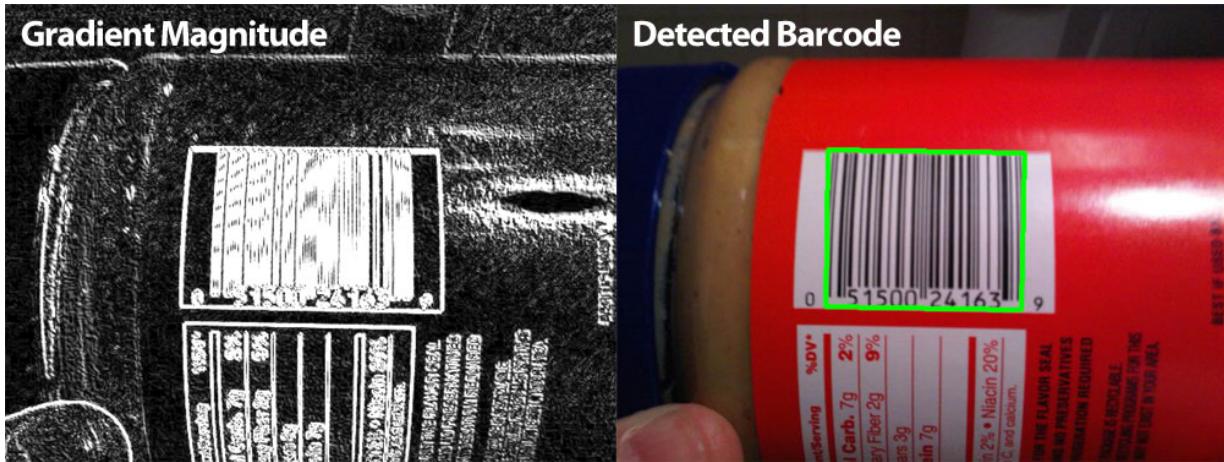


The Ultimate Guide to Barcode Detection

pyimagesearch

Part I: Detecting Barcodes in Images with Python and OpenCV

by Adrian Rosebrock



The goal of this blog post is to demonstrate a basic implementation of barcode detection using computer vision and image processing techniques. My implementation of the algorithm is originally based loosely on [this StackOverflow question](#). I have gone through the code and provided some updates and improvements to the original algorithm.

It's important to note that this algorithm will not work for all barcodes, but it should give you the basic intuition as to what types of techniques you should be applying.

For this example, we will be detecting the barcode in the following image:



Let's go ahead and start writing some code. Open up a new file, name it detect_barcode.py , and let's get coding:

```
# import the necessary packages
import numpy as np
import argparse
import cv2

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True, help = "path to the image file")
args = vars(ap.parse_args())
```

The first thing we'll do is import the packages we'll need. We'll utilize NumPy for numeric processing, `argparse` for parsing command line arguments, and `cv2` for our OpenCV bindings.

Then we'll setup our command line arguments. We need just a single switch here, `--image` , which is the path to our image that contains a barcode that we want to detect.

Now, time for some actual image processing:

```
# load the image and convert it to grayscale
image = cv2.imread(args["image"])
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# compute the Scharr gradient magnitude representation of the images
# in both the x and y direction
gradX = cv2.Sobel(gray, ddepth = cv2.cv.CV_32F, dx = 1, dy = 0, ksize = -1)
gradY = cv2.Sobel(gray, ddepth = cv2.cv.CV_32F, dx = 0, dy = 1, ksize = -1)

# subtract the y-gradient from the x-gradient
gradient = cv2.subtract(gradX, gradY)
gradient = cv2.convertScaleAbs(gradient)
```

On **Lines 12 and 13** we load our `image` off disk and convert it to grayscale.

Then, we use the Scharr operator (specified using `ksize = -1`) to construct the gradient magnitude representation of the grayscale image in the horizontal and vertical directions on **Lines 17 and 18**.

From there, we subtract the y-gradient of the Scharr operator from the x-gradient of the Scharr operator on **Lines 21 and 22**. By performing this subtraction we are left with regions of the image that have high horizontal gradients and low vertical gradients.

Our `gradient` representation of our original image above looks like:



Notice how the barcoded region of the image has been detected by our gradient operations.
The next steps will be to filter out the noise in the image and focus solely on the barcode region.

```
# blur and threshold the image  
blurred = cv2.blur(gradient, (9, 9))  
(_, thresh) = cv2.threshold(blurred, 225, 255, cv2.THRESH_BINARY)
```

The first thing we'll do is apply an average blur on **Line 25** to the gradient image using a 9×9 kernel. This will help smooth out high frequency noise in the gradient representation of the image.

We'll then threshold the blurred image on **Line 26**. Any pixel in the gradient image that is not greater than 225 is set to 0 (black). Otherwise, the pixel is set to 255 (white).

The output of the blurring and thresholding looks like this:



However, as you can see in the threshold image above, *there are gaps between the vertical bars of the barcode*. In order to close these gaps and make it easier for our algorithm to detect the “blob”-like region of the barcode, we’ll need to perform some basic morphological operations:

```
# construct a closing kernel and apply it to the thresholded image
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (21, 7))
closed = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)
```

We’ll start by constructing a rectangular kernel using the `cv2.getStructuringElement` on **Line 29**. This kernel has a width that is larger than the height, thus allowing us to close the gaps between vertical stripes of the barcode.

We then perform our morphological operation on **Line 30** by applying our kernel to our thresholded image, thus attempting to close the the gaps between the bars.

You can now see that the gaps are substantially more closed, as compared to the thresholded image

above:



Of course, now we have small blobs in the image that are not part of the actual barcode, but may interfere with our contour detection.

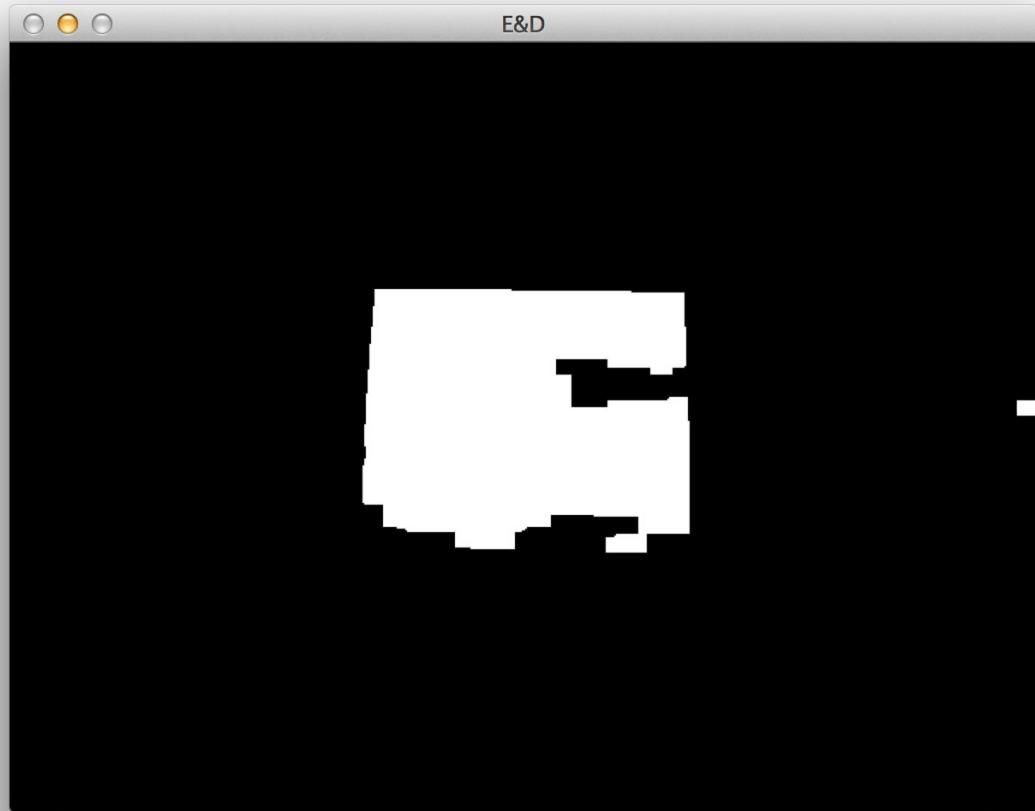
Let's go ahead and try to remove these small blobs:

```
# perform a series of erosions and dilations
closed = cv2.erode(closed, None, iterations = 4)
closed = cv2.dilate(closed, None, iterations = 4)
```

All we are doing here is performing 4 iterations of erosions, followed by 4 iterations of dilations. An erosion will “erode” the white pixels in the image, thus removing the small blobs, whereas a dilation will “dilate” the remaining white pixels and grow the white regions back out.

Provided that the small blobs were removed during the erosion, they will not reappear during the dilation.

After our series of erosions and dilations you can see that the small blobs have been successfully removed and we are left with the barcode region:



Finally, let's find the contours of the barcoded region of the image:

```
# find the contours in the thresholded image, then sort the contours
# by their area, keeping only the largest one
(cnts, _) = cv2.findContours(closed.copy(), cv2.RETR_EXTERNAL,
    cv2.CHAIN_APPROX_SIMPLE)
c = sorted(cnts, key = cv2.contourArea, reverse = True)[0]

# compute the rotated bounding box of the largest contour
rect = cv2.minAreaRect(c)
box = np.int0(cv2.cv.BoxPoints(rect))

# draw a bounding box around the detected barcode and display the
# image
cv2.drawContours(image, [box], -1, (0, 255, 0), 3)
cv2.imshow("Image", image)
cv2.waitKey(0)
```

Luckily, this is the easy part. On **Lines 38-40** we simply find the largest contour in the image, which if we have done our image processing steps correctly, should correspond to the barcoded region.

We then determine the minimum bounding box for the largest contour on **Lines 43 and 44** and finally display the detected barcode on **Lines 48-50**.

As you can see in the following image, we have successfully detected the barcode:



In the next section we'll try a few more images.

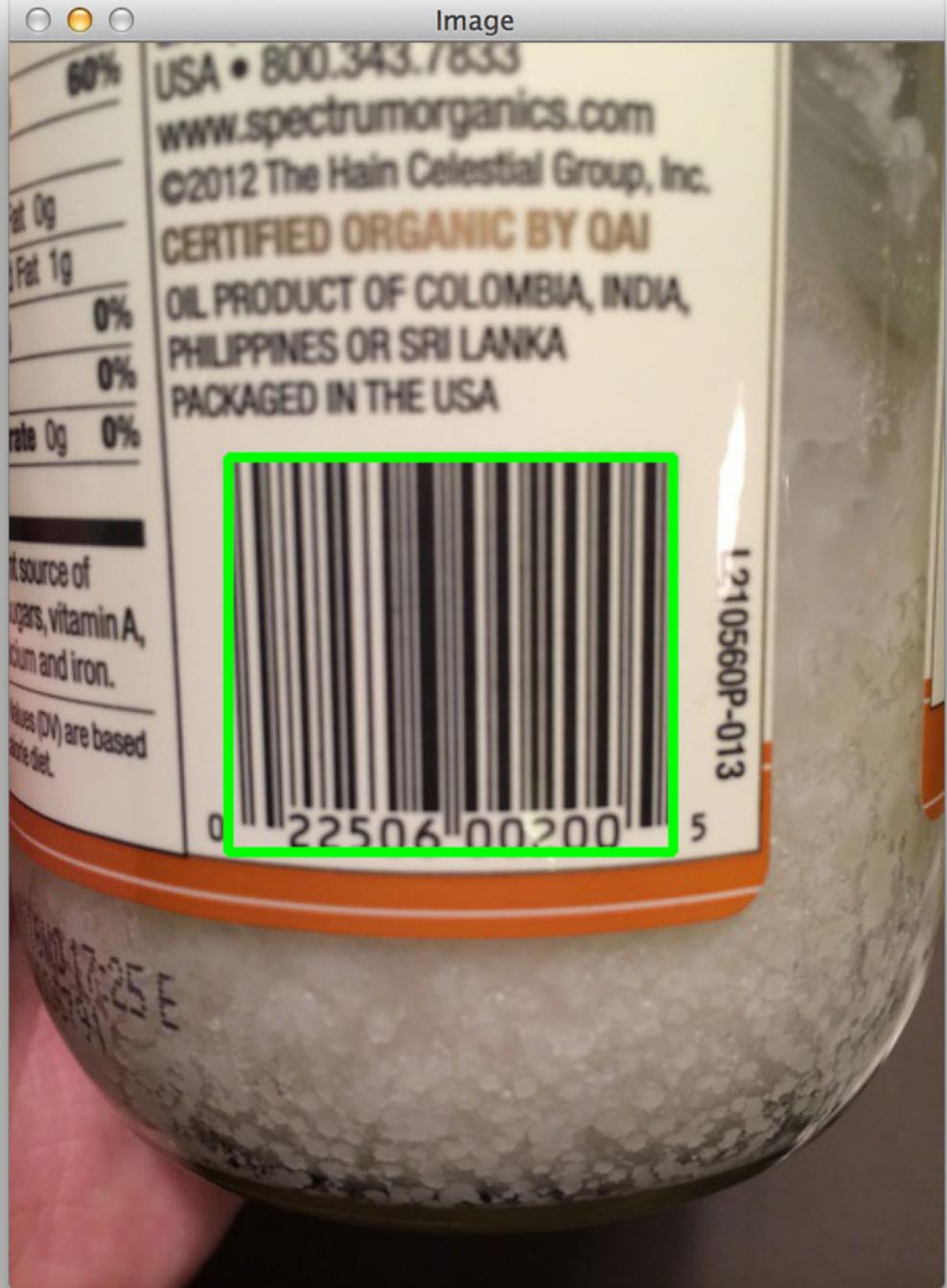
Successful Barcode Detections

To follow along with these results, use the form at the bottom of this post to download the source code and accompanying images for this blog post.

Once you have the code and images, open up a terminal and execute the following command:

```
$ python detect_barcode.py --image images/barcode_02.jpg
```

Image

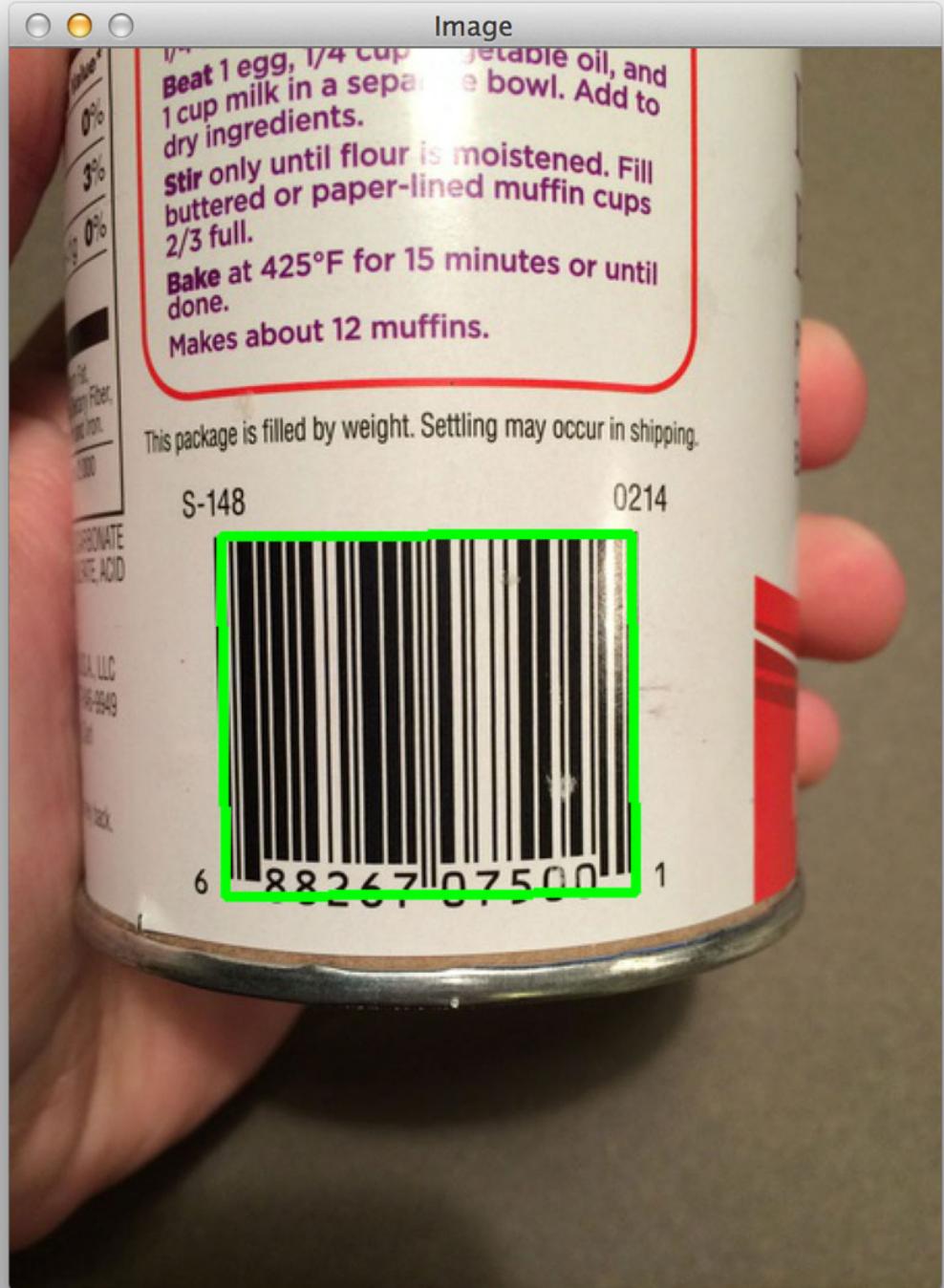


No problem detecting the barcode on that jar of coconut oil!

Let's try another image:

```
$ python detect_barcode.py --image images/barcode_03.jpg
```

Image

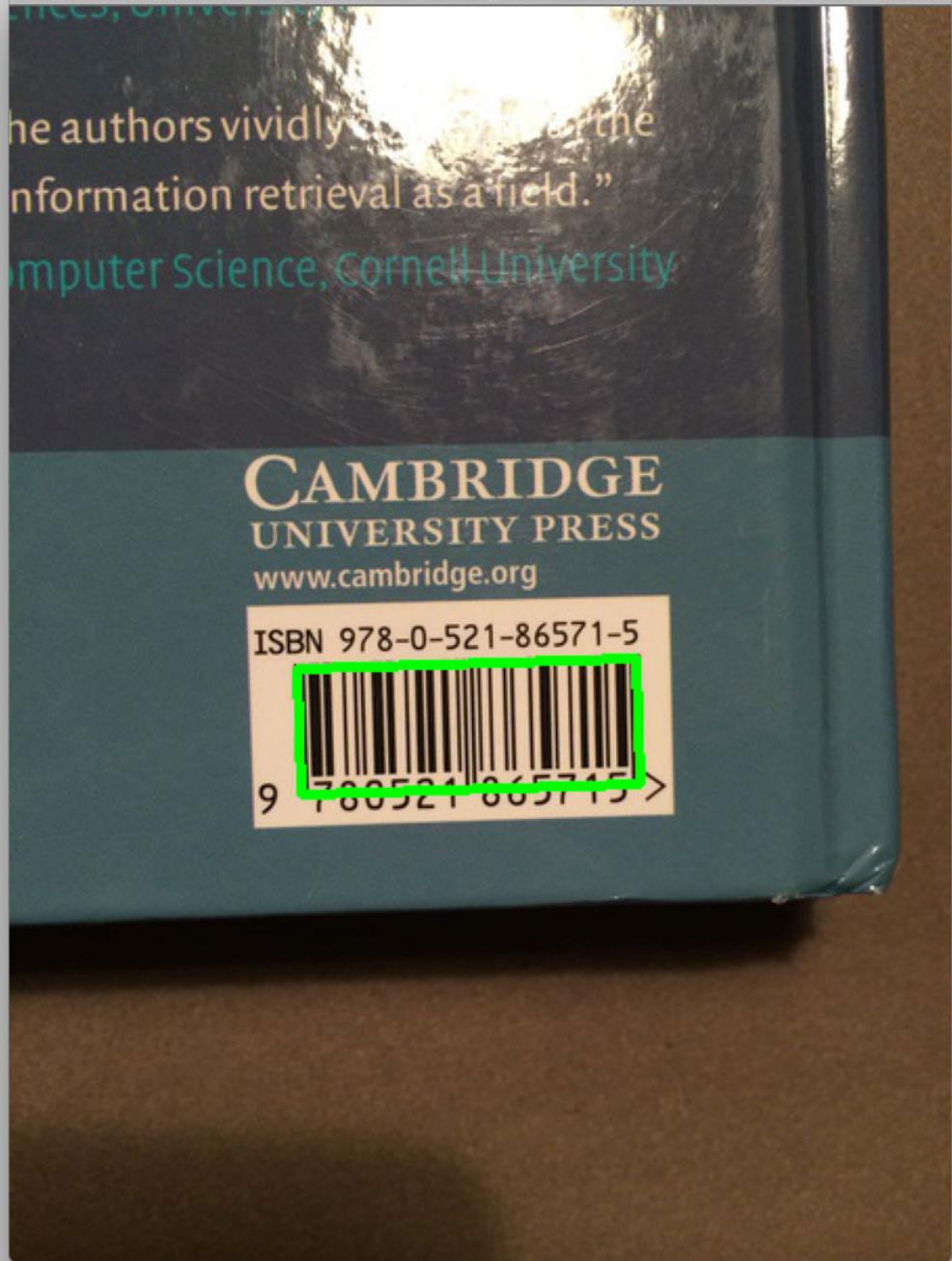


We were able to find the barcode in that image too!

But enough of the food products, what about the barcode on a book:

```
$ python detect_barcode.py --image images/barcode_04.jpg
```

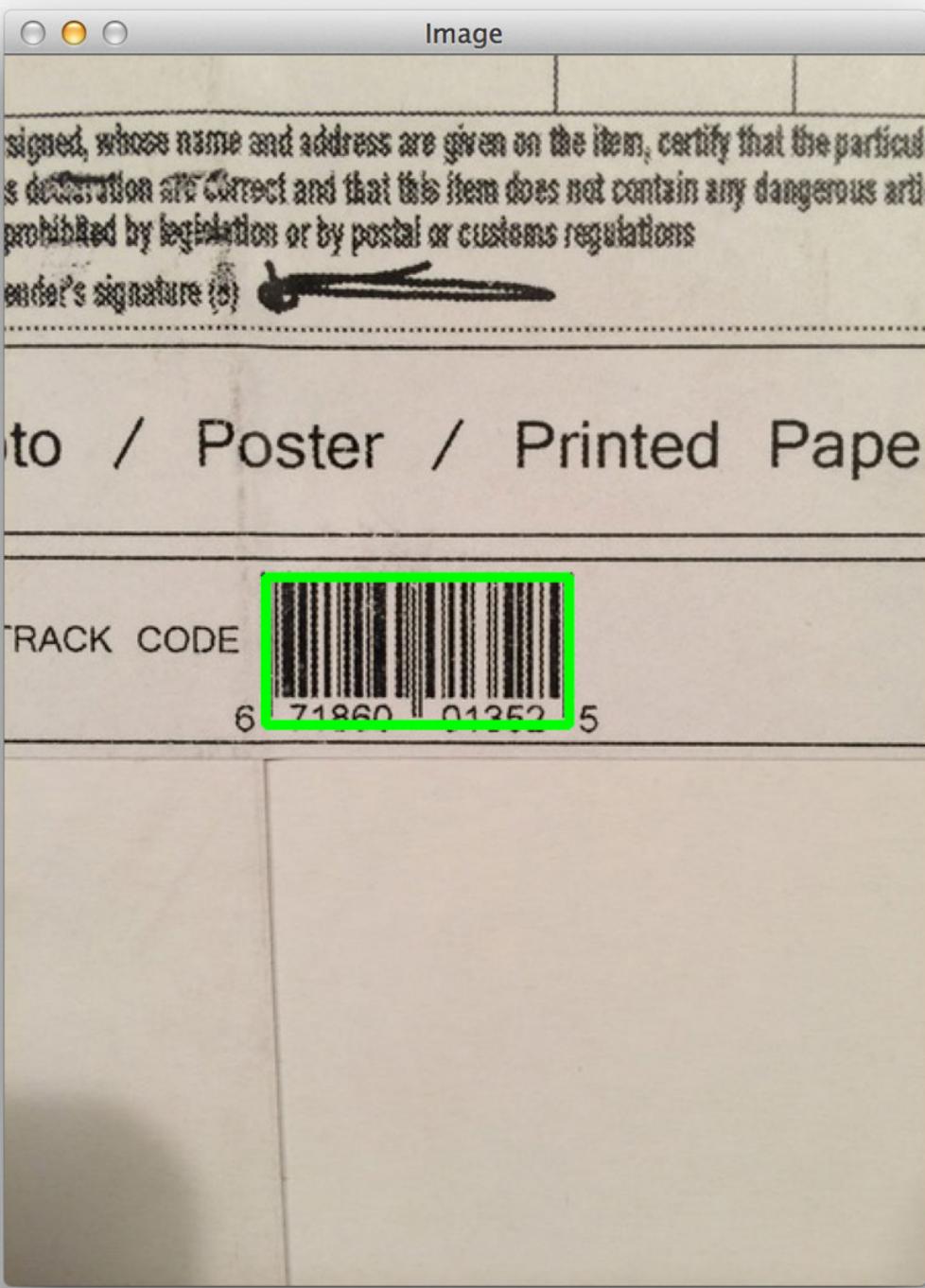
Image



Again, no problem!

How about the tracking code on a package?

```
$ python detect_barcode.py --image images/barcode_05.jpg
```

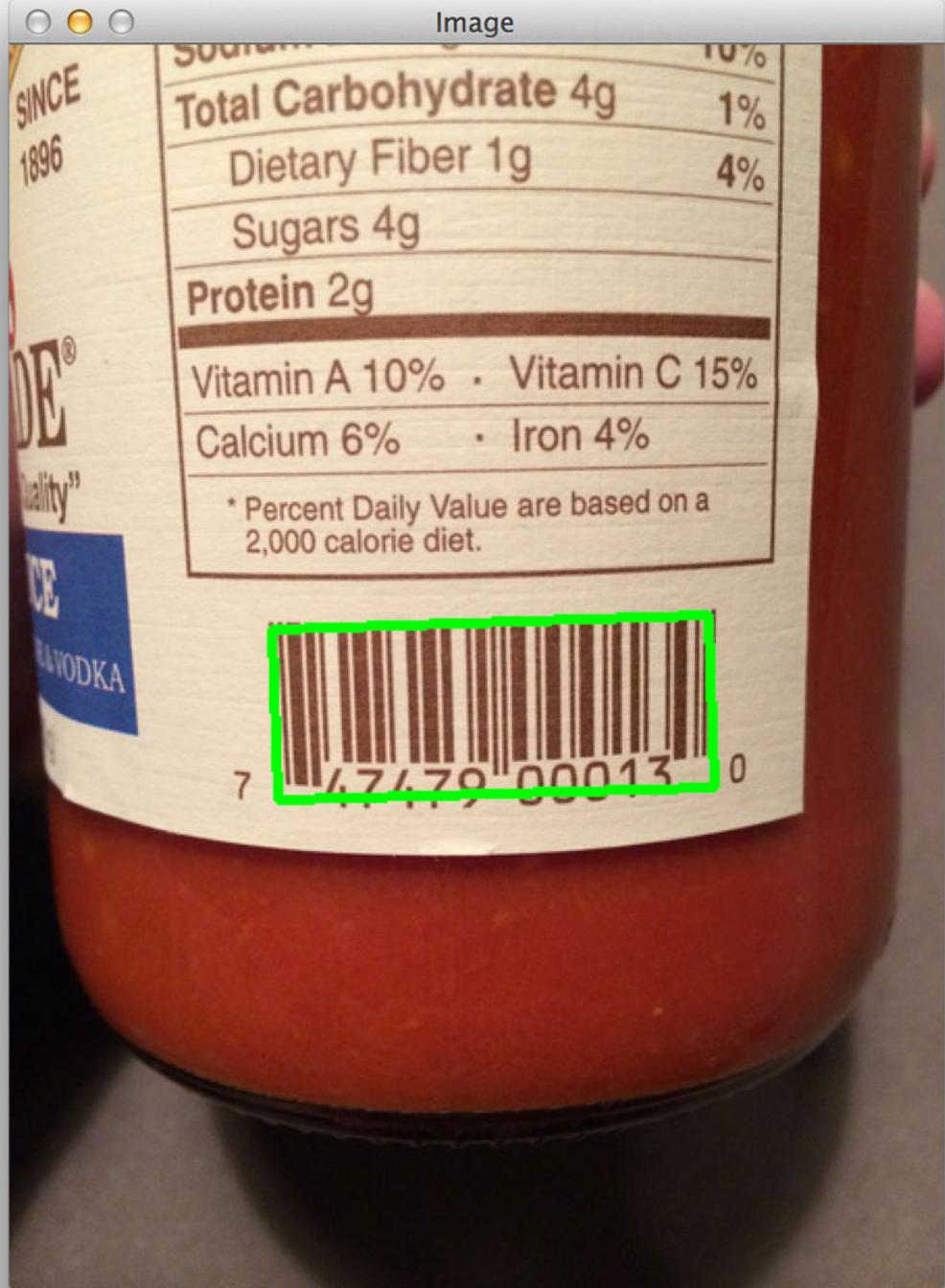


Again, our algorithm is able to successfully detect the barcode.

Finally, let's try one more image. This one is of my favorite pasta sauce, Rao's Homemade Vodka Sauce:

```
$ python detect_barcode.py --image images/barcode_06.jpg
```

Image



We were once again able to detect the barcode!

Summary

In this blog post we reviewed the steps necessary to detect barcodes in images using computer vision techniques. We implemented our algorithm using the Python programming language and the OpenCV library.

The general outline of the algorithm is to:

1. Compute the Scharr gradient magnitude representations in both the x and y direction.
2. Subtract the y-gradient from the x-gradient to reveal the barcoded region.
3. Blur and threshold the image.
4. Apply a closing kernel to the thresholded image.
5. Perform a series of dilations and erosions.
6. Find the largest contour in the image, which is now presumably the barcode.

It is important to note that since this method makes assumptions regarding the gradient representations of the image, and thus will only work for horizontal barcodes.

If you wanted to implement a more robust barcode detection algorithm, you would need to take the orientation of the image into consideration, or better yet, apply machine learning techniques such as Haar cascades or HOG + Linear SVM to “scan” the image for barcoded regions.

Downloads:

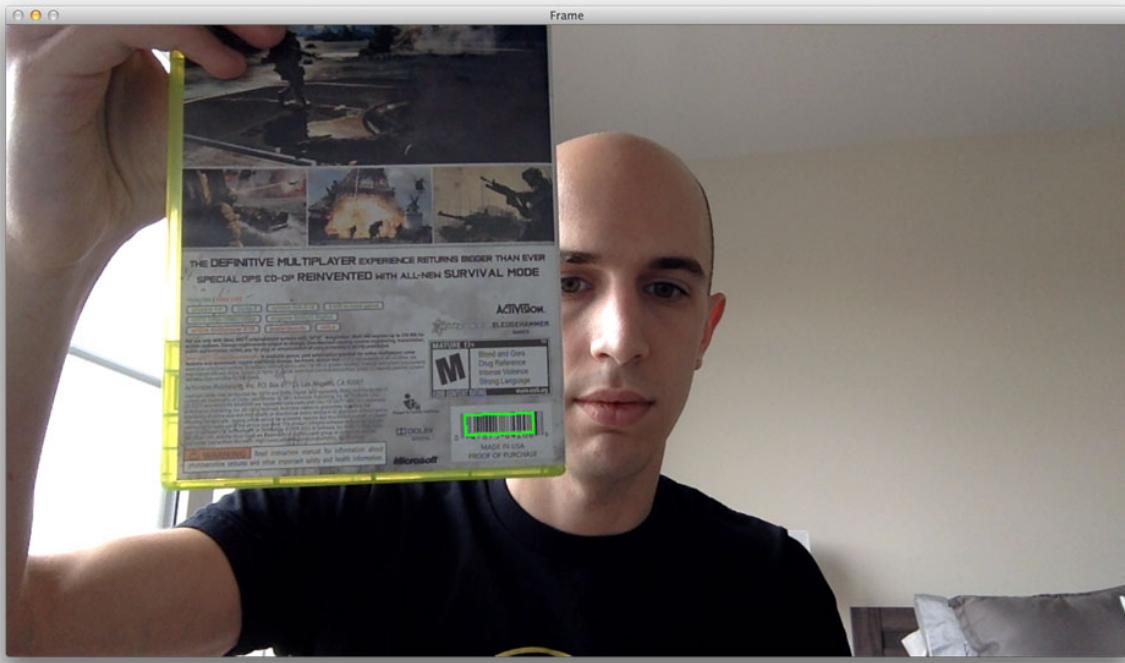
To download the source code to this article, just use this link: <http://www.pyimagesearch.com/wp-content/uploads/2014/11/detecting-barcodes-in-images.zip>

Part II: Real-time barcode detection in video with Python and OpenCV

In Part I of this guide we explored how to detect and find barcodes in images. But now we are going to extend our barcode detection algorithm and refactor the code to detect barcodes in video.



As an example, check out the screenshot below (captured using a video stream from my webcam) of me holding up the back cover of Modern Warfare 3 and the barcode being detected successfully.



For example, let's pretend that we are working at GameStop on the 26th of December. There are a line of kids ten blocks long outside our store — all of them wanting to return or exchange a game (obviously, their parents or relatives didn't make the correct purchase).

To speedup the exchange process, we are hired to wade out into the sea of teenagers and start the return/exchange process by scanning barcodes. But we have a problem — the laser guns at the register are wired to the computers at the register. And the chords won't reach far enough into the 10-block long line!

However, we have a plan. ***We'll just use our smartphones instead!***

Using our trusty iPhones (or Androids), we open up our camera app, set it to video mode, and head into the abyss.

Whenever we hold a video game case with a barcode in front of our camera, our app will detect it, and then relay it back to the register.

Sound too good to be true?

Well. Maybe it is. After all, you can accomplish this exact same task using laser barcode readers and a wireless connection. And as we'll see later on in this post that our approach only works in certain conditions.

But I still think this a good tutorial on how to utilize OpenCV and Python to read barcodes in video — and more importantly, it shows you how you can glue OpenCV functions together to build a real-world application.

Anyway, continue reading to learn how to detect barcodes in video using OpenCV and Python!

Real-time barcode detection in video with Python and OpenCV

So here's the game plan. Our barcode detection in video system can be broken into two components:

- **Component #1:** A module that handles detecting barcodes in images (or in this case, frames of a video) Luckily, we already have this (Part II of this guide). We'll just clean the code up a bit and reformat it to our purposes.
- **Component #2:** A driver program that obtains access to a video feed and runs the barcode detection module.

We'll go ahead and start with the first component, a module to detect barcodes in single frames of a video.

Component 1: Barcode detection in frames of a video

I'm not going to do a complete and exhaustive code review of this component, that was handled in Part I of this guide.

However, I will provide a quick review for the sake of completeness (and review a few minor updates). Open up a new file, name it `simple_barcode_detection.py`, and let's get coding:

```

# import the necessary packages
import numpy as np
import cv2

def detect(image):
    # convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # compute the Scharr gradient magnitude representation of the images
    # in both the x and y direction
    gradX = cv2.Sobel(gray, ddepth = cv2.cv.CV_32F, dx = 1, dy = 0, ksize = -1)
    gradY = cv2.Sobel(gray, ddepth = cv2.cv.CV_32F, dx = 0, dy = 1, ksize = -1)

    # subtract the y-gradient from the x-gradient
    gradient = cv2.subtract(gradX, gradY)
    gradient = cv2.convertScaleAbs(gradient)

    # blur and threshold the image
    blurred = cv2.blur(gradient, (9, 9))
    (_, thresh) = cv2.threshold(blurred, 225, 255, cv2.THRESH_BINARY)

    # construct a closing kernel and apply it to the thresholded image
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (21, 7))
    closed = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)

    # perform a series of erosions and dilations
    closed = cv2.erode(closed, None, iterations = 4)
    closed = cv2.dilate(closed, None, iterations = 4)

    # find the contours in the thresholded image
    (cnts, _) = cv2.findContours(closed.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)

    # if no contours were found, return None
    if len(cnts) == 0:
        return None

    # otherwise, sort the contours by area and compute the rotated
    # bounding box of the largest contour
    c = sorted(cnts, key = cv2.contourArea, reverse = True)[0]
    rect = cv2.minAreaRect(c)
    box = np.int0(cv2.cv.BoxPoints(rect))

    # return the bounding box of the barcode
    return box

```

If you read Part I of this guide then this code should look extremely familiar.

The first thing we'll do is import the packages we'll need — NumPy for numeric processing and `cv2` for our OpenCV bindings.

From there we define our `detect` function on **Line 5**. This function takes a single argument, the `image` (or frame of a video) that we want to detect a barcode in.

Line 7 converts our image to grayscale, while **Lines 9-16** find regions of the image that have high horizontal gradients and low vertical gradients.

We then blur and threshold the image on **Lines 19 and 20** so we can apply morphological operations to the image on **Lines 23-28**. These morphological operations are used to reveal the rectangular region of the barcode and ignore the rest of the contents of the image.

Now that we know the rectangular region of the barcode, we find its contour (or simply, its "outline") on **Lines 30 and 31**.

If no outline can be found, then we make the assumption that there is no barcode in the image (**Lines 35 and 36**).

However, if we do find contours in the image, then we sort the contours by their area on **Line 40** (where the contours with the largest area appear at the front of the list). Again, we are making the assumption that the contour with the largest area is the barcoded region of the frame.

Finally, we take the contour and compute its bounding box (**Lines 41 and 42**). This will give us the (x, y) coordinates of the barcoded region, which is returned to the calling function on **Line 45**.

Now that our simple barcode detector is finished, let's move on to Component #2, the driver that glues everything together.

Component #2: Accessing our camera to detect barcodes in video

Let's move on to building the driver to detect barcodes in video. Open up a new file, name it `detect_barcode.py`, and let's create the second component:

```

# import the necessary packages
from pyimagesearch import simple_barcode_detection
import argparse
import cv2

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-v", "--video", help = "path to the (optional) video file")
args = vars(ap.parse_args())

# if the video path was not supplied, grab the reference to the
# camera
if not args.get("video", False):
    camera = cv2.VideoCapture(0)

# otherwise, load the video
else:
    camera = cv2.VideoCapture(args["video"])

```

Again, we'll start our by importing the packages we need. I've placed our `simple_barcode_detection` function in the `pyimagesearch` module for organizational purposes. Then, we import `argparse` for parsing command line arguments and `cv2` for our OpenCV bindings.

Lines 6-9 handle parsing our command line arguments. We'll need a single (optional) switch, `--video`, which is the path to the video file on desk that contains the barcodes we want to detect.

Note: This switch is useful for running the example videos provided in the source code for this blog post. By omitting this switch you will be able to utilize the webcam of your laptop or desktop.

Lines 13-18 handle grabbing a reference to our `camera` feed. If we did not supply a `--video` switch, then we grab a handle to our webcam on **Lines 13 and 14**. However, if we did provide a path to a video file, then the file reference is grabbed on **Lines 17 and 18**.

Now that the setup is done, we can move on to applying our actual barcode detection module:

```

# keep looping over the frames
while True:
    # grab the current frame
    (grabbed, frame) = camera.read()

    # check to see if we have reached the end of the
    # video
    if not grabbed:
        break

    # detect the barcode in the image
    box = simple_barcode_detection.detect(frame)

    # if a barcode was found, draw a bounding box on the frame
    cv2.drawContours(frame, [box], -1, (0, 255, 0), 2)

    # show the frame and record if the user presses a key
    cv2.imshow("Frame", frame)
    key = cv2.waitKey(1) & 0xFF

    # if the 'q' key is pressed, stop the loop
    if key == ord("q"):
        break

    # cleanup the camera and close any open windows
    camera.release()
    cv2.destroyAllWindows()

```

On **Line 21** we start looping over the frames of our video — this loop will continue to run until (1) the video runs out of frames or (2) we press the q key on our keyboard and break from the loop.

We query our `camera` on **Line 23**, which returns a 2-tuple. This 2-tuple contains a boolean, `grabbed`, which indicates if the frame was successfully grabbed from the video file/webcam, and `frame`, which is the actual frame from the video.

If the frame was not successfully grabbed (such as when we reach the end of the video file), we break from the loop on **Lines 27 and 28**.

Now that we have our frame, we can utilize our barcode detection module to detect a barcode in it — this handled on **Line 31** and our bounding box is returned to us.

We draw our resulting bounding box around the barcoded region on **Line 34** and display our frame to our screen on **Lines 37 and 38**.

Finally, **Lines 41 and 42** handle breaking from our loop if the `q` key is pressed on our keyboard while **Lines 45 and 46** cleanup pointers to our camera object.

So as you can see, there isn't much to our driver script!

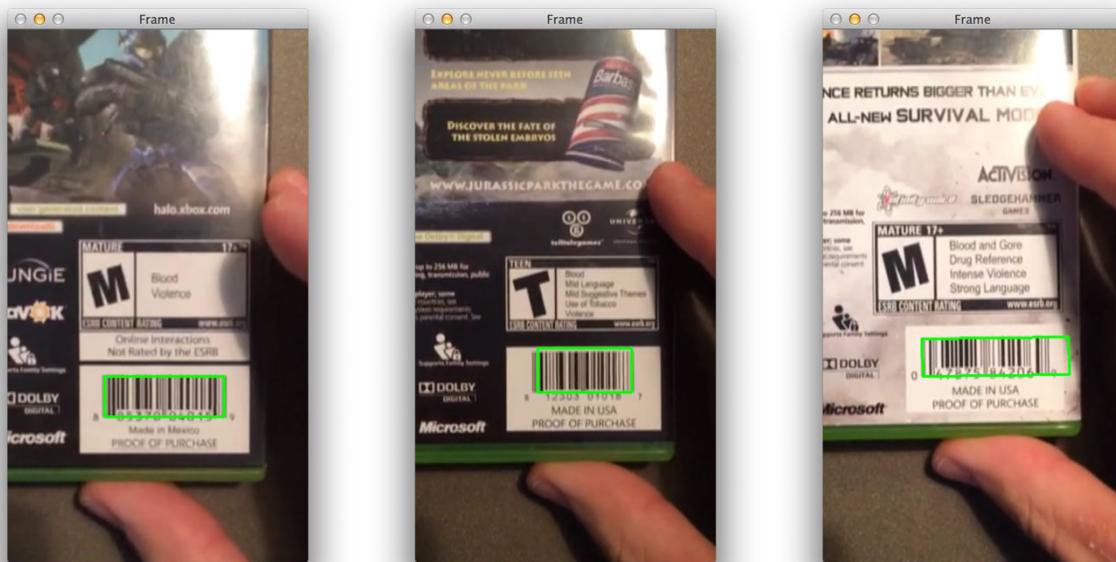
Let's put this code into action and look at some results.

Successful barcode detections in video

Let's try some examples. Open up a terminal and issue the following command:

```
$ python detect_barcode.py --video video/video_games.mov
```

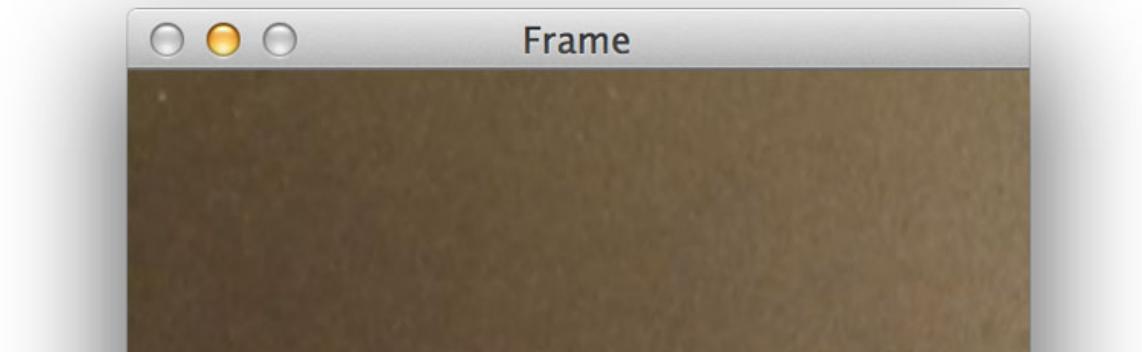
The video at the top of this post demonstrates the output of our script. And below is a screenshot for each of the three successful barcode detections on the video games:



Let's see if we can detect barcodes on a clothing coupon:

```
$ python detect_barcode.py --video video/coupon.mov
```

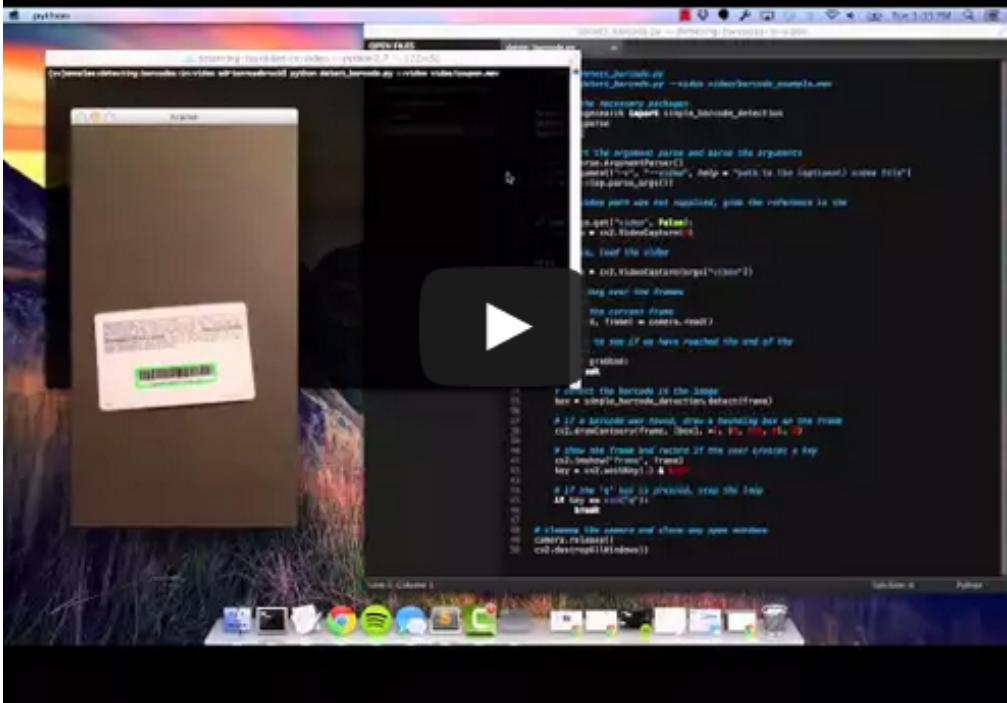
Here's an example screenshot from the video stream:





And the full video of the output:

Using OpenCV and Python to Detect Coupon Barcodes in Video Streams



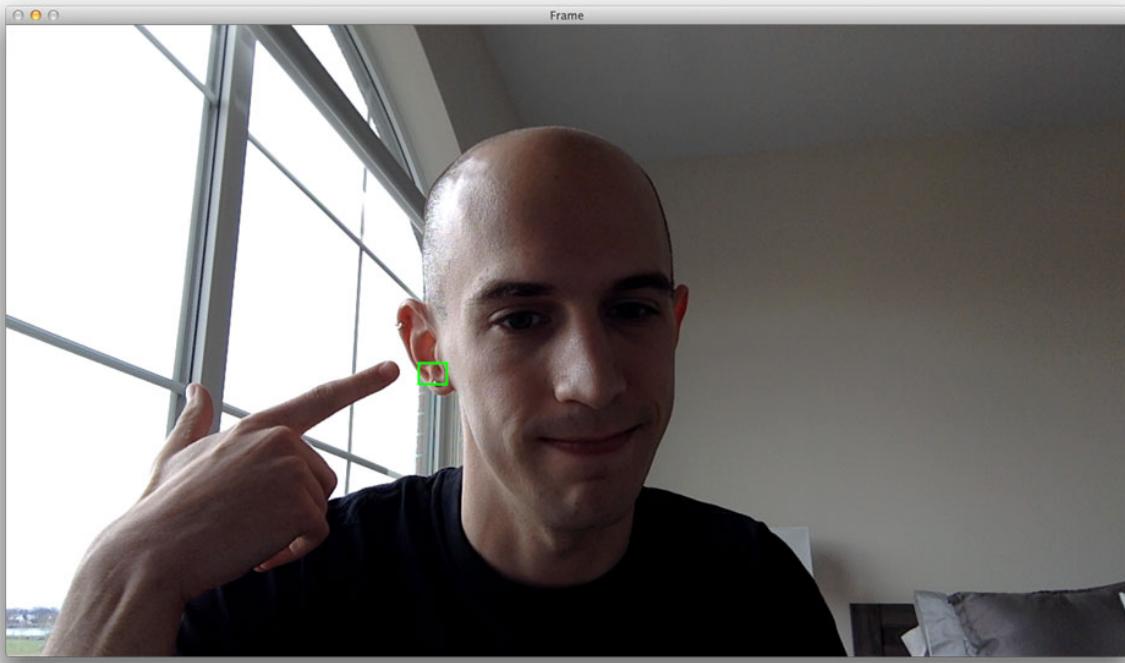
Of course, like I said that this approach only works in optimal conditions (see the following section for a detailed description of limitations and drawbacks).

Here is an example of where the barcode detection did not work:



In this case, the barcode is too far away from the camera and there are too many “distractions” and “noise” in the image, such as large blocks of text on the video game case.

This example is also clearly a failure, I just thought it was funny to include:



Again, this simple implementation of barcode detection will not work in all cases. It is not a robust solution, but rather an example of how simple image processing techniques can give surprisingly good results, provided that assumptions in the following section are met.

Limitations and Drawbacks

So as we've seen in this blog post, our approach to detecting barcodes in images works well — *provided we make some assumptions regarding the videos we are detecting barcodes in.*

The **first assumption** is that we have a static camera that is “looking down” on the barcode at a 90-degree angle. This will ensure that the gradient region of the barcoded image will be found by our simple barcode detector.

The **second assumption** is that our video has a “close up” of the barcode, meaning that we are holding our smartphones directly overtop of the barcode, and not holding the barcode far away from the lens. The farther we move the barcode away from the camera, the less successful our simple barcode detector will be.

So how do we improve our simple barcode detector?

Great question.

Christoph Oberhofer has provided a [great review on how robust barcode detection is done in QuaggaJS](#). If you're looking for the next steps, be sure to check his post!

Summary

In this blog post we built upon our previous codebase to detect barcodes in images. We extended our code into two components:

- A component to detect barcodes in individual frames of a video.
- And a “driver” component that accesses the video feed of our camera or video file.

We then applied our simple barcode detector to detect barcodes in video.

However, our approach does make some assumptions:

- The first assumption is that we have a static camera view that is “looking down” on the barcode at a 90-degree angle.
- And the second assumption is that we have a “close up” view of the barcode without other interfering objects or noise in the view of the frame.

In practice, these assumptions may or may-not be guaranteeable. It all depends on the application you are developing!

At the very least I hope that this article was able to demonstrate to you some of the basics of image processing, and how these image processing techniques can be leveraged to build a simple barcode detector in video using Python and OpenCV.

Downloads:

To download the source code to this article, just use this link: <http://www.pyimagesearch.com/wp-content/uploads/2014/12/detecting-barcodes-in-video.zip>