1.9: Thresholding

In the last lesson, I introduced the topic of lighting conditions and color spaces (https://gurus.pyimagesearch.com/lessons/lighting-and-color-spaces/) with a discussion of ID My Pill, an app and web API I created that allows prescription pills to be identified in a snap of a smartphone camera.

Inside that lesson I mentioned that adequate lighting conditions are ***absolutely critical*** in successfully identifying a medication — if the lighting conditions are less than adequate, the identification will fail.

Feedback

There are many reasons why lighting conditions can hurt pill identification performance. Lighting conditions could "wash out" the imprint on the pill. They could cast shadows on the pill itself. And in many cases, we may not be able to *segment* the pill from the background of the image.

Thresholding is one of the most common (and basic) segmentation techniques in computer vision and it allows us to separate the *foreground* (i.e. the objects that we are interested in) from the *background* of the image.

Thresholding comes in many forms. We have simple thresholding where we manually supply parameters to segment the image — this works extremely well in controlled lighting conditions where we can ensure high contrast between the foreground and background of the image.

We also have methods such as Otsu's thresholding that attempt to be more *dynamic* and automatically compute the optimal threshold value based on the input image.

And we have *adaptive thresholding* which, instead of trying to threshold an image *globally* using a single value, instead breaks the image down into smaller pieces, and thresholds each of these pieces *separately* and *individually*.

We'll be discussing all three of these types of thresholding methods inside this lesson — and I must say, thresholding is one of my favorite computer vision topics. The techniques are fairly straightforward and intuitive, but as I'll demonstrate later in this lesson, we can use thresholding to segment the license plate characters from the license plate itself!

Objectives:

By the time you are finished with this lesson you should:

1. Be able to define what thresholding is.
2. Understand *simple thresholding* and why a thresholding value *T* must be manually provided.
3. Grasp *Otsu's thresholding* method.
4. Comprehend the importance of *adaptive thresholding* and why it's useful in situations where lighting conditions cannot be controlled.

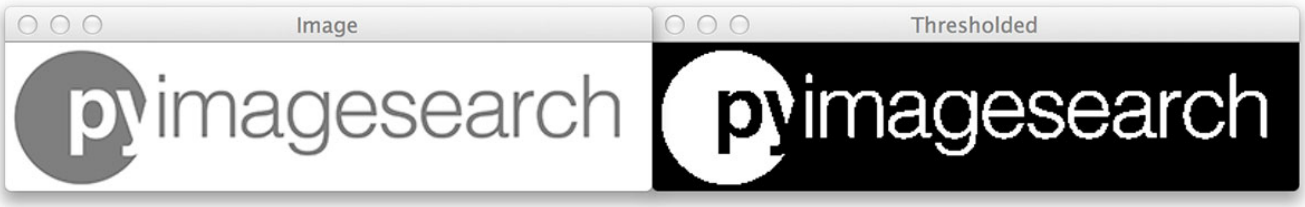
What is thresholding?

Feedback

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

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

For example, take a look at the (grayscale) PyImageSearch logo below and its thresholded counterpart:



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

FIGURE 1: THRESHOLDING THE PYIMAGESEARCH LOGO. ON THE *LEFT* WE HAVE THE ORIGINAL IMAGE, AND ON THE *RIGHT* WE HAVE ITS THRESHOLDED COUNTERPART.

On the *left* we have the original PyImageSearch logo that has been converted to grayscale. And on the *right* we have the thresholded, binary representation of the PyImageSearch logo.

To construct this thresholded image I simply set my threshold value *T=225*. That way, all pixels *p* in the logo where *p* < *T* are set to *255*, and all pixels *p* >= *T* are set to *0*.

By performing this thresholding I have been able to segment the PyImageSearch logo from the background.

Normally, we use thresholding to focus on objects or areas of particular interest in an image. In the examples in the lesson below, we will be using thresholding to detect coins in images, segment the pieces of the OpenCV logo, and separate license plate letters and characters from the license plate itself.

Simple Thresholding

Applying simple thresholding methods requires human intervention. We must specify a threshold value *T*. All pixel intensities below *T* are set to 255. And all pixel intensities greater than *T* are set to 0 (like in the example above).

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

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

simple_thresholding.py Python

```python
1  # import the necessary packages
2  import argparse
3  import cv2
4  
5  # construct the argument parser and parse the arguments
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required=True, help="Path to the image")
8  args = vars(ap.parse_args())
9  
10 # load the image, convert it to grayscale, and blur it slightly
11 image = cv2.imread(args["image"])
12 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
13 blurred = cv2.GaussianBlur(gray, (7, 7), 0)
14 cv2.imshow("Image", image)
```

On **Lines 1-11** we import our packages, parse our arguments, and load our image. From there, we convert the image from the RGB color space to grayscale on **Line 12**.

At this point, we apply Gaussian blurring on **Line 13** with a *7×7* kernel. Applying Gaussian blurring helps remove some of the high frequency edges in the image that we are not concerned with and allow us to obtain a more "clean" segmentation.

Now, let's go ahead and apply the actual thresholding:

```python
simple_thresholding.py                                                     Python
16 # apply basic thresholding -- the first parameter is the image
17 # we want to threshold, the second value is our threshold check
18 # if a pixel value is greater than our threshold (in this case,
19 # 200), we set it to be BLACK, otherwise it is WHITE.
20 (T, threshInv) = cv2.threshold(blurred, 200, 255, cv2.THRESH_BINARY_INV)
21 cv2.imshow("Threshold Binary Inverse", threshInv)
22
23 # using normal thresholding (rather than inverse thresholding),
24 # we can change the last argument in the function to make the coins
25 # black rather than white.
26 (T, thresh) = cv2.threshold(blurred, 200, 255, cv2.THRESH_BINARY)
27 cv2.imshow("Threshold Binary", thresh)
28
29 # finally, we can visualize only the masked regions in the image
30 cv2.imshow("Output", cv2.bitwise_and(image, image, mask=threshInv))
31 cv2.waitKey(0)
```

After the image is blurred, we compute the thresholded image on **Line 20** using the `cv2.threshold` function. This method requires four arguments. The first is the grayscale image that we wish to threshold. We supply our blurred image here.

Then, we manually supply our *T* threshold value. We use a value of *T=200*.

Our third argument is the *output value* applied during thresholding. Any pixel intensity *p* that is greater than *T* is set to zero and any *p* that is less than *T* is set to the *output value*:

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxval} & \text{otherwise} \end{cases}$$

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

FIGURE 2: THE 'IF' STATEMENT TEST FOR CV2.THRESH_BINARY_INV

In our example, any pixel value that is greater than *200* is set to *0*. Any value that is less than *200* is set to *255*.

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

The `cv2.threshold` function then returns a tuple of 2 values: the first, *T*, is the threshold value. In the case of simple thresholding, this value is trivial since we manually supplied the value of *T* in the first place. But in the case of Otsu's thresholding where *T* is dynamically computed for us, it's nice to have that value. The second returned value is the thresholded image itself.

But what if we wanted to perform the **reverse** operation, like this:

$$\mathtt{dst}(x,y) = \begin{cases} \mathtt{maxval} & \text{if } \mathtt{src}(x,y) > \mathtt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

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

FIGURE 3: THE 'IF' STATEMENT TEST FOR CV2.THRESH_BINARY

What if wanted to set all pixels *p* greater than *T* to the output value? Is that possible?

Of course! And there are two ways to do it.

The first method is to simply take the bitwise NOT of the output threshold image. But that adds an extra line of code.

Instead, we can just supply a different flag to the `cv2.threshold` function. On **Line 26** we apply a different thresholding method by supplying `cv2.THRESH_BINARY` .

In most cases you are normally seeking your segmented objects to appear as *white* on a *black* background, hence using `cv2.THRESH_BINARY_INV` . But in the case that you want your objects to appear as *black* on a *white* background, be sure to supply the `cv2.THRESH_BINARY` flag.

The last task we are going to perform is to reveal the foreground objects in the image and hide everything else. Remember when we discussed masking (https://gurus.pyimagesearch.com/topic/masking/)? That will come in handy here.

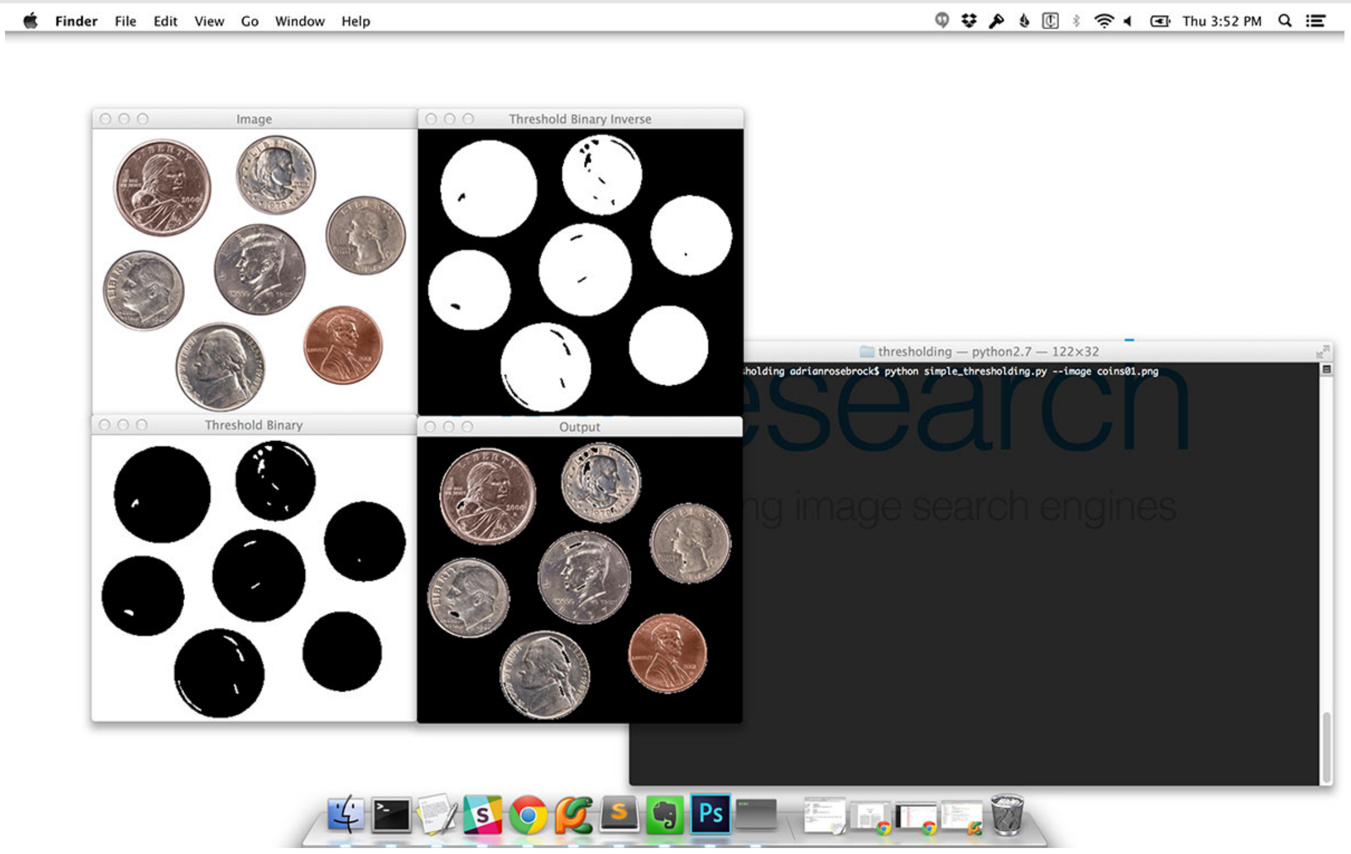
On **Line 30** we perform masking by using the `cv2.bitwise_and` function. We supply our original input image as the first two arguments, and then our inverted thresholded image as our mask. Remember, a mask only considers pixels in the original image where the mask is greater than zero.

So that was a fair chunk of code to explain — let's see what simple thresholding looks like in action. Open up your terminal, navigate to your source code, and execute the following command:

simple_thresholding.py Shell

```shell
$ python simple_thresholding.py --image coins01.png
```

You should then see the following output image:


Finder File Edit View Go Window Help
Thu 3:52 PM
Image
Threshold Binary Inverse
Threshold Binary
Output
thresholding — python2.7 — 122×32

```
holding adrianrosebrock$ python simple_thresholding.py --image coins01.png
```

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/simple_threshold_coins01.jpg)

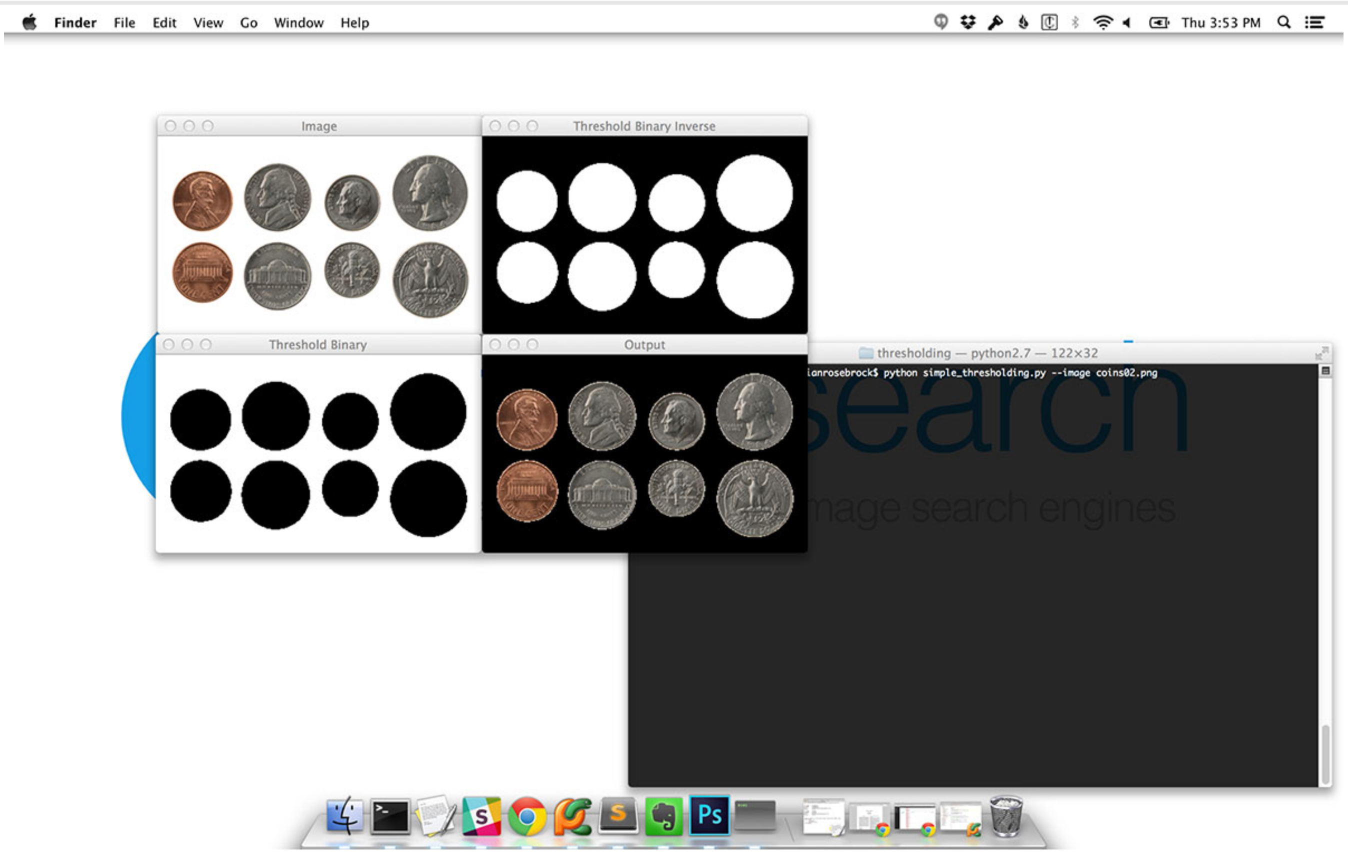
FIGURE 4: USING SIMPLE THRESHOLDING TO SEGMENT COINS (FOREGROUND) FROM THE BACKGROUND.

On the *top-left* we have our original input image. And on the *top-right* we have the segmented image using inverse thresholding, where the coins appear as *white* on a *black* background. Similarly, on the *bottom-left* we flip the thresholding method and now the coins appear as *black* on a *white* background. Finally, the *bottom-right* applies our bitwise AND with the threshold mask and we are left with just the coins in the image (no background).

Let's try a second image of coins:

simple_thresholding.py Shell
```
$ python simple_thresholding.py --image coins02.png
```

And the output:


Finder File Edit View Go Window Help
Thu 3:53 PM
Image
Threshold Binary Inverse
Threshold Binary
Output
thresholding — python2.7 — 122×32

```
ianrosebrock$ python simple_thresholding.py --image coins02.png
```

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/simple_threshold_coins02.jpg)

FIGURE 5: APPLYING SIMPLE THRESHOLDING TO A SECOND IMAGE.

Once again we are able to successfully segment the foreground of the image from the background.

But take a close look and compare the output of **Figure 4** and **Figure 5**. You'll notice that in **Figure 4** there are some coins that appear to have "holes" in them. This is because the thresholding test failed to pass — and thus we could not include that region of the coin in the output thresholded image. However, in **Figure 5** you'll notice that there *are no holes* — indicating that the segmentation is (essentially) perfect.

Note: *Realistically, this isn't a problem. All that really matters is that we are able to obtain the* contour or outline *of the coins. These small gaps inside the thresholded coin mask can be filled in using* *morphological operations (https://gurus.pyimagesearch.com/lessons/morphological-operations/)* *or* *contour methods (https://gurus.pyimagesearch.com/lessons/contours/)*.

So given that the thresholding worked perfectly in **Figure 5**, why did it not work just as perfectly in **Figure 4**?

The answer is simple: *lighting conditions.*

While subtle, these two images were indeed captured under different lighting conditions. And since we have manually supplied a thresholding value, there is no guarantee that this threshold value *T* is going to work from one image to the next in the presence of lighting changes.

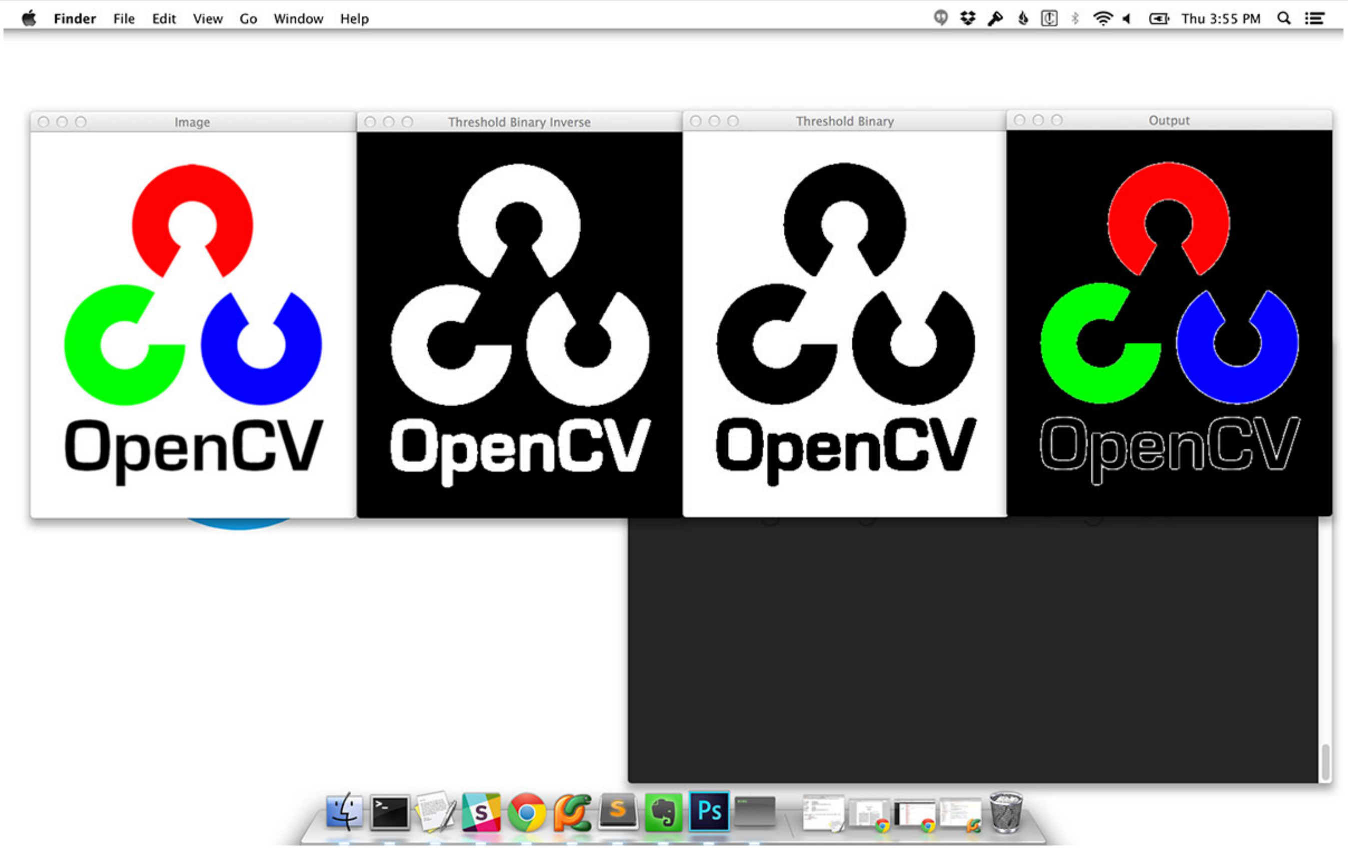
One method to combat this is to simply provide a threshold value *T* for each image you want to threshold. But that's a serious problem, especially if we want to our system to be *dynamic* and work under various lighting conditions.

The solution is to use methods such as *Otsu's method* and *adaptive thresholding* to aide us in obtaining better results.

But for the time being, let's look at one more example where we segment the pieces of the OpenCV logo:

```
simple_thresholding.py                                                    Shell
$ python simple_thresholding.py --image opencv_logo.png
```

And here is our output:



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

FIGURE 6: SEGMENTING THE OPENCV LOGO USING SIMPLE THRESHOLDING.

Notice how we have been able to segment the semi-circles of the OpenCV logo along with the "OpenCV" text itself from the input image. While this may not look very interesting, being able to segment an image into pieces is an extremely valuable skill to have. This will become more apparent when we dive into contours (https://gurus.pyimagesearch.com/lessons/contours/) and use them to quantify and identify different objects in an image.

But for the time being, let's move on to some more advanced thresholding techniques where we do not have to manually supply a value of *T*.

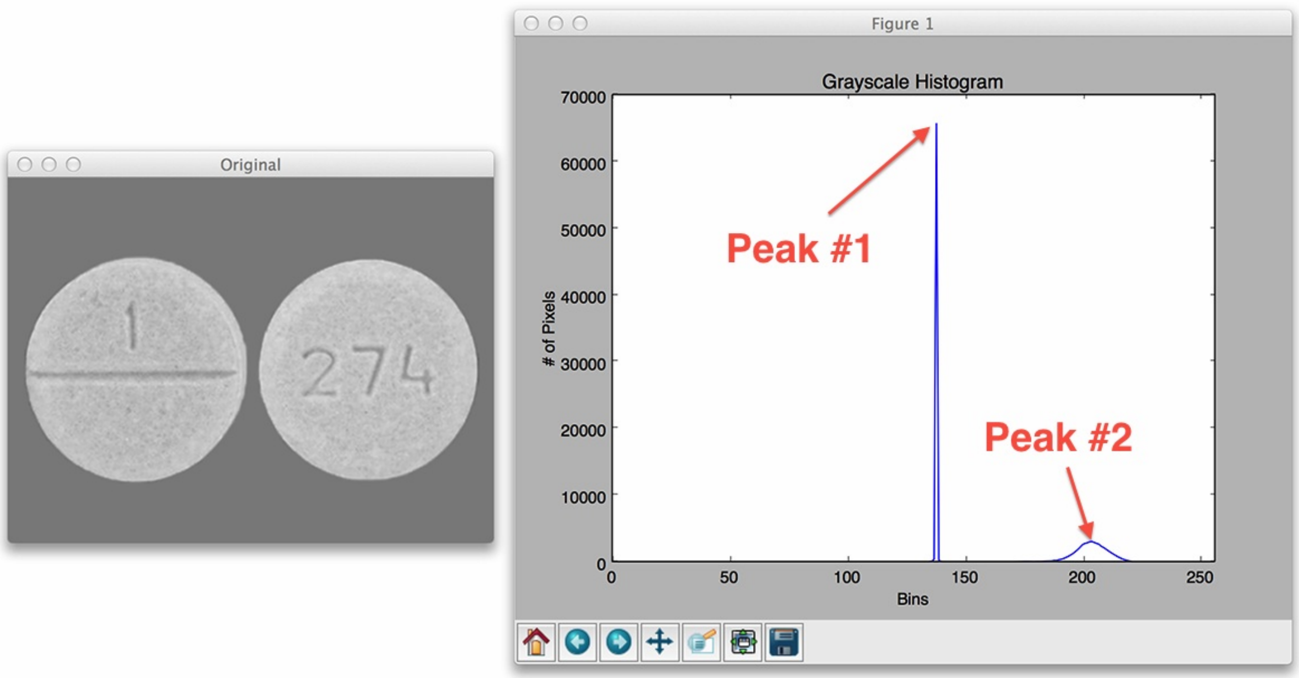
Otsu's Method

In the previous section on **Simple Thresholding** we needed to manually supply a threshold value of *T*. For simple images in controlled lighting conditions, it might be feasible for us to hardcode this value.

But in real-world conditions where we do not have any *a priori* knowledge of the lighting conditions, we actually automatically compute an optimal value of *T* using Otsu's method.

Otsu's method assumes that our image contains two classes of pixels: the *background* and the *foreground*. Furthermore, Otsu's method makes the assumption that the grayscale histogram of our pixel intensities of our image is *bi-modal*, which simply means that the histogram is two peaks.

For example, take a look at the following image of a prescription pill and its associated grayscale histogram:



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

FIGURE 7: A PILL IMAGE AND IT'S ASSOCIATED GRAYSCALE HISTOGRAM. NOTICE HOW THE HISTOGRAM CLEARLY HAS TWO PEAKS, AND IS THUS *BI-MODAL*.

Notice how the histogram clearly has two peaks — the first sharp peak corresponds to the uniform background color of the image, while the second peak corresponds to the pill region itself.

If the concept of histograms is a bit confusing to you right now, don't worry — we'll be covering them in more detail in **Module 1.11** (https://gurus.pyimagesearch.com/lessons/histograms/). But for the time being just understand that a histogram is simply a tabulation or a "counter" on the number of times a pixel value appears in the image.

Based on the grayscale histogram, Otsu's method then computes an optimal threshold value *T* such that the variance between the background and foreground peaks is minimal.

However, Otsu's method has no *a priori* knowledge of what pixels belong to the foreground and which pixels belong to the background — it's simply trying to optimally separate the peaks of the histogram.

It's also important to note that Otsu's method is an example of **global thresholding** — implying that a single value of *T* is computed for the **entire** image. In some cases, having a single value of *T* for an entire image is perfectly acceptable — but in other cases, this can lead to sub-par results, as we'll see when we try to segment license plate characters and letters from the license plate itself in the next section.

Let's go ahead and take a look at some code to perform Otsu's thresholding:

otsu_thresholding.py   Python

```python
1  # import the necessary packages
2  import argparse
3  import cv2
4
5  # construct the argument parser and parse the arguments
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required=True, help="Path to the image")
8  args = vars(ap.parse_args())
9
10 # load the image, convert it to grayscale, and blur it slightly
11 image = cv2.imread(args["image"])
12 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
13 blurred = cv2.GaussianBlur(gray, (7, 7), 0)
14 cv2.imshow("Image", image)
15
16 # apply Otsu's automatic thresholding -- Otsu's method automatically
17 # determines the best threshold value `T` for us
18 (T, threshInv) = cv2.threshold(blurred, 0, 255,
19     cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)
20 cv2.imshow("Threshold", threshInv)
21 print("Otsu's thresholding value: {}".format(T))
22
23 # finally, we can visualize only the masked regions in the image
24 cv2.imshow("Output", cv2.bitwise_and(image, image, mask=threshInv))
25 cv2.waitKey(0)
```

Just like the previous example in the **Simple Thresholding** section, **Lines 1-14** import our packages, parse our command line arguments, load our image from disk, convert it to grayscale, and then blur it slightly.

Applying Otsu's method is handled on **Line 18 and 19**, again using the `cv2.threshold` function of OpenCV.

We start by passing in the (blurred) image that we want to threshold. But take a look at the second parameter — this is supposed to be our threshold value *T*. So why are we setting it to zero? Remember that Otsu's method is going to automatically compute the optimal value of *T* for us. We could technically specify any value we wanted for this argument; however, I like to supply a value of *0* as a type of "don't care" parameter.

The third argument is the output value of the threshold, provided the given pixel passes the threshold test.

The last argument is one we need to pay *extra special* attention to. Previously, we had supplied values of `cv2.THRESH_BINARY` or `cv2.THRESH_BINARY_INV` depending on what type of thresholding we wanted to perform. But now we are passing in a second flag that is logically OR'd with the previous method. Notice that this method is `cv2.THRESH_OTSU` , which obviously corresponds to Otsu's thresholding method.

The `cv2.threshold` function will again return a tuple of 2 values for us: the threshold value *T* and the thresholded image itself. In the **Simple Thresholding** section, the value of *T* returned was redundant and irrelevant — we already knew this value of *T* since we had to manually supply it.

Feedback

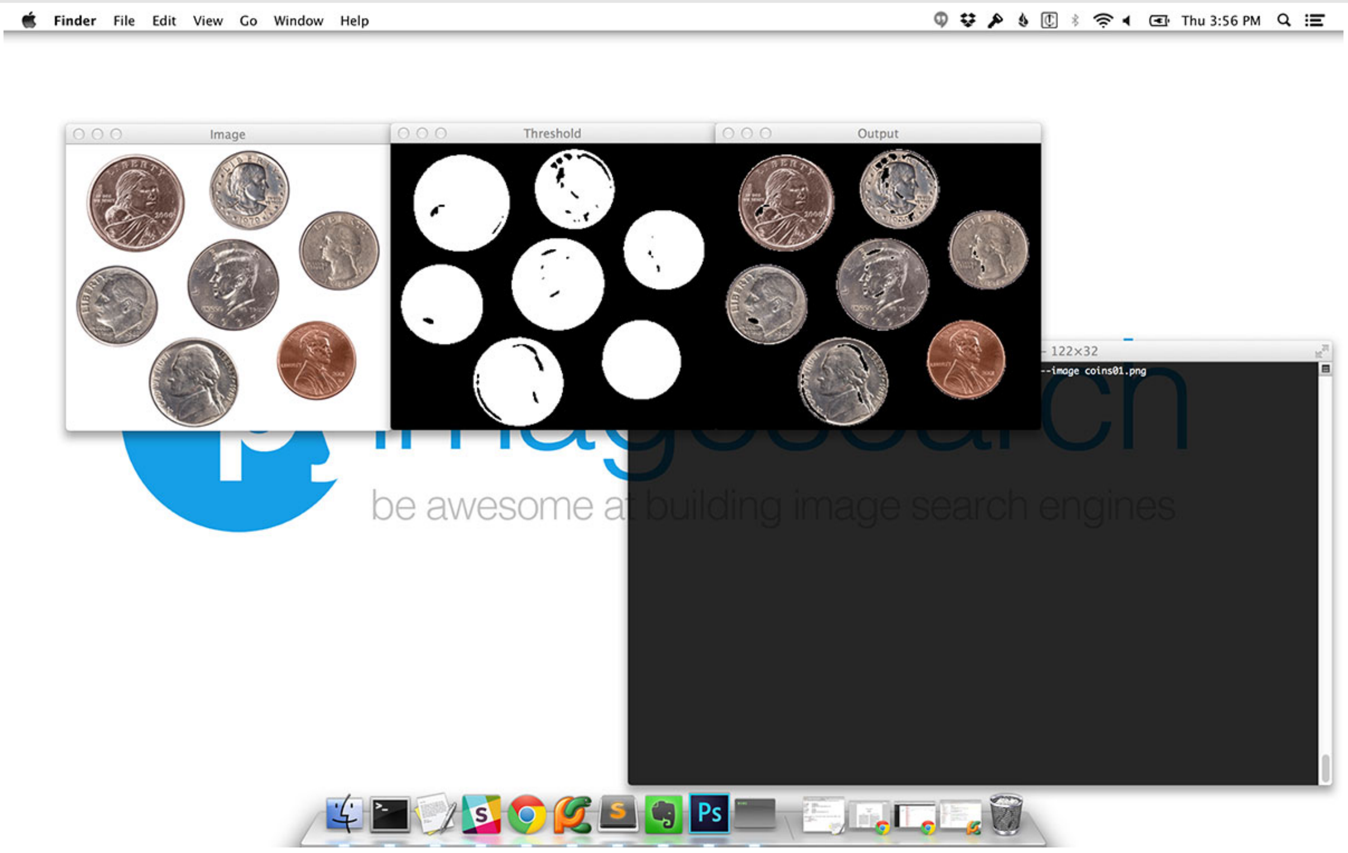
But now that we are using Otsu's method for automatic thresholding, this value of *T* becomes interesting — we do not know what the optimal value of *T* is ahead of time, hence why we are using Otsu's method to compute it for us. **Line 21** prints out the value of *T* as determined by Otsu's method.

Finally, we display the output thresholded image to our screen on **Lines 24 and 25**.

To see Otsu's method running, open up your terminal and execute the following command:

```
otsu_thresholding.py                                                    Shell
$ python otsu_thresholding.py --image coins01.png
```

And you'll see that our coins image has been automatically thresholded for us:



[(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/otsu_threshold_coins01.jpg)](https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/otsu_threshold_coins01.jpg)

FIGURE 8: USING OTSU'S METHOD TO AUTOMATICALLY DETERMINE THE OPTIMAL THRESHOLD.

Pretty nice, right? We didn't even have to supply our value of *T* — Otsu's method automatically took care of this for us. And we still got a nice threshold image as an output. And if we inspect our terminal we'll see that Otsu's method computed a value of *T*:

Feedback

```
otsu_thresholding.py                                                    Shell
Otsu's thresholding value: 191.0
```

So based on our input image, the optimal value of *T* is *191*; therefore, any pixel *p* that is greater than *191* is set to *0*, and any pixel less than *191* is set to *255* (since we supplied the `cv2.THRESH_BINARY_INV` flag detailed in **Figure 2** above).

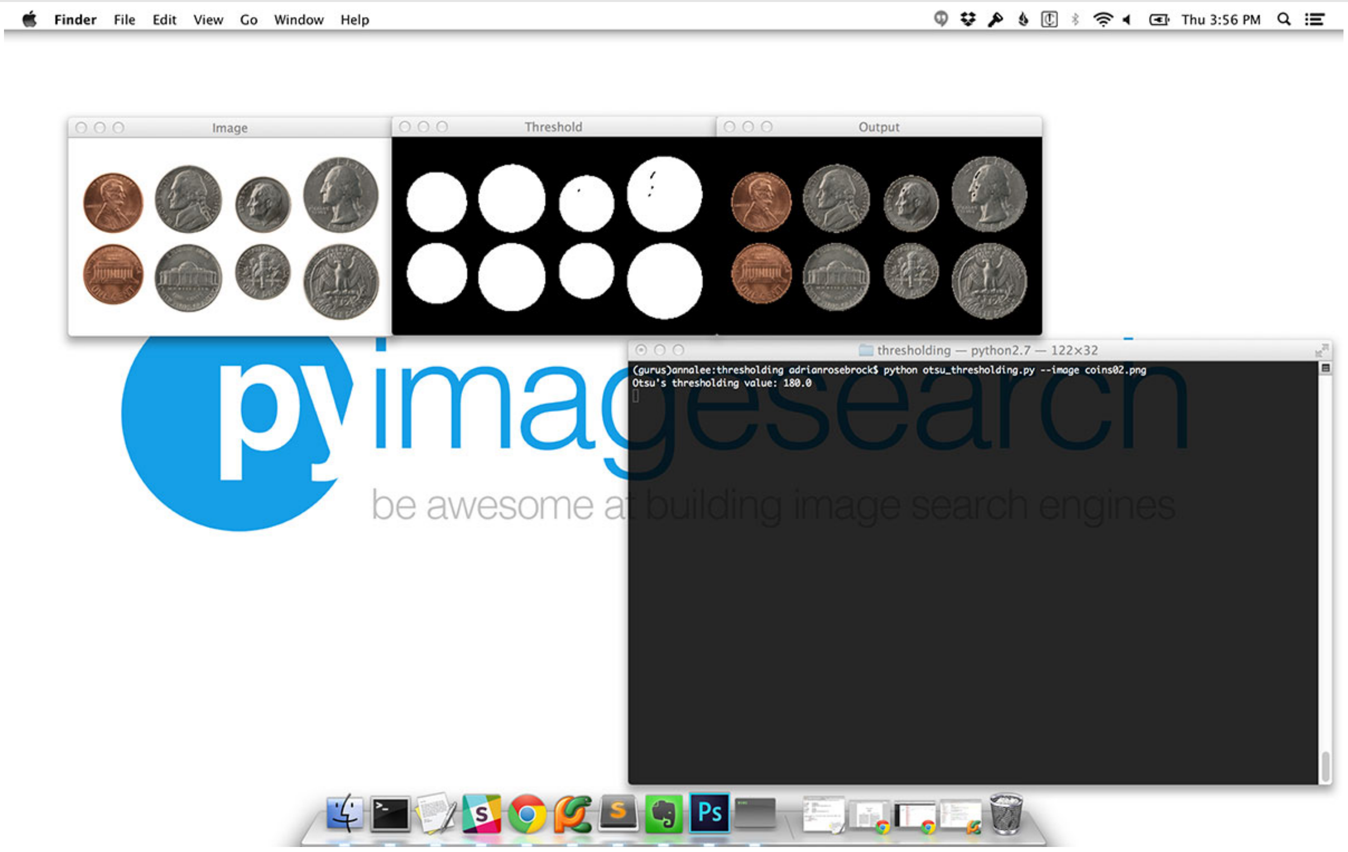
Before we move on to the next example, let's take a second and discuss what is meant by the term "optimal". The value of *T* returned by Otsu's method **may not** be optimal in visual investigation of our image — we can clearly see some gaps and holes in the coins of the thresholded image. But this value *is optimal* in the sense that it does the best possible job to split the foreground and the background ***assuming a bi-modal distribution of grayscale pixel values***.

If the grayscale image does not follow a bi-modal distribution, then Otsu's method will still run, but it may not give us our intended results. In that case, we will have to try *adaptive thresholding*, which we'll cover later in this topic.

Anyway, let's try a second image:

```
otsu_thresholding.py                                                    Shell
$ python otsu_thresholding.py --image coins02.png
```

And here is the output:


Finder
File
Edit
View
Go
Window
Help
Thu 3:56 PM
Image
Threshold
Output
thresholding — python2.7 — 122×32
pyimagesearch
be awesome at building image search engines

```
(gurus)annalee:thresholding adrianrosebrock$ python otsu_thresholding.py --image coins02.png
Otsu's thresholding value: 180.0
```

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/otsu_threshold_coins02.jpg)

FIGURE 9: OTSU'S AUTOMATIC THRESHOLDING METHOD APPLIED TO A SECOND COIN IMAGE.

Feedback

Again, notice that Otsu's method has done a good job separating the foreground from the background for us. And this time, Otsu's method has determined the optimal value of *T* to be *180*. Any pixel value greater than to *180* is set to *0*, and any pixel less than *180* is set to *255* (again, assuming inverse thresholding, as detailed in **Figure 2**).

As you can see, Otsu's method can save us a lot of time guessing and checking the best value of *T*. However, there are some major drawbacks.

The first is that Otsu's method assumes a bi-modal distribution of the grayscale pixel intensities of our input image. If this is not the case, then Otsu's method can return sub-par results.

Secondly, Otsu's method is a ***global thresholding*** method. In situations where lighting conditions are semi-stable and the objects we want to segment have sufficient contrast from the background, we might be able to get away with Otsu's method.

But when the lighting conditions are non-uniform — such as when different parts of the image are illuminated more than others, we can run into some serious problem. And when that's the case, we'll need to rely on *adaptive thresholding*.

Adaptive Thresholding

As we discussed earlier in this section, one of the downsides of using simple thresholding methods is that we need to manually supply our threshold value *T*. Furthermore, finding a good value of *T* may require many manual experiments and parameter tunings, which is simply not practical in most situations.

To aid us in automatically determining the value of *T*, we leveraged Otsu's method. And while Otsu's method can save us a lot of time playing the "guess and checking" game, we are left with only a ***single*** value of *T* to threshold the ***entire*** image.

For simple images with controlled lighting conditions, this usually isn't a problem. But for situations when the lighting is non-uniform across the image, having only a single value of *T* can seriously hurt our thresholding performance.

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

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

In adaptive thresholding, sometimes called local thresholding, our goal is to *statistically examine* the pixel intensity values in the neighborhood of a given pixel *p*.

The general assumption that underlies all adaptive and local thresholding methods is that smaller regions of an image are more likely to have approximately uniform illumination. This implies that local regions of an image will have similar lighting, as opposed to the image as a whole, which may have dramatically different lighting for each region.

However, choosing the size of the pixel neighborhood for local thresholding is ***absolutely crucial***.

The neighborhood must be large enough to cover sufficient background and foreground pixels, otherwise the value of *T* will be more or less irrelevant.

Feedback

But if we make our neighborhood value too large, then we completely violate the assumption that local regions of an image will have approximately uniform illumination. Again, if we supply a very large neighborhood, then our results will look very similar to global thresholding using the simple thresholding or Otsu's methods.

In practice, tuning the neighborhood size is (usually) not that hard of a problem. You'll often find that there is a broad range of neighborhood sizes that provide you with adequate results — it's not like finding an optimal value of *T* that could make or break your thresholding output.

So as I mentioned above, our goal in adaptive thresholding is to ***statistically*** examine local regions of our image and determine an optimal value of *T* for each region — which begs the question: *Which statistic do we use to compute the threshold value* T *for each region?*

It is common practice to use either the arithmetic mean or the Gaussian mean of the pixel intensities in each region (other methods do exist, but the arithmetic mean and the Gaussian mean are by far the most popular).

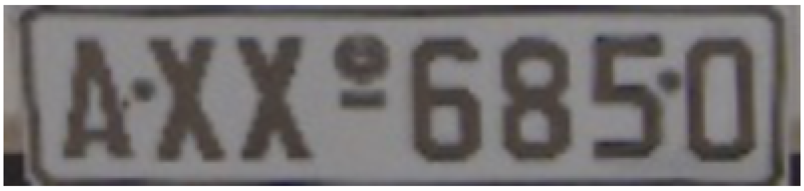
In the arithmetic mean, each pixel in the neighborhood contributes equally to computing *T*. And in the Gaussian mean, pixel values farther away from the *(x, y)*-coordinate center of the region contribute less to the overall calculation of *T*.

The general formula to compute *T* is thus:

$$T = mean(I_L) - C$$

where the $mean$ is either the arithmetic or Gaussian mean, I_L is the local sub-region of the image I, and C is some constant which we can use to fine tune the threshold value *T*.

To demonstrate the utility of adaptive thresholding versus global thresholding, we are going to segment the license plate characters and letters from the license plate itself in the following image:

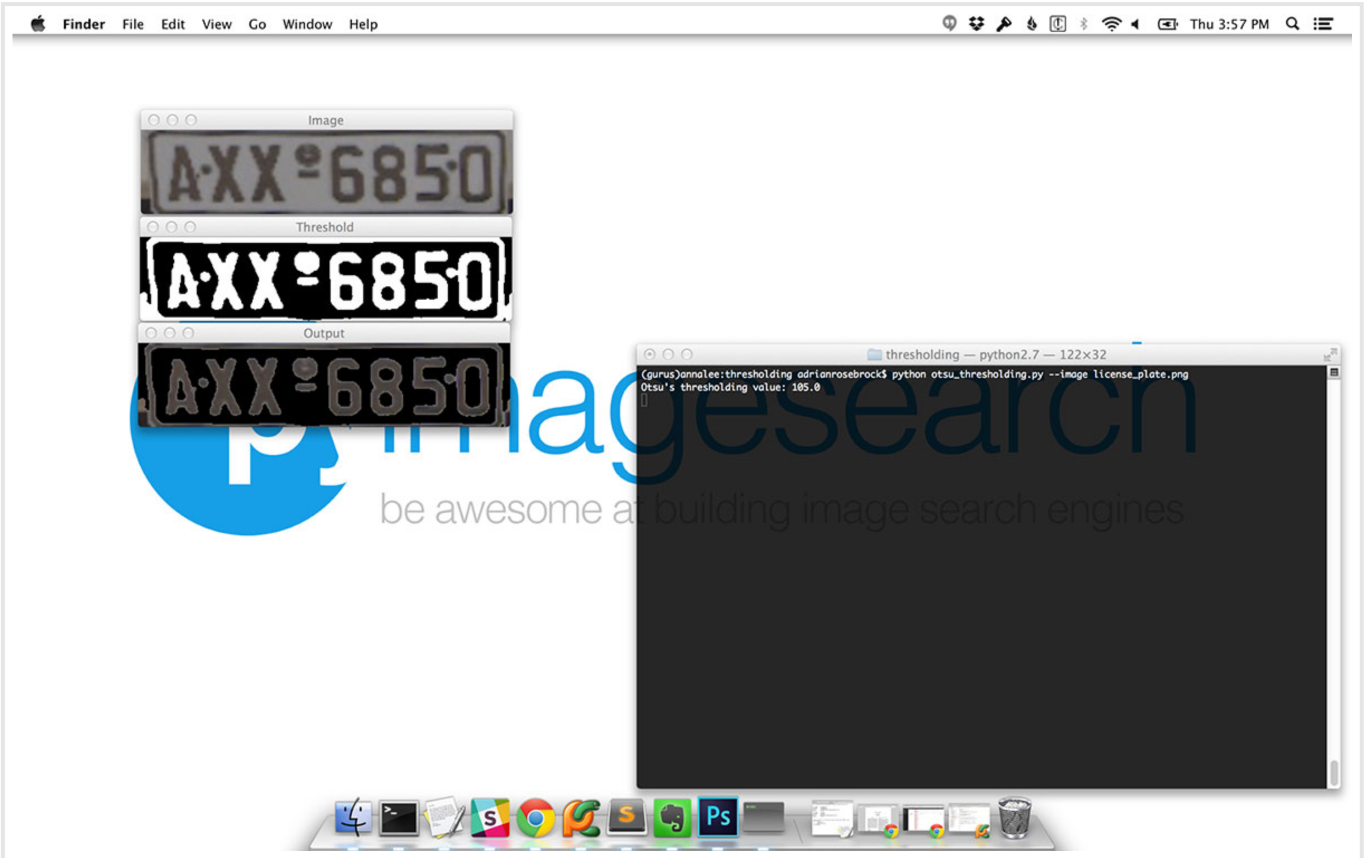


(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/adaptive_threhsold_license_plate.png)

FIGURE 10: OUR GOAL IS TO SEGMENT THE "AXX 6850" CHARACTERS FROM THE BACKGROUND OF THE LICENSE PLATE.

Given this input image, we want to apply thresholding to separate the "AXX 6850" characters from the license plate background. Once we have these thresholded characters, we could then pass them on to algorithms for actually *recognizing* the characters, which we'll be covering in **Module 6 (https://gurus.pyimagesearch.com/lessons/what-is-anpr/)** on automatic license plate detection and recognition.

Let's start by applying Otsu's method and seeing where that gets us:


Finder File Edit View Go Window Help
Thu 3:57 PM
Image
Threshold
Output
thresholding — python2.7 — 122×32
pyimagesearch
be awesome at building image search engines

```
(gurus)annalee:thresholding adrianrosebrock$ python otsu_thresholding.py --image license_plate.png
Otsu's thresholding value: 105.0
```

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

FIGURE 11: APPLYING OTSU'S THRESHOLDING METHOD IN AN ATTEMPT TO SEGMENT THE FOREGROUND CHARACTERS FROM THE BACKGROUND LICENSE PLATE.

On the *top* we have our original input image. And in the *center* we have the output of Otsu's thresholding method.

At first glance, this output doesn't look too bad. Each character appears to be neatly segmented from the background — except for that number *0* at the end.

In this case, it looks like the actual bolt of the license plate that holds the plate to the car is very close to the number *0*. And since we are using global thresholding, our method is not able to detect the very small gap in between the bolt and the number *0*.

While this doesn't seem like a big deal, failing to extract high quality representations of each license plate character can really hurt our performance when we go to *classify* and *recognize* our license plate characters in **Module 6** (https://gurus.pyimagesearch.com/lessons/what-is-anpr/).

So what are we going to do to disconnect the license plate bolt and the number *0*?

Apply adaptive thresholding of course!

Let's take a look at some code to help us solve this problem:

adaptive_thresholding.py Python

```python
1  # import the necessary packages
2  from skimage.filters import threshold_local
3  import argparse
4  import cv2
5
6  # construct the argument parser and parse the arguments
7  ap = argparse.ArgumentParser()
8  ap.add_argument("-i", "--image", required=True, help="Path to the image")
9  args = vars(ap.parse_args())
10
11 # load the image, convert it to grayscale, and blur it slightly
12 image = cv2.imread(args["image"])
13 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
14 blurred = cv2.GaussianBlur(image, (5, 5), 0)
15 cv2.imshow("Image", image)
16
17 # instead of manually specifying the threshold value, we can use adaptive
18 # thresholding to examine neighborhoods of pixels and adaptively threshold
19 # each neighborhood -- in this example, we'll calculate the mean value
20 # of the neighborhood area of 25 pixels and threshold based on that value;
21 # finally, our constant C is subtracted from the mean calculation (in this
22 # case 15)
23 thresh = cv2.adaptiveThreshold(blurred, 255,
24     cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 25, 15)
25 cv2.imshow("OpenCV Mean Thresh", thresh)
26
27 # personally, I prefer the scikit-image adaptive thresholding, it just
28 # feels a lot more "Pythonic"
29 T = threshold_local(blurred, 29, offset=5, method="gaussian")
30 thresh = (blurred < T).astype("uint8") * 255
31 cv2.imshow("scikit-image Mean Thresh", thresh)
32 cv2.waitKey(0)
```

Lines 1-15 simply setup the problem for us. We import our packages, setup the argument parser, load our license plate image, convert it to grayscale, and then blur it slightly to help reduce noise that might throw off our thresholding.

We'll be using OpenCV to perform our adaptive thresholding — but we'll also be using another favorite computer vision + image processing library of mine, scikit-image (http://scikit-image.org/).

I actually prefer the scikit-image implementation of adaptive thresholding over the OpenCV one, since it is less verbose and more Pythonic than the OpenCV one, so I'll demonstrate how to use both in this example.

Lines 23 and 24 handle applying adaptive thresholding using the OpenCV `cv2.adaptiveThreshold` function.

We start by passing in the (blurred) license plate image. The second parameter is the output threshold value, just as in simple thresholding and Otsu's method. The third argument is the adaptive thresholding method. Here we supply a value of `cv2.ADAPTIVE_THRESH_MEAN_C` to indicate that we are using the arithmetic mean of the local pixel neighborhood to compute our threshold value of *T*. We could also supply a value of `cv2.ADAPTIVE_THRESH_GAUSSIAN_C` to indicate we want to use the Gaussian average — which method you choose is entirely dependent on your application and situation, so you'll want to play around with both methods.

The fourth value to `cv2.adaptiveThreshold` is the threshold method, again just like in the **Simple Thresholding** and **Otsu's Method** sections. Here we pass in a value of `cv2.THRESH_BINARY_INV` to indicate that any pixel value that passes the threshold test will have an output value of *0*. Otherwise, it will have a value of *255*.

The fifth parameter is our pixel neighborhood size. Here you can see that we'll be computing the mean grayscale pixel intensity value of each 25×25 sub-region in the image to compute our threshold value *T*.

Feedback

The final argument to `cv2.adaptiveThreshold` is the constant *C* which I mentioned above — this value simply lets us *fine tune* our threshold value. There may be situations where the mean value alone is not discriminating enough between the background and foreground — thus by adding or subtracting some value *C*, we can improve the results of our threshold. Again, the value you use for *C* is entirely dependent on your application and situation, but this value tends to be fairly easy to tune.

Lines 29 and 30 then perform adaptive thresholding using the scikit-image `threshold_local` function, which you can read the full documentation about here (http://scikit-image.org/docs/dev/api/skimage.filters.html#threshold-adaptive). While this method takes an extra line of code, I actually prefer it to the OpenCV method.

To use the `threshold_local` function, we start by passing in the blurred image we want to threshold. We then supply a value of *29* for our 29×29 pixel neighborhood we are going to inspect. The `offset` parameter is equivalent to our *C* parameter in the `cv2.adaptiveThreshold` function — it simply controls how much we are going to add or subtract to the mean of the local region.

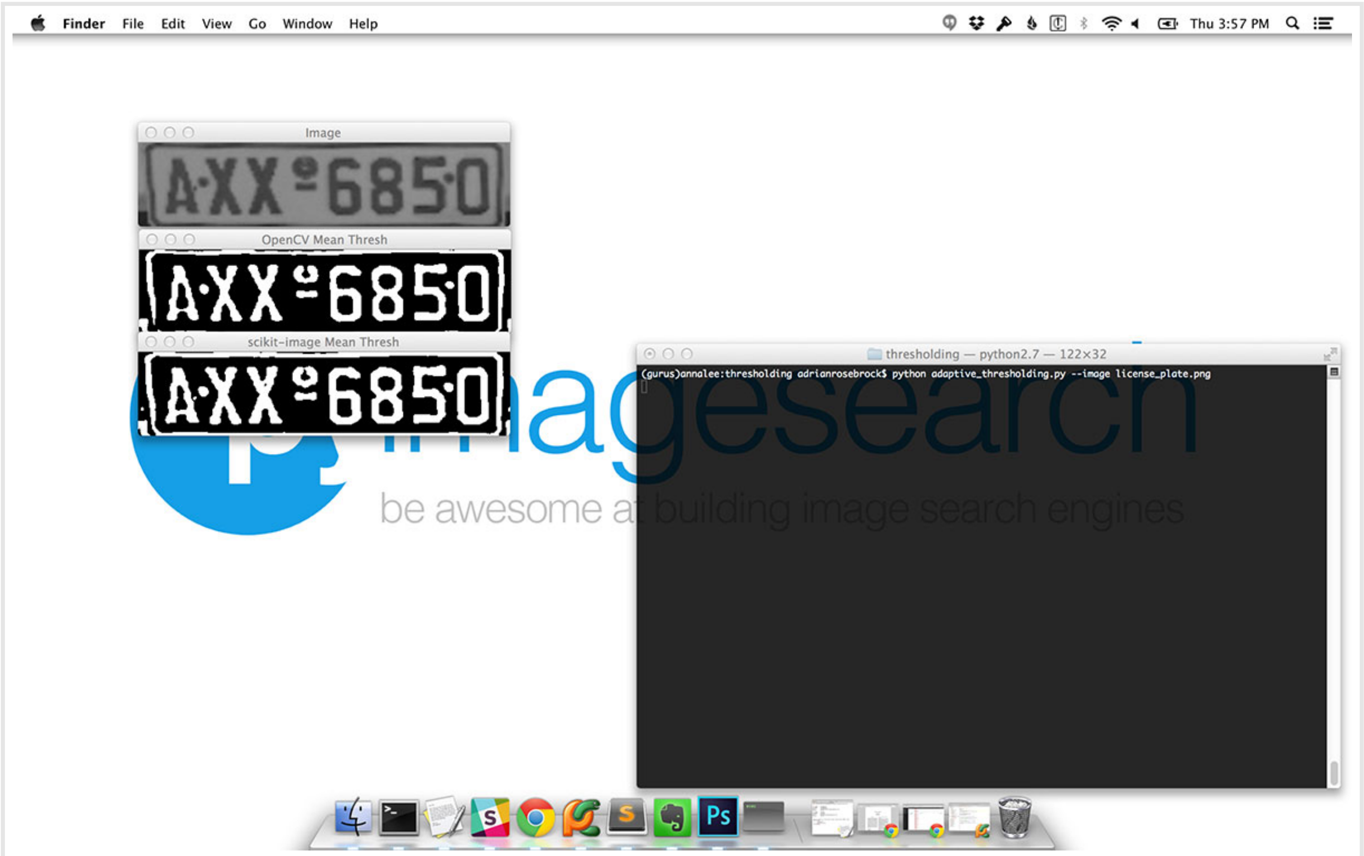
The `threshold_local` function defaults to the Gaussian mean of the local region, but we could also use the arithmetic mean, median, or any other custom statistic by adjusting the optional `method` argument (see the scikit-image documentation (http://scikit-image.org/docs/dev/api/skimage.filters.html#threshold-local) for more details).

The `threshold_local` function actually returns our segmented objects as *black* appearing on a *white* background, so to fix this, we just take the bitwise NOT on **Line 30**. Again, it's an extra line of code when compared to the OpenCV method, but I find the scikit-image method of adaptive thresholding to be much more intuitive and less verbose. Feel free to disagree with me and use whichever method you feel more comfortable with.

To see the results of our adaptive thresholding, just execute the following command:

```
adaptive_thresholding.py                                                    Python
$ python adaptive_thresholding.py --image license_plate.png
```

And then examine the output:



```
(gurus)annalee:thresholding adrianrosebrock$ python adaptive_thresholding.py --image license_plate.png
```

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

FIGURE 12: THE RESULT OF APPLYING ADAPTIVE THRESHOLDING — THE GAP BETWEEN THE BOLT AND THE NUMBER 0 IS NOW CLEARLY VISIBLE.

On the *top* we have our original input image. In the *center* we have the output of the OpenCV `cv2.thresholdAdaptive` function. And on the *bottom* we have the output of the scikit-image `threshold_local` function.

In *both cases* we can see that the connection between the license plate bolt and the number *0* has been cleared! We can clearly see that using adaptive thresholding can achieve better results than normal thresholding, especially in the presence of non-uniform lighting conditions.

Summary

In this lesson we learned all about thresholding: what thresholding is, why we use thresholding, and how to perform thresholding.

We started by performing *simple thresholding* which requires us to manually supply a value of *T* to perform the threshold. However, we quickly realized that manually supplying a value of *T* is very tedious and requires us to hardcode this value, implying that this method will not work in all situations.

We then moved on to *Otsu's thresholding method*, which *automatically* computes the optimal value of *T* for us assuming a bi-modal distribution of the grayscale representation of our input image. The problem here is that (1) our input images need to be bi-modal for Otsu's method to obtain visually appealing results, and (2) Otsu's method is a *global thresholding* method, which implies that we need to have at least some decent control over our lighting conditions.

Feedback

In situations where our lighting conditions are less than ideal, or we simply cannot control them, we discussed *adaptive thresholding* (which is also known as *local thresholding*). Adaptive thresholding makes the assumption that local regions of an image will have more uniform illumination and lighting than the image as a whole. Thus, to obtain better thresholding results we should investigate sub-regions of an image and threshold them individually to obtain our final output image.

Adaptive thresholding tends to produce good results, but is more computationally expensive than Otsu's method or simple thresholding — but in cases where you haven non-uniform illumination conditions, adaptive thresholding is a very useful tool to have.

Downloads:

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

Quizzes		Status
1	Thresholding Quiz (https://gurus.pyimagesearch.com/quizzes/thresholding-quiz/)	

← Previous Lesson (https://gurus.pyimagesearch.com/lessons/lighting-and-color-spaces/) Next Lesson → (https://gurus.pyimagesearch.com/lessons/gradients-and-edge-detection/)

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

Feedback

- 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