *The Book of the Dead* book cover is easily matched.

"Ghostface Killah is the best member of *Wu-Tang Clan*", Gregory announced to Jeff, turning his laptop to show in the results.

But this time, Jeff could not ignore him.

Gregory had actually done it. He had created a book cover identification algorithm that was working with beautifully.

"Next stop, acquisition by Amazon!", exclaimed Jeff. "Now, let's get some tacos and beers to celebrate."

It looks like you have reached the end of the sample chapters!

To pickup your copy of *Practical Python and OpenCV*, **just click the button below.**

And remember, I am so confident that I can teach you the basics of computer vision and image processing **in a single weekend** that I am offering you a **100% money back guarantee** on it.

With an offer like that, *what have you got to lose?*

So grab a copy today -- I recommend the *Case Studies* or *Premium Bundle*. By the end of the weekend, **you'll be a computer vision guru**, I promise.

Click here to pickup your copy!