1.11.4: Contour approximation

Topic Progress: (https://gurus.pyimagesearch.com/topic/finding-and-drawing-contours/) (https://gurus.pyimagesearch.com/topic/simple-contour-properties/) (https://gurus.pyimagesearch.com/topic/advanced-contour-properties/) (https://gurus.pyimagesearch.com/topic/contour-approximation/) (https://gurus.pyimagesearch.com/topic/sorting-contours/)

← Back to Lesson (https://gurus.pyimagesearch.com/lessons/contours/)

Contour approximation is perhaps one of my favorite topics to discuss when it comes to working with objects in images — it is such a powerful (and perhaps even overlooked) technique that you can leverage extensively when building real-world computer vision applications. In fact, you'll see how we can use contour approximations to build the basics of a mobile document scanner later in this lesson.

Objectives:

This topic has three primary objectives:

1. Understand (at a very high level) the process of contour approximation.
2. Apply contour approximation to distinguish between circles and squares/rectangles.
3. Use contour approximation to find "documents" in images.

Contour Approximation

As the name suggests, *contour approximation* is an algorithm for reducing the number of points in a curve with a reduced set of points — thus, an approximation. This algorithm is commonly known as the Ramer-Douglas-Peucker algorithm, or simply: the split-and-merge algorithm.

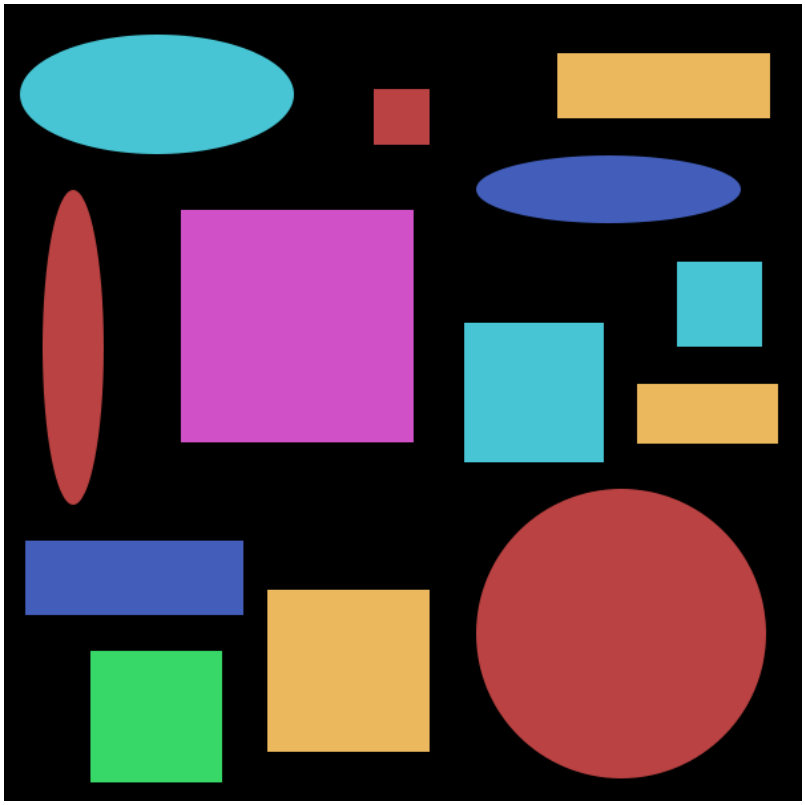
The general assumption of this algorithm is that a curve can be approximated by a series of short line segments. And we can thus *approximate* a given number of these line segments to reduce the number of points it takes to construct a curve.

Overall, the resulting approximated curve consists of a subset of points that were defined by the original curve.

The actual algorithm itself is already implemented in OpenCV via the `cv2.approxPolyDP` function, so luckily we do not have to implement the algorithm by hand. However, the recursive algorithm is fairly straightforward and I would definitely suggest giving the excellent Wikipedia article (http://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm) on the approximation algorithm a quick read.

Before we get too far, I want to take a look at an example image so we can understand the utility of contour approximation. Then, after this little example image and source code, I'll return to the contour approximation algorithm and discuss the parameters that we need to pay attention to.

Take a look at the example image below — our goal here is to detect *only* the rectangles, while *ignoring* the circles/ellipses:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contours_circles_and_squares.png)

FIGURE 1: IN THIS EXAMPLE PROBLEM, OUR GOAL IS TO DEFINE A METHOD TO DETECT THE RECTANGLES AND IGNORE THE CIRCLES/ELLIPSES.

So how are we going to accomplish this?

We could use simple contour properties, like the ones discussed in previous sections. But let's instead solve the problem using contour approximations:

approx_simple.py Python

```
1  # import the necessary packages
2  import cv2
3  import imutils
4
5  # load the the cirles and squares image and convert it to grayscale
6  image = cv2.imread("images/circles_and_squares.png")
7  gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
8
9  # find contours in the image
10 cnts = cv2.findContours(gray.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
11 cnts = imutils.grab_contours(cnts)
12
13 # loop over the contours
14 for c in cnts:
15     # approximate the contour
16     peri = cv2.arcLength(c, True)
17     approx = cv2.approxPolyDP(c, 0.01 * peri, True)
18
19     # if the approximated contour has 4 vertices, then we are examining
20     # a rectangle
21     if len(approx) == 4:
22         # draw the outline of the contour and draw the text on the image
23         cv2.drawContours(image, [c], -1, (0, 255, 255), 2)
24         (x, y, w, h) = cv2.boundingRect(approx)
25         cv2.putText(image, "Rectangle", (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX,
26             0.5, (0, 255, 255), 2)
27
28 # show the output image
29 cv2.imshow("Image", image)
30 cv2.waitKey(0)
```


We'll start off by loading our `circles_and_squares.png` image from disk and converting it to grayscale (**Lines 6 and 7**). And now that we have our grayscale image, we can find the contours of the shapes (**Line 10**).

Next up, we'll loop over the contours one-by-one on **Line 14**.

Approximating the actual contour *c* is handled on **Lines 16 and 17**. First, we need to compute the actual *perimeter* of the contoured region. And once we have the length of the perimeter, we can use it to approximate it by making a call to `cv2.approxPolyDP` on **Line 17**. Here we are telling OpenCV that we want a special ϵ value to be 1% of the original contour perimeter.

If you're confused about this ϵ value, don't worry — I'll be explaining it in more depth once we are done with this example.

Remember, in our toy example image above, our goal is to *detect the actual rectangles* while *ignoring the circles/ellipses*.

So let's take a second to consider if we can exploit the geometry of this problem.

A rectangle has 4 sides. And a circle has no sides. Or, in this case, since we need to represent a circle as a series of points: a circle is composed of many, *many* tiny line segments — far more than the 4 sides that compose a rectangle.

So if we ***approximate the contour*** and then ***examine the number of points within the approximated contour***, we'll be able to determine if the contour is a rectangle or not! In fact, that's exactly what **Lines 16 and 17** are doing.

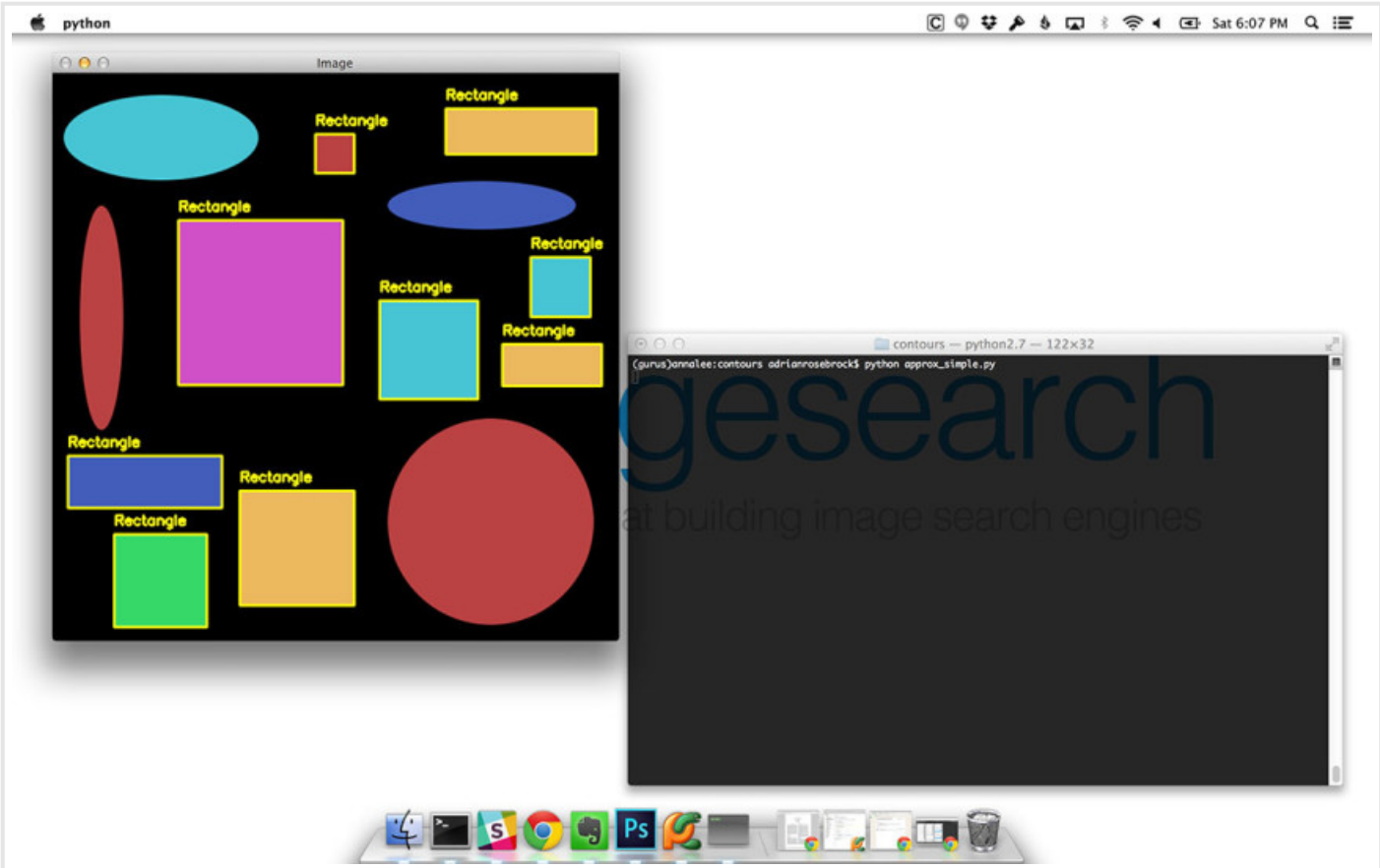
Once we have the approximated contour, we check the `len` (i.e. the length, or number of entries in the list) to see how many vertices (i.e. points) our approximated contour has. If our approximated contour has a four vertices, we can thus mark it as a rectangle, which is handled by **Lines 21-26**.

Finally, we display our output image on **Lines 29 and 30**.

To see our example script in action, open up your terminal, navigate to where your source code resides, and execute the following command:

```
approx_simple.py                                                    Python
1 $ python approx_simple.py
```

What you should see is the following image:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contours_simple_approx.jpg)

FIGURE 2: DISTINGUISHING BETWEEN RECTANGLES AND CIRCLES/ELLIPSES USING ONLY CONTOUR APPROXIMATIONS.

Feedback

As you can see, only the rectangles and squares have been outlined in yellow with the *"Rectangle"* text placed above them. Meanwhile, the circles and ellipses have been entirely ignored.

So at this point you're probably scratching your head and wondering how I knew to define the ϵ value of `cv2.polyApproxDP` to be 1% of the original contour perimeter — and I would definitely sympathize. That's certainly a point of confusion, especially when I first learned about contour approximation. So let's chat about this parameter a little more and try to clear up any confusion:

To control the level of tolerance for the approximation, we need to define a ϵ (epsilon) value. In practice, we define this ϵ relative to the *perimeter* of the shape we are examining. Commonly, we'll define ϵ as some percentage (usually between 1-5%) of the original contour perimeter.

This is because the internal contour approximation algorithm is looking for points to ***discard***. The larger the ϵ value is, the more points will be discarded. Similarly, the smaller the ϵ value is, the more points will be ***kept.***

So the question becomes: what's the *optimal* value for ϵ? And how do we go about defining it?

It's very clear that an ϵ value that will work well for some shapes will not work well for others (larger shapes versus smaller shapes, for instance). This means that we can't simply hardcode an ϵ value into our code — it must be computed dynamically based on the individual contour.

Thus, we define ϵ *relative to the perimeter length* so we understand how large the contour region actually is. Doing this ensures that we achieve a ***consistent*** approximation for all shapes inside the image.

And like I mentioned above, it's typical to use roughly 1-5% of the original contour perimeter length for a value of ϵ. Anything larger, and you'll be over-approximating your contour to almost a single straight line. Similarly, anything smaller and you won't be doing much of an actual approximation.

This was a nice theoretical example, but let's apply contour approximation to an actual ***real world*** problem.

In this second, more advanced example, our goal is to utilize contour approximation to find the sales receipt in the following image:

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contours_receipt_original.jpg)

FIGURE 3: GIVEN THIS IMAGE, OUR GOAL IS TO FIND THE ACTUAL RECEIPT IN THE IMAGE AND DRAW A 4-POINT BOUNDING BOX AROUND IT.

In fact, some of you may remember this image from the original PyImageSearch blog post on building a kick-ass mobile document scanner (http://www.pyimagesearch.com/2014/09/01/build-kick-ass-mobile-document-scanner-just-5-minutes/).

As you can see from the image above, our receipt is not exactly laying flat. It has some folds and wrinkles in it. So it's certainly not a perfect rectangle. Which leads us to the question:

If the receipt is not a perfect rectangle, how are we going to find the actual receipt in the image?

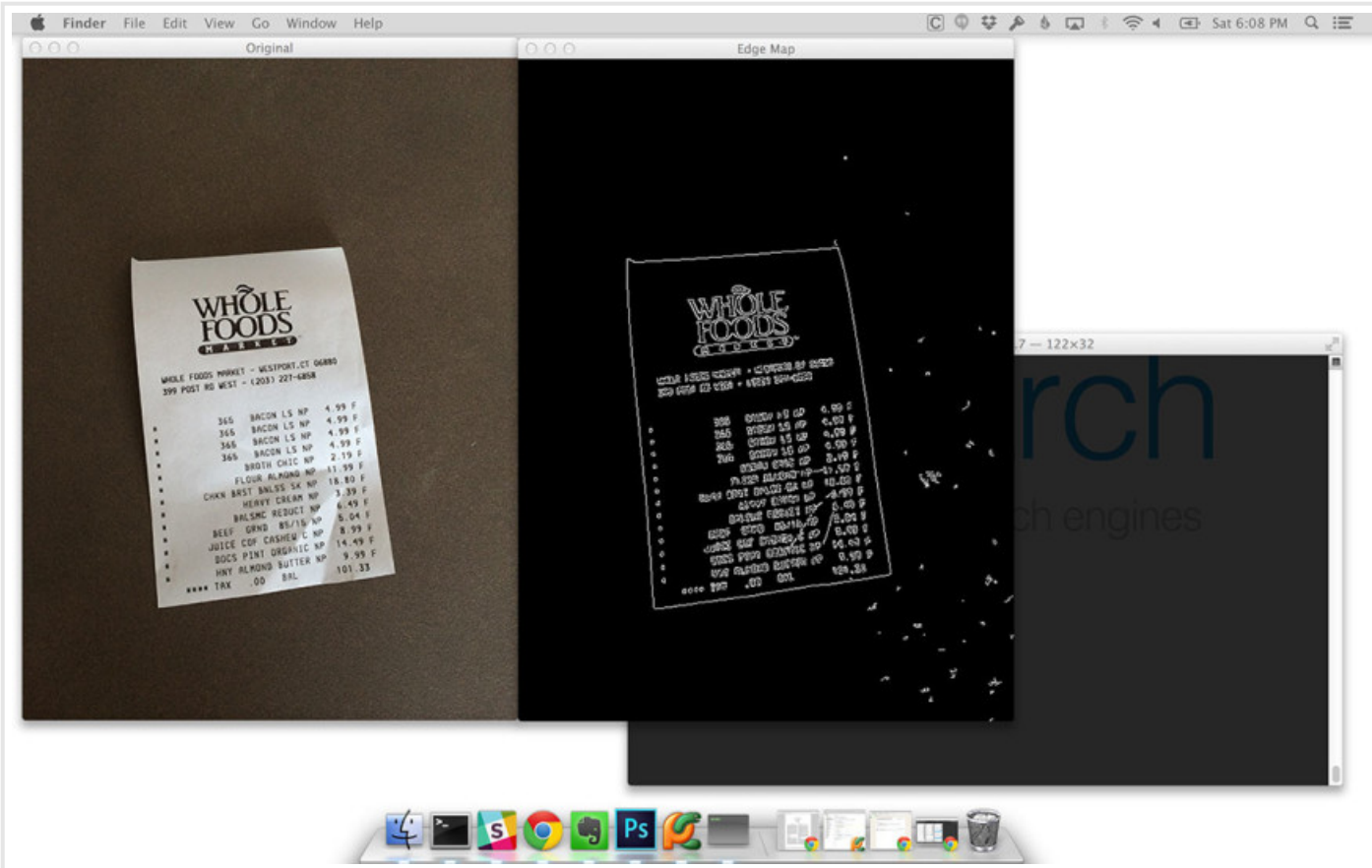
Simple: we'll just apply the techniques we learned from the earlier shapes example.

A receipt *looks* like a rectangle, after all — even though it's not a perfect rectangle. So if we apply contour approximation and look for *rectangle-like* regions, I'm willing to bet that we'll be able to find the receipt in the image:

approx_realworld.py — Python

```
1  # import the necessary packages
2  import cv2
3  import imutils
4
5  # load the receipt image, convert it to grayscale, and detect
6  # edges
7  image = cv2.imread("images/receipt.png")
8  gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
9  edged = cv2.Canny(gray, 75, 200)
10
11 # show the original image and edged map
12 cv2.imshow("Original", image)
13 cv2.imshow("Edge Map", edged)
```

We'll go ahead and load our receipt image off disk on **Line 7**, convert it to grayscale on **Line 8**, and detect edges in the grayscale image on **Line 9**. After applying edge detection, our receipt looks like this:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contours_receipt_edgemap.jpg)

FIGURE 4: DETECTING EDGES IN OUR SALES RECEIPT.

Awesome. We can clearly see the outline of the receipt now. But we're also getting a lot of other stuff we aren't interested in. For example, we're getting a lot of noise from the shadowing in the lower-right hand corner of the image. We're also getting the outlines of all the actual letters and characters on the receipt itself.

So how in the *hell* are we going to disregard all this noise and find ***only*** the receipt outline?

The answer is a two-step process. The first step is to sort the contours by their size, keeping only the largest ones (a topic that I'll cover in more detail in the next section), and the second step is to apply contour approximation.

Let's see how these two steps are done:

approx_realworld.py — Python

```
15 # find contours in the image and sort them from largest to smallest,
16 # keeping only the largest ones
17 cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
18 cnts = imutils.grab_contours(cnts)
19 cnts = sorted(cnts, key=cv2.contourArea, reverse=True)[:7]
20
21 # loop over the contours
22 for c in cnts:
23     # approximate the contour and initialize the contour color
24     peri = cv2.arcLength(c, True)
25     approx = cv2.approxPolyDP(c, 0.01 * peri, True)
26
27     # show the difference in number of vertices between the original
28     # and approximated contours
29     print("original: {}, approx: {}".format(len(c), len(approx)))
30
31     # if the approximated contour has 4 vertices, then we have found
32     # our rectangle
33     if len(approx) == 4:
34         # draw the outline on the image
35         cv2.drawContours(image, [approx], -1, (0, 255, 0), 2)
36
37 # show the output image
38 cv2.imshow("Output", image)
39 cv2.waitKey(0)
```

We'll start by finding contours in the edge map image on **Line 17**. We'll then ***sort*** these contours from largest-to-smallest (keeping only the 7 largest ones) by using a combination of the built-in Python `sorted` function and the `cv2.contourArea` function (**Line 19**).

Now that we have only the 7 largest contours in the image, we start looping over them individually on **Line 22**.

And just as in the first simple shapes example above, we apply contour approximation on **Lines 24 and 25**. We'll set our ϵ value to be 1% of the original contour perimeter length.

Line 29 then prints out the number of points in the ***original*** contour along with the number of points in the ***approximated*** contour — as we'll see in a few seconds, these are both really interesting properties to investigate.

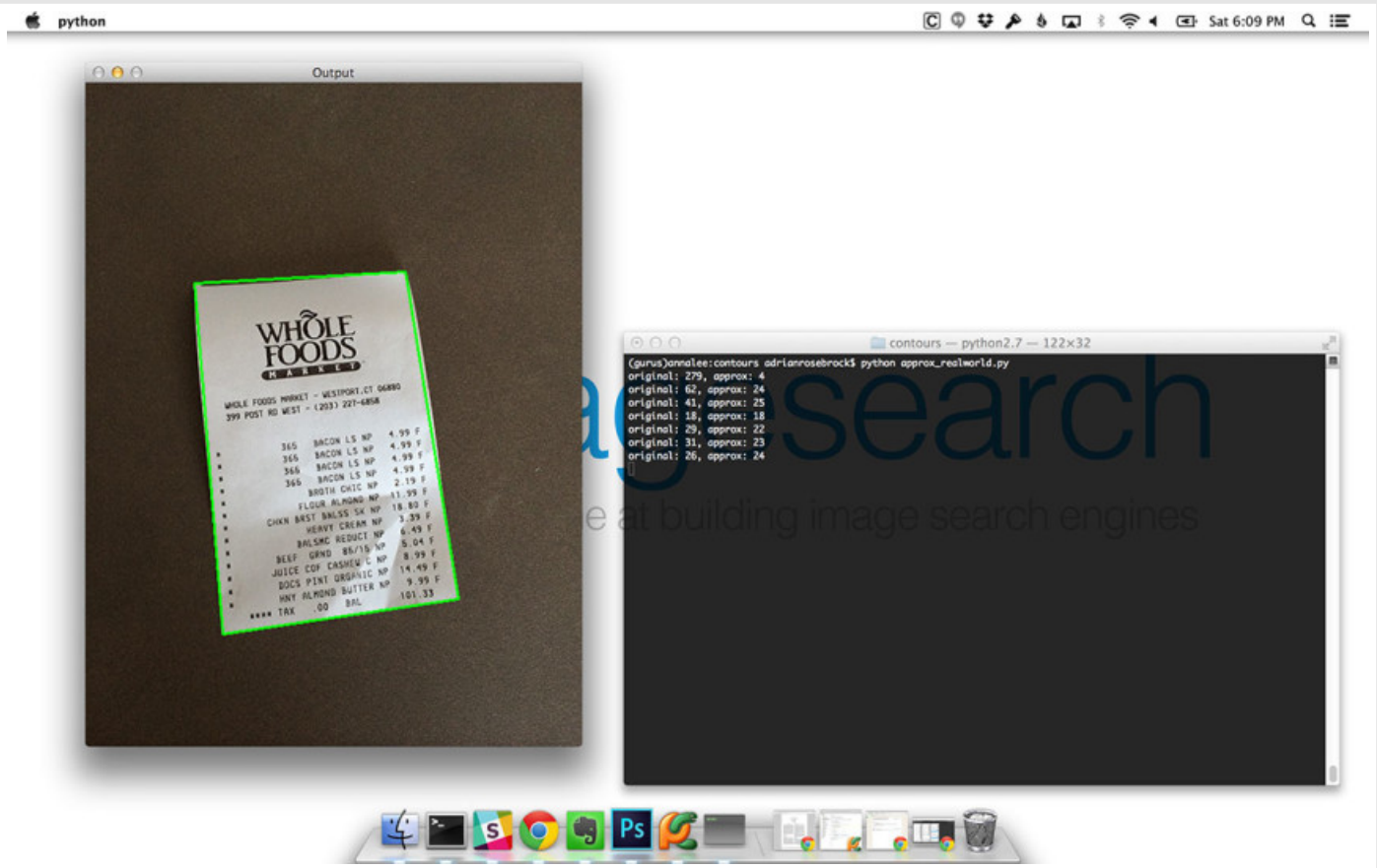
Finally, we make a check on **Line 33** to see if we have found our receipt — if the approximated contour consists of only 4 points, then we have found a rectangle. And since our receipt should be the largest rectangle in the image, we can thus assume we have found it.

Line 35 then draws the approximated contour (i.e. the rectangle) around the receipt in the original image.

To try our example out, just navigate to the source code for this example and issue the following command:

```
approx_realworld.py                                                    Python
1 $ python approx_realworld.py
```

Your output image should look like this:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contours_receipt_output.jpg)

FIGURE 5: WE HAVE SUCCESSFULLY IDENTIFIED THE RECEIPT IN THE IMAGE USING ONLY SIMPLE CONTOUR APPROXIMATIONS.

As can be seen by the image on the *left* we have successfully found our receipt. But more interestingly, take a look at the output of our terminal on the *right*.

The original receipt contour had over **279 points** prior to approximation — that original shape was by no means a rectangle! However, by applying contour approximation we were able to sift through all the noise and reduce those **279 points** down to **4 points**. And since our 4 points formed a rectangle, we can thus label the region as our receipt.

So as you can see, contour approximations are a powerful little tool to have in your tool-belt. Not only did they allow us to distinguish between rectangles and circles/ellipses, but they also allowed us to help solve a **real-world** computer vision problem by finding an actual receipt in an image!

If you liked this particular example, be sure to head to the PyImageSearch blog and read up on the other necessary steps to build a mobile document scanner (http://www.pyimagesearch.com/2014/09/01/build-kick-ass-mobile-document-scanner-just-5-minutes/) — contour approximation is only just one piece of the puzzle!

Summary

This lesson discussed, at a high level, the process of contour approximation. We discovered at that contour approximation is simply finding an optimal set of sub-lines to represent the outline of the object. Most importantly, we learned that it's crucial to use the perimeter of the object to aide us in determine the ϵ value used by the contour approximation algorithm.

We then provided two examples of contour approximation. First was a simple example where we used contours to distinguish between circles and rectangles/squares in an image. The second example was much more real-world and detailed how we can use contours to find documents in images.

Downloads:

Download the Code (https://gurus.pyimagesearch.com/protected/code/computer_vision_basics/cont

Feedback

Quizzes		Status
1	Contour Approximation Quiz (https://gurus.pyimagesearch.com/quizzes/contour-approximation-quiz/)	

← Previous Topic (https://gurus.pyimagesearch.com/topic/advanced-contour-properties/) Next Topic → (https://gurus.pyimagesearch.com/topic/sorting-contours/)

Course Progress

Ready to continue the course?

Click the button below to **continue your journey to computer vision guru**.

I'm ready, let's go! (/pyimagesearch-gurus-course/)

Resources & Links

- PyImageSearch Gurus Community (https://community.pyimagesearch.com/)
- PyImageSearch Virtual Machine (https://gurus.pyimagesearch.com/pyimagesearch-virtual-machine/)
- Setting up your own Python + OpenCV environment (https://gurus.pyimagesearch.com/setting-up-your-python-opencv-development-environment/)
- Course Syllabus & Content Release Schedule (https://gurus.pyimagesearch.com/course-syllabus-content-release-schedule/)
- Member Perks & Discounts (https://gurus.pyimagesearch.com/pyimagesearch-gurus-discounts-perks/)
- Your Achievements (https://gurus.pyimagesearch.com/achievements/)
- Official OpenCV documentation (http://docs.opencv.org/index.html)

Your Account

- Account Info (https://gurus.pyimagesearch.com/account/)
- Support (https://gurus.pyimagesearch.com/contact/)
- Logout (https://gurus.pyimagesearch.com/wp-login.php?action=logout&redirect_to=https%3A%2F%2Fgurus.pyimagesearch.com%2F&_wpnonce=5736b21cae)

Search

Feedback

/