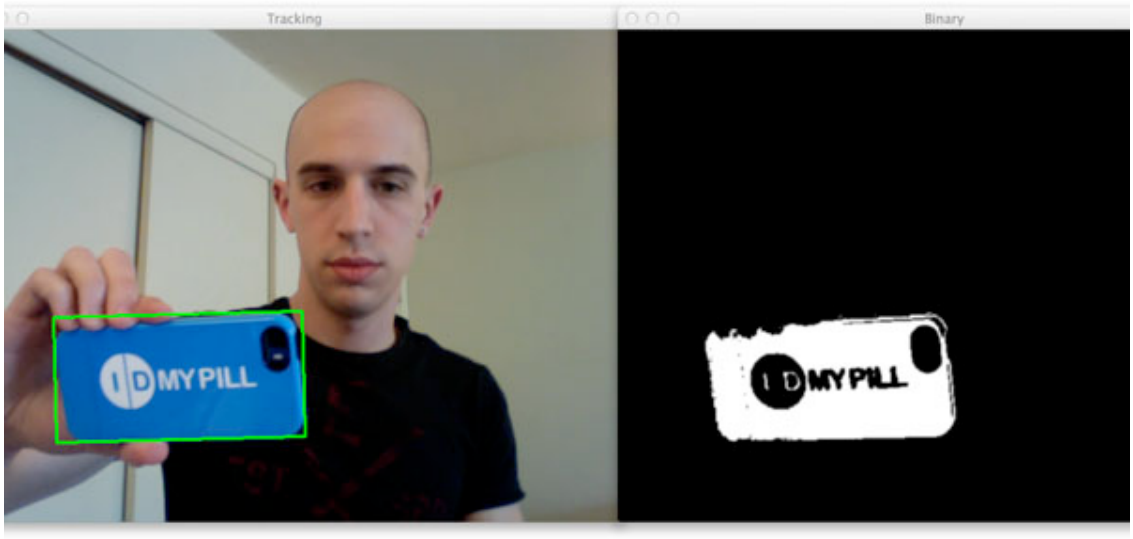
Thank you for downloading a sample chapter of *Practical Python and OpenCV + Case Studies!*

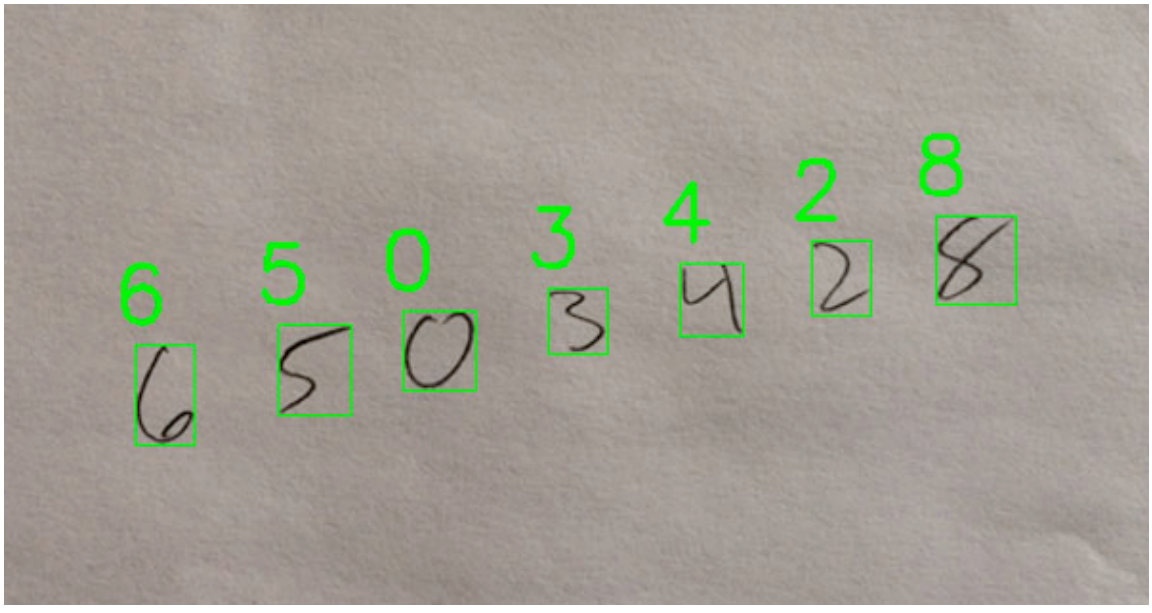
Before we get started, let me ask you a question...

Are you curious how to *detect faces in images, video,* and *webcam streams?*

What about *track objects in video?*



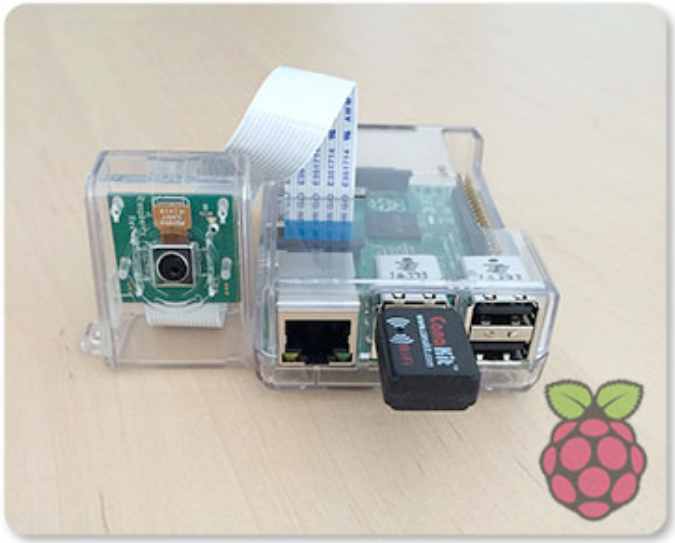
Ever wonder how computers can *recognize handwritten digits?*



Or apply *machine learning* to classify plant species?

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

Not to mention, do all of this on your *Raspberry Pi?*



These techniques aren't as advanced as you think...*and you can learn them too.*

Just follow *my proven approach* and *you'll be a computer vision ninja* in no time.

Interested?

Keep reading to learn more...

If you've gotten this far in the sample chapter then I assume you are *serious* about learning computer vision.

And if you're serious, then I have a *two-step process* guaranteed to make you awesome at computer vision.

The first step is to read through Part #1 of my book, *Practical Python and OpenCV*: the *definitive* quickstart guide to learning the fundamentals of computer vision.

The second step is to go through Part #2, *Case Studies*, where you'll utilize your knowledge of the fundamentals to solve *actual, real-world* computer vision problems.

It's really that easy.

And to prove it to you, I have included *two sample chapters* from *Practical Python and OpenCV* + *Case Studies* in this PDF.

- ***Part #1: Practical Python and OpenCV:*** Are you interested in computer vision and image processing and don't know where to start? Part #1 of my book is your **guaranteed quickstart guide** to learning the fundamentals of computer vision using Python and OpenCV.

- ***Part #2: Case Studies:*** From here you'll apply your knowledge of computer vision basics to solve *actual, real-world* computer vision problems including ***detecting faces in images & video***, ***tracking objects***

in video, ***handwriting recognition***, and ***book cover identification*** utilizing SIFT and keypoint matching.

Using the techniques you learn inside my book, you'll unlock the secrets the computer vision pros use...*and become a pro yourself.*

Of course, don't take my word for it, just check out these great testimonials:

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

"Many thanks for Practical Python and OpenCV + Case Studies. I am working through it steadily. It is fantastic. I am new to image processing and have found the details and explanations intuitive and easy to understand. Well worth the money" - **Mary Kennedy**

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

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

These books are your *guaranteed, quickstart guides* to learning computer vision.

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

3RD EDITION

Dr. Adrian Rosebrock

pyimagesearch

CONTENTS

PYTHON AND OPENCV VERSIONS

Over a year ago, when I wrote the first edition of *Practical Python and OpenCV + Case Studies*, the current version of OpenCV was 2.4.9, which only supported Python 2.7. While many scientific developers (myself included) are very much accustomed to using Python 2.7, newcomers to computer vision and machine learning were often confused and frustrated by the lack of Python 3 support – Python 3 is the future of the Python programming language, after all!

However, this all changed on June 4th, 2015, which marked a momentous date in the history of OpenCV:

OpenCV 3.0 was finally released!

The benefits of OpenCV 3.0 are numerous, including improved stability, performance, increases, and even transparent OpenCL support.

But *by far* the most exciting update to us in the Python world is:

Python 3 support!

After years of being stuck and sequestered to Python 2.7, **we can now finally use OpenCV with Python 3+!**

While the differences in OpenCV (at least with the Python bindings) are relatively small between OpenCV 2.4.X and OpenCV 3.0+, there are enough to warrant a new edition of *Practical Python and OpenCV + Case Studies*.

In this edition, I have updated all chapters, code samples, and datasets to be compatible with **OpenCV 3.0**. Furthermore, all code examples will run in both the **Python 2.7** *and* the **Python 3+** environments!

If you are looking for the OpenCV 2.4.X and Python 2.7 version of this book, please look in the download directory associated with your purchase – inside you will find the OpenCV 2.4.X + Python 2.7 edition.

Thank you so much for making this book possible, and please enjoy the new edition of *Practical Python and OpenCV + Case Studies!*

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 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 set up 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 can 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 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 modulus op-

eration 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)
29 cv2.waitKey(0)
```

Using the Sobel operator, we can compute gradient magnitude representations along the x and y axis, allowing us

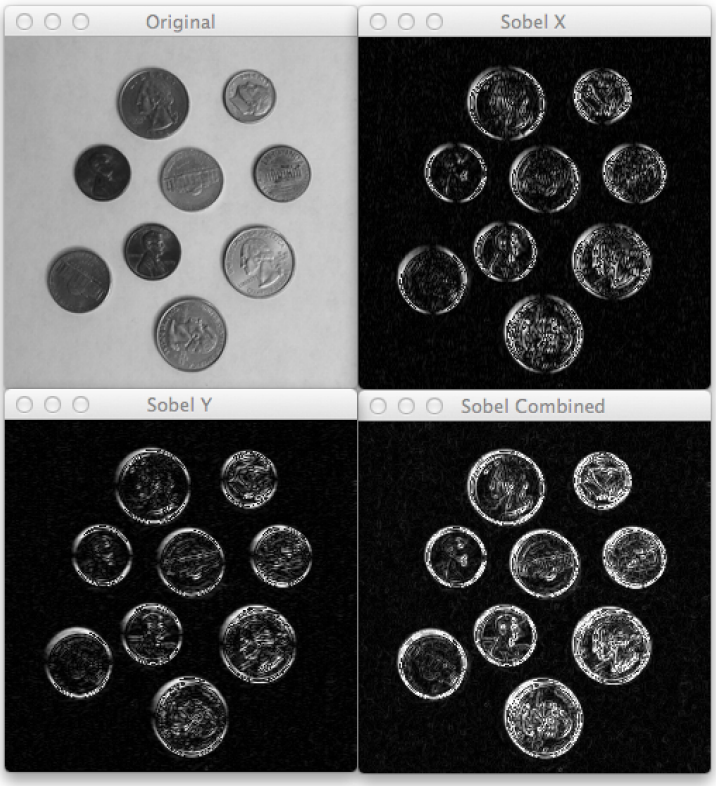


Figure 10.2: *Top-Left:* The original coins image. *Top-Right:* Computing the Sobel gradient magnitude along the x-axis (finding vertical edges). *Bottom-Left:* Computing the Sobel gradient along the y-axis (finding horizontal edges). *Bottom-Right:* Applying a bitwise OR to combine the two Sobel representations.

to find both horizontal and vertical edge-like regions.

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 26-29**.

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

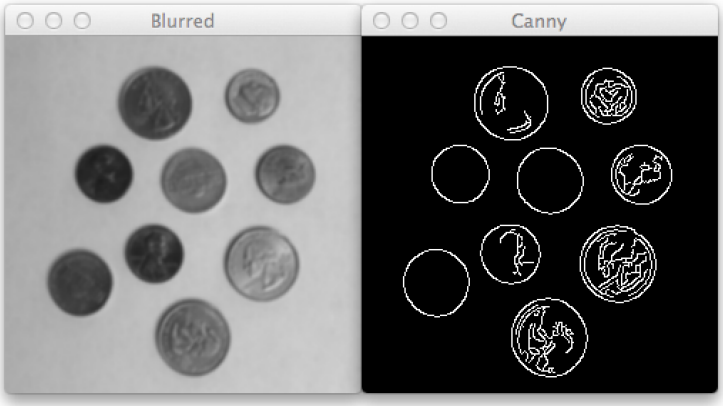


Figure 10.3: *Left:* Our coins image in grayscale and blurred slightly. *Right:* Applying the Canny edge detector to the blurred image to find edges. Notice how our edges are more "crisp" and the outlines of the coins are found.

the Canny edge detector in the next section.

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, suppressing 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 "noisy" 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` is considered to be an edge. Any value below `threshold1` is 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 values 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 the 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 when 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.

Further Reading

Just like thresholding is a common method for segmenting foreground objects from background objects, the same can be said for edge detection – only instead of obtaining a large *blob* representing the foreground, the Canny detector gives us the *outline*.

However, a common challenge of using the Canny edge detector is getting the lower and upper edge thresholds just right. In order to help you (automatically) determine these lower and upper boundaries, be sure to read about the *automatic Canny edge detector* in the Chapter 10 supplementary material:

http://pyimg.co/91daw

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.

Supports OpenCV 3 & Python 3+!

Practical Python and OpenCV

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

3RD EDITION

CASE STUDIES

Dr. 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 startup 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.

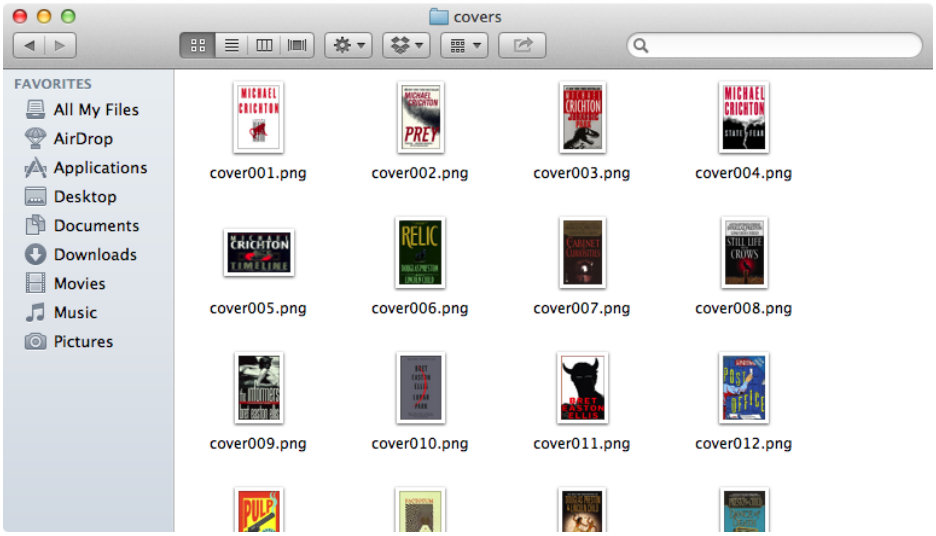


Figure 8.1: A sample of Gregory's book cover database. He has collected images for a small number of books and then plans on recognizing them using keypoint matching techniques.

But working from a San Francisco park bench next to a taco truck that serves kimchi and craft beers definitely has its upsides.

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

See, Gregory's plan was to compete with the Amazon Flow app. Flow allows users to use their smartphone cam-

era as a digital shopping device. By simply taking a picture of a cover of a book, DVD, or video game, Flow automatically 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 cater *strictly* to 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, useSIFT = False):
6         self.useSIFT = useSIFT
```

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 methods for finding keypoints in an image and then describes the area surrounding each keypoint using local invariant descriptors.

The `__init__` constructor is defined on **Line 5**, requiring one optional argument: `useSIFT`, a boolean indicating whether the SIFT keypoint detector and descriptor should be used or not.

8.1 KEYPOINTS, FEATURES, AND OPENCV 3

Now, before we get too far into this chapter, let's briefly discuss an important change to the organization of the OpenCV library.

In the 1st edition of *Practical Python and OpenCV + Case Studies*, our book cover identification system defaulted to using David Lowe's Difference of Gaussian (DoG) keypoint detector and the SIFT local invariant descriptor. [1]

However, the OpenCV 3.0 release brings some *very important* changes to the library – specifically to the modules that contain keypoint detectors and local invariant descriptors.

With the v3.0 release, OpenCV has moved SIFT, SURF, FREAK, and other keypoint detector and local invariant descriptor implementations into the optional `opencv_contrib` package. This move was to consolidate (1) experimental implementations of algorithms, and (2) what OpenCV calls "non-free" (i.e., patented) algorithms, which include many popular keypoint detectors and local invariant descriptors, into a 100% optional module that is not required for OpenCV to install and function. In short, if you have ever used the `cv2.FeatureDetector_create` or `cv2.DescriptorExtractor_create` functions from OpenCV 2.4.X, they are **no longer**

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

part of OpenCV. You can still access the free, non-patented methods such as ORB and BRISK, but if you need SIFT and SURF, you'll have to *explicitly enable* them at compile and install time.

Again, in order to obtain access to these keypoint detectors and descriptors, you will need to follow the OpenCV 3 install instructions for your appropriate operating system and Python version provided in Chapter 2 of *Practical Python and OpenCV*.

Note: *For more information regarding this change to OpenCV and the implications it has to keypoint detectors, local invariant descriptors, and which Python versions have access to which functions, I suggest you read through my detailed blog post which you can find here: http://pyimg.co/9vwm9*.

Since OpenCV is no longer shipped with the the SIFT module enabled automatically, we now supply a boolean to our `__init__` method called `useSIFT` with a default value of `False` to indicate that SIFT should only be used if the programmer **explicitly** wants it to be.

However, just because SIFT is not be a part of our OpenCV install doesn't mean we're out of luck – *far from it!*

OpenCV 3 has still retained many non-patented keypoint detectors and local invariant descriptors such as BRISK, ORB, KAZE, and AKAZE. Rather than using SIFT to build our book cover identification system like we did in the 1st edition of this book, we'll instead be using BRISK – the code will change slightly, but our results will still be the same.

8.2 IDENTIFYING THE COVERS OF BOOKS

Now that we have clarified an important difference between OpenCV 2.4.X and OpenCV 3.0, let's get back to Gregory's code:

Listing 8.2: coverdescriptor.py

```
8     def describe(self, image):
9         descriptor = cv2.BRISK_create()
10
11        if self.useSIFT:
12            descriptor = cv2.xfeatures2d.SIFT_create()
13
14        (kps, descs) = descriptor.detectAndCompute(image, None)
15        kps = np.float32([kp.pt for kp in kps])
16
17        return (kps, descs)
```

To extract both keypoints and descriptors from an image, Gregory defines the `describe` method on **Line 8**, which accepts a single parameter – the `image` in which keypoints and descriptors should be extracted from.

On **Line 9**, Gregory initializes his `descriptor` method to utilize `BRISK`. However, if he has set `useSIFT=True` in the constructor, then he re-initializes the `descriptor` to use `SIFT` on **Lines 11 and 12**.

Now that Gregory has his descriptor initialized, he makes a call to the `detectAndCompute` method on **Line 14**. As this name suggests, this method both *detects* keypoints (i.e., "interesting" regions of an image) and then *describes* and *quantifies* the region surrounding each of the keypoints. Thus, the keypoint detection is the "detect" phase, whereas the actual description of the region is the "compute" phase.

Again, the choice between SIFT and BRISK is dependent upon your OpenCV install. If you are using OpenCV 2.4.X, take a look at the 1st edition of this book included in the download of your bundle – it will detail how to use the `cv2.FeatureDetector_create` and `cv2.DescriptorExtractor_create` functions to obtain access to SIFT, SURF, BRISK, etc. However, if you are using OpenCV 3, then you'll want to use the `BRISK_create` method unless you have taken *explicit care* to install OpenCV 3 with the `xfeatures2d` module enabled.

Regardless of whether Gregory is using SIFT or BRISK, 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, and 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 8.3: covermatcher.py

```
1 import numpy as np
2 import cv2
3
4 class CoverMatcher:
5     def __init__(self, descriptor, coverPaths, ratio = 0.7,
6         minMatches = 40, useHamming = True):
7         self.descriptor = descriptor
8         self.coverPaths = coverPaths
9         self.ratio = ratio
10        self.minMatches = minMatches
11        self.distanceMethod = "BruteForce"
12
13        if useHamming:
14            self.distanceMethod += "-Hamming"
```

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 required parameters and three optional ones. The two required parameters are 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.

The three optional arguments are detailed below:

- **ratio**: The ratio of nearest neighbor distances suggested by Lowe to prune down the number of keypoints a homography needs to be computed for.
- **minMatches**: The minimum number of matches required for a homography to be calculated.

- **useHamming**: A boolean indicating whether the Hamming or Euclidean distance should be used to compare feature vectors.

The first two arguments, `ratio` and `minMatches`, we'll discuss in more detail when we get to the `match` function below. However, the third argument, `useHamming`, is especially important and worth diving into now.

You see, it is important to note that `SIFT` and `SURF` produce *real-valued* feature vectors whereas `ORB`, `BRISK`, and `AKAZE` produce *binary* feature vectors. When comparing real-valued descriptors, like `SIFT` or `SURF`, we would want to use the Euclidean distance. However, if we are using `BRISK` features which produce binary feature vectors, the Hamming distance should be used instead. Again, your choice in feature descriptor above (`SIFT` vs. `BRISK`) will also influence your choice in distance method. However, since Gregory is defaulting to `BRISK` features which produce binary feature vectors, he'll indicate that the Hamming method should be used (again, by default) on **Lines 13 and 14**.

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

Listing 8.4: covermatcher.py

```
16     def search(self, queryKps, queryDescs):
17         results = {}
18
19         for coverPath in self.coverPaths:
20             cover = cv2.imread(coverPath)
21             gray = cv2.cvtColor(cover, cv2.COLOR_BGR2GRAY)
22             (kps, descs) = self.descriptor.describe(gray)
23
24             score = self.match(queryKps, queryDescs, kps, descs)
```

```
25            results[coverPath] = score
26
27        if len(results) > 0:
28            results = sorted([(v, k) for (k, v) in results.items()
                  if v > 0],
29                reverse = True)
30
31        return results
```

Gregory defines his `search` method on **Line 16**, requiring two arguments – the set of keypoints and descriptors extracted from the *query image*. The goal of this method is to 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 17**. 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 19**, Gregory starts looping over the list of book cover paths. The book cover is loaded from disk on **Line 20**, converted to grayscale on **Line 21**, and then keypoints and descriptors are extracted from it using the `CoverDescriptor` passed into the constructor on **Line 22**.

The number of matched keypoints is then determined using the `match` method (defined below) and the `results` dictionary updated on **Lines 24-25**.

Gregory does a quick check to make sure that at least some results exist on **Line 27**. Then the results are sorted in *descending* order, with book covers with more keypoint

Pages 96-105 have been omitted from the sample chapter.

First, Gregory makes a check on **Line 34** 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 49**.

The unique filename of the book is extracted, and the author and book title are grabbed from the book database on **Line 50** and displayed to the user on **Line 51**.

Finally, the actual book cover itself is loaded off disk and displayed to the user on **Lines 54-56**.

Gregory executes his script using the following command:

Listing 8.10: search.py

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

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 together, 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 without a problem.

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*.

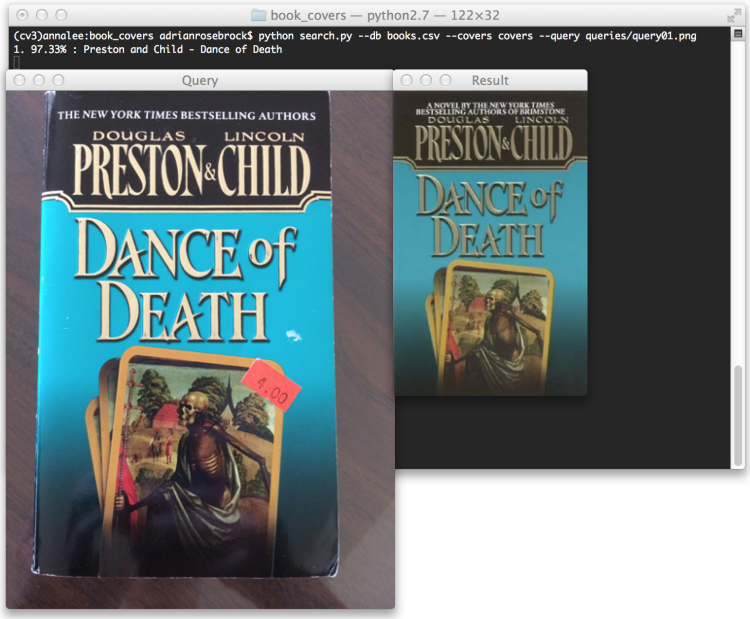


Figure 8.3: *Left:* The query image Gregory wants to match in his database. *Right:* The book cover in his database with the most matches (97.33%). The book cover has been successfully matched.

Figure 8.4: Gregory now attempts to match Crichton's *Next*. Even despite the laptop in the background and the hand covering part of the book, 73.47% of the keypoints were matched.

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.

Despite the laptop in the background and the hand covering part of the book, 73.47% 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 cov-

ers 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 Preston and Child book, *The Book of the Dead*, in Figure 8.7. Despite the 30% off stickers in the upper right hand corner and substantially different lighting conditions, the two covers are matched without an issue.

"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."

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. Nonetheless, Gregory's algorithm is still easily able to identify the book.

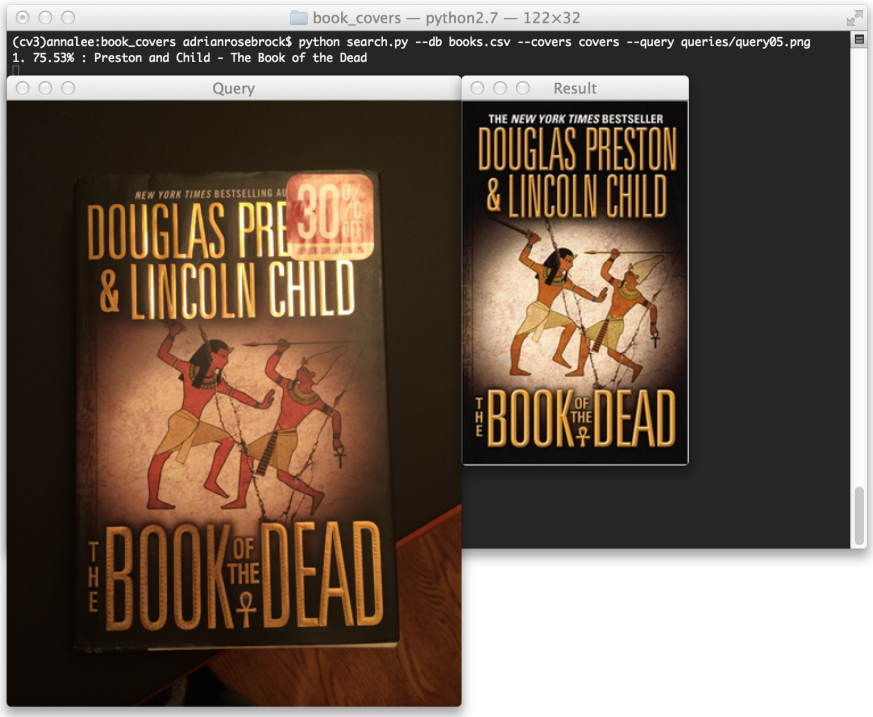


Figure 8.7: Despite the 30% off sticker and the substantially different lighting conditions, *The Book of the Dead* book cover is easily matched.

Further Reading

Keypoint detectors and local invariant descriptors are just the start to building scalable image search engines. The problem is that such systems scale *linearly*, meaning that as more images are added to our system, *the longer it will take to perform a search*.

So, how do we overcome this issue and build image search engines that can scale to *millions* of images? The answer is to apply the bag of visual words (BOVW) model:

http://pyimg.co/8klea

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

To pickup your copy of *Practical Python and OpenCV + Case Studies*, **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 either the *Quickstart Bundle* or *Hardcopy Bundle* (if you like the feel of a physical book in your hands). By the end of the weekend, **you'll be a computer vision ninja**, I promise.

Click here to pickup your copy!