1.6: Morphological operations

Morphological operations are simple transformations applied to binary or grayscale images. More specifically, we apply morphological operations to *shapes* and *structures* inside of images. We can use morphological operations to *increase* the size of objects in images as well as *decrease* them. We can also utilize morphological operations to *close* gaps between objects as well as *open* them.

Morphological operations "probe" an image with a *structuring element*. This structuring element defines the neighborhood to be examined around each pixel. And based on the given operation and the size of the structuring element we are able to adjust our output image.

This may sound vague. That's because it is. There are many different morphological transformations that perform "opposite" operations from one another — just as addition is the "opposite" of subtraction, we can think of the erosion morphological operation as the "opposite" of dilation.

If this sounds confusing, don't worry — we'll be reviewing many examples of each of these morphological transformations, and by the time you are done reading through this lesson, you'll have a crystal clear view of morphological operations.

Objective:

After reading this lesson you should be able to apply the following morphological operations to images:

- Erosion
- Dilation

- Opening
- Closing
- Morphological gradient
- Black hat
- Top hat (or "White hat")

Morphological Operations

Morphological operations are one of my favorite topics to cover in image processing.

Why is that?

Because these transformations are so *powerful*.

Often times I see computer vision researchers and developers trying to solve a problem and immediately dive into advanced computer vision and machine learning techniques. It seems that once you learn to wield a hammer, every problem looks like a nail.

However, there are times where a more "elegant" solution can be found using less advanced techniques. Sure, these techniques may not be floating around on a cloud of buzzwords, but they can get the job done.

For instance, I once wrote an article on the PyImageSearch blog regarding detecting barcodes in images (http://www.pyimagesearch.com/2014/11/24/detecting-barcodes-images-python-opencv/). I didn't use any fancy techniques. I didn't use any machine learning.

In fact, I was able to detect barcodes in images using ***nothing more than the topics covered in Module 1.***

Crazy, isn't it?

But seriously, pay attention to these transformations — there will be times in your computer vision career when you'll be ready to swing your hammer down on a problem, only to realize that a more elegant, simple solution may already exist. And more than likely, you may find that elegant solution in morphological operations.

Let's go ahead and get started by discussing the component that makes morphological operations possible: the *structuring element*.

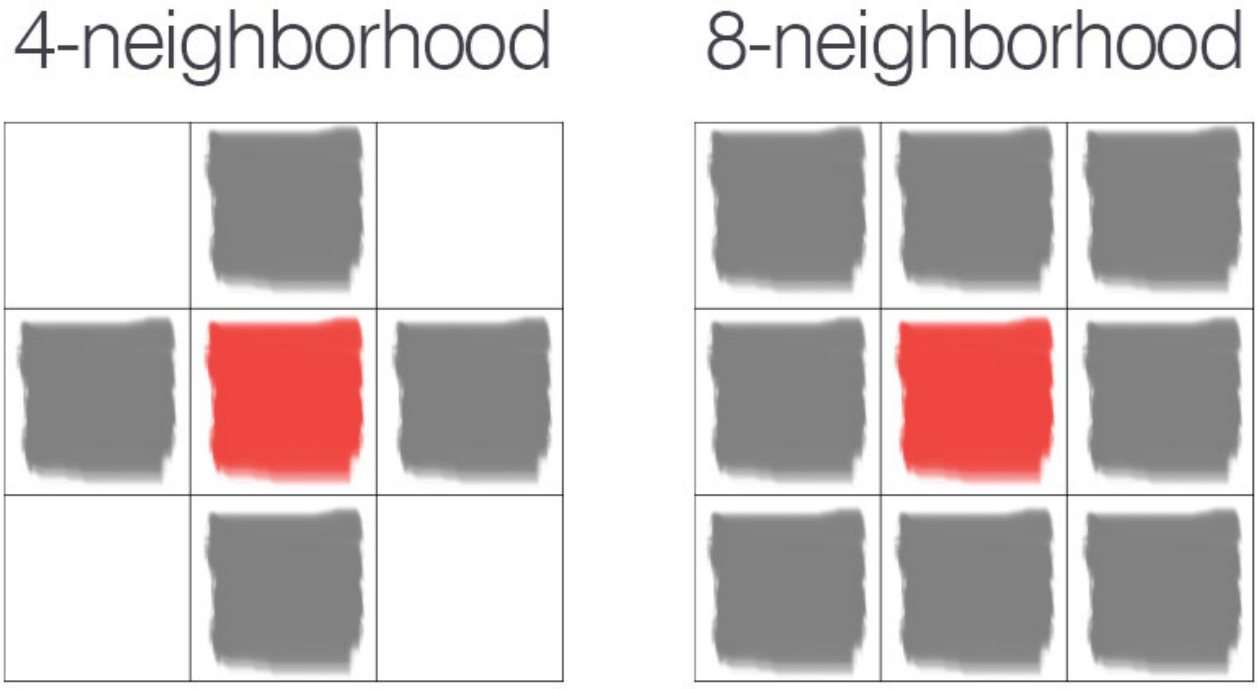
Structuring Element

So remember in **Module 1.5 (https://gurus.pyimagesearch.com/lessons/kernels/)** where we discussed image kernels?

Well, you can (conceptually) think of a *structuring element* as a type of *kernel* or *mask*. However, instead of applying a convolution, we are only going to perform simple tests on the pixels.

And just like in image kernels, the structuring element slides from left-to-right and top-to-bottom for each pixel in the image. Also just like kernels, structuring elements can be of arbitrary neighborhood sizes.

For example, let's take a look at the 4-neighborhood and 8-neighborhood of the central pixel red below:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/4_versus_8_neighborhood.jpg)

FIGURE 1: *(LEFT)* A 4 PIXEL NEIGHBORHOOD SURROUNDING THE CENTER RED PIXEL. *(RIGHT)* AN 8 PIXEL NEIGHBORHOOD SURROUNDING THE CENTER PIXEL.

Here we can see that the central pixel (i.e. the red pixel) is located at the center of the neighborhood. The 4-neighborhood *(left)* then defines the region surrounding the central pixel as the pixels to the north, south, east, and west. The 8-neighborhood *(right)* then extends this region to include the corner pixels as well.

Feedback

This is just an example of two simple structuring elements. But we could also make them arbitrary rectangle or circular structures as well — it all depends on your particular application.

In OpenCV, we can either use the `cv2.getStructuringElement` function or NumPy itself to define our structuring element. Personally, I prefer to use the `cv2.getStructuringElement` function since it gives you more control over the returned element, but again, that is a personal choice.

If the concept of structuring elements is not entirely clear, that's okay. We'll be reviewing many examples of them inside this lesson. For the time being, understand that a structuring element behaves similar to a kernel or a mask — but instead of *convolving* the input image with our structuring element, we're instead only going to be applying simple pixel tests.

Now that we have a basic understanding of structuring elements, let's get started with our first morphological operation: *erosion*.

Erosion

Just like water rushing along a river bank *erodes* the soil, an erosion in an image "erodes" the foreground object and makes it smaller. Simply put, pixels near the boundary of an object in an image will be discarded, "eroding" it away.

Erosion works by defining a structuring element and then sliding this structuring element from left-to-right and top-to-bottom across the input image.

A foreground pixel in the input image will be kept ***only if ALL pixels*** inside the structuring element are > *0*. Otherwise, the pixels are set to *0* (i.e. background).

Erosion is useful for removing small blobs in an image or disconnecting two connected objects.

We can perform erosion by using the `cv2.erode` function. Let's open up a new file, name it `morphological.py`, and start coding:

```
morphological.py                                                    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 and convert it to grayscale
11 image = cv2.imread(args["image"])
12 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
13 cv2.imshow("Original", image)
14
15 # apply a series of erosions
16 for i in range(0, 3):
17     eroded = cv2.erode(gray.copy(), None, iterations=i + 1)
18     cv2.imshow("Eroded {} times".format(i + 1), eroded)
19     cv2.waitKey(0)
```

We start off by importing our necessary packages, parsing our command line arguments, and then loading our image from disk on **Lines 1-11**.

In most examples in this lesson we'll be applying morphological operations to the PyImageSearch logo, which we can see below:

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

FIGURE 2: THE EXAMPLE PYIMAGESEARCH LOGO THAT WE'LL BE APPLYING MORPHOLOGICAL OPERATIONS TO IN THIS LESSON.

Feedback

As I mentioned earlier in this lesson, we normally apply morphological operations to *binary* images. As we'll see later in this lesson, there are exceptions to that, especially when using the *black hat* and *white hat* operators, but for the time being, we are going to assume we are working with a binary image, where the background pixels are *black* and the foreground pixels are *white*.

Given our logo image, we apply a series of erosions on **Lines 15-19**. The `for` loop controls the number of times, or *iterations*, we are going to apply the erosion. As the number of erosions increases, the foreground logo will start to "erode" and disappear.

We perform the actual erosion on **Line 17** by making a call to the `cv2.erode` function. This function

takes two required arguments and a third optional one. The first argument is the image that we want to erode — in this case, it's our binary image (i.e. the PyImageSearch logo). The second argument to `cv2.erode` is the structuring element. If this value is `None`, then a 3×3 structuring element, identical to the 8-neighborhood structuring element we saw above will be used. Of course, you could supply your own custom structuring element here instead of `None` as well. The last argument is the number of `iterations` the erosion is going to be performed. As the number of iterations increases, we'll see more and more of the PyImageSearch logo eaten away.

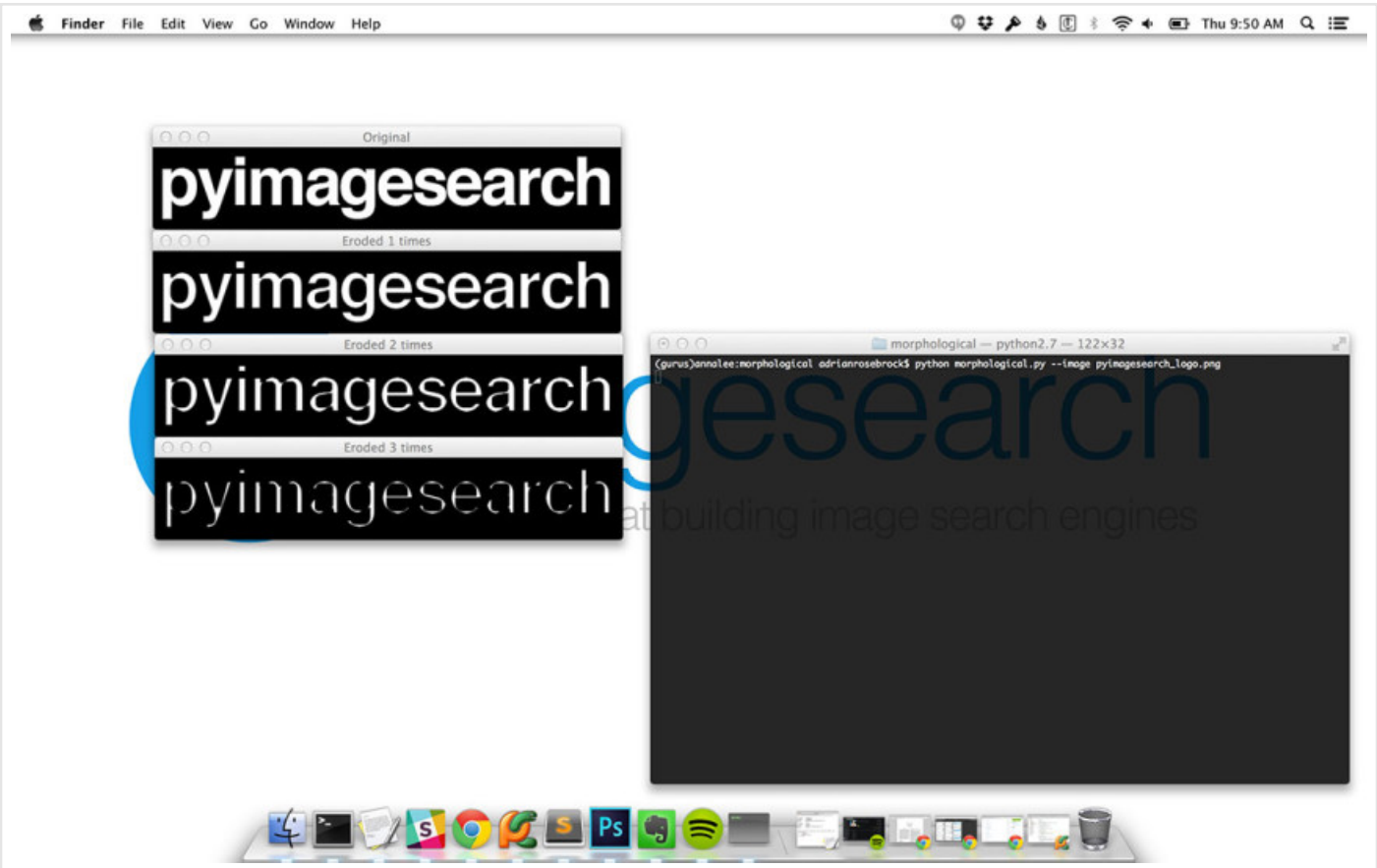
Finally, **Line 18 and 19** show us our eroded image.

To see erosion in action, open up a terminal, navigate to your source code, and execute the following command:

morphological.py Python

```
1 $ python morphological.py --image pyimagesearch_logo.png
```

You should then see the following output image:



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

FIGURE 3: APPLYING EROSION TO OUR INPUT IMAGE. AS THE NUMBER OF ITERATIONS INCREASES, THE MORE AND MORE OF THE LOGO IS ERODED AWAY.

Feedback

On the very top we have our original image. And then underneath the image, we have the logo being eroded a total of 1, 2, and 3 times, respectively.

Notice as the number of erosion iterations increases, more and more of the logo is eaten away.

Again, erosions are most useful for removing small blobs from an image or disconnected two connected components. With this in mind, take a look at the letter *p* in the PyImageSearch logo. Notice how the circular region of the *p* has disconnected from the stem after 2 erosions — this is an example of disconnecting two connected components of an image.

Dilation

The opposite of an *erosion* is a *dilation*. Just like an *erosion* will *eat away* at the foreground pixels, a *dilation* will *grow* the foreground pixels.

Dilations *increase* the size of foreground object and are especially useful for joining broken parts of an image together.

Dilations, just as an erosion, also utilize structuring elements — a center pixel *p* of the structuring element is set to *white* if ***ANY*** pixel in the structuring element is > *0*.

We apply dilations using the `cv2.dilate` function:

morphological.py Python

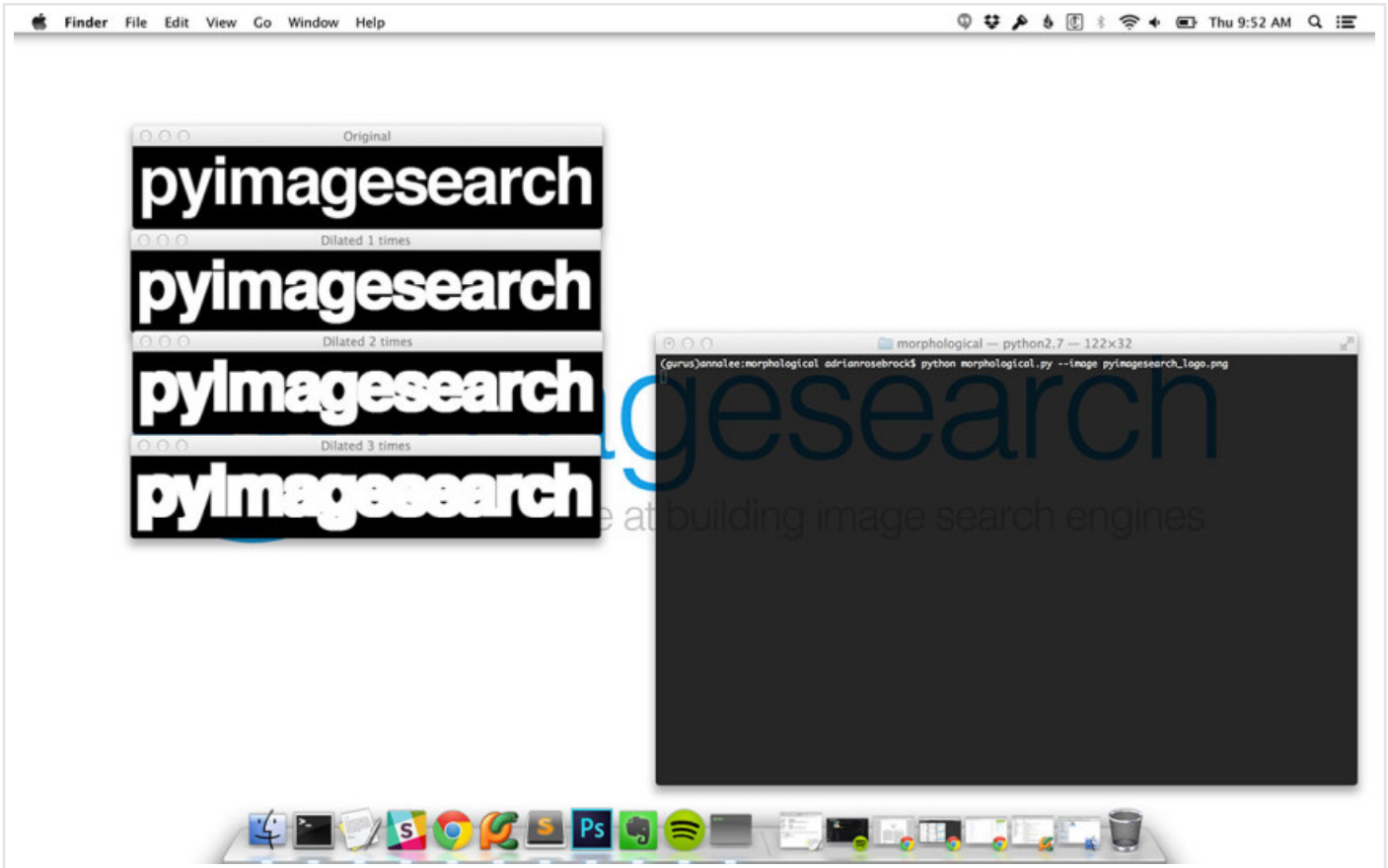
```
21 # close all windows to cleanup the screen
22 cv2.destroyAllWindows()
23 cv2.imshow("Original", image)
24
25 # apply a series of dilations
26 for i in range(0, 3):
27     dilated = cv2.dilate(gray.copy(), None, iterations=i + 1)
28     cv2.imshow("Dilated {} times".format(i + 1), dilated)
29     cv2.waitKey(0)
```

Lines 22 and 23 simply close all open windows and display our original image to give us a fresh start.

Line 26 then starts looping over the number of iterations, just as we did with the `cv2.erode` function.

The actual dilation is performed on **Line 27** by making a call to the `cv2.dilate` function, where the actual function signature is identical to that of `cv2.erode`. The first argument is the image we want to dilate; the second is our structuring element, which when set to `None` is a 3×3 8-neighborhood structuring element; and the final argument is the number of dilation `iterations` we are going to apply.

The output of our dilation can be seen below.



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

FIGURE 4: APPLYING A DILATION TO OUR INPUT IMAGE. NOTICE HOW THE FOREGROUND REGION HAS *GROWN*.

Feedback

Again, at the very *top* we have our original input image. And below the input image we have our image dilated 1, 2, and 3 times, respectively.

Unlike an erosion where the foreground region is slowly eaten away at, a dilation actually *grows* our foreground region.

Dilations are especially useful when joining broken parts of an object — for example, take a look at the *bottom* image where we have applied a dilation with 3 iterations. By this point, the gaps between ***all*** letters in the logo have been joined.

Opening

An opening is an ***erosion followed by a dilation***.

Performing an opening operation allows us to remove small blobs from an image: first an erosion is applied to remove the small blobs, then a dilation is applied to regrow the size of the original object.

Let's look at some example code to apply an *opening* to an image:

```
morphological.py                                                    Python
31 # close all windows to cleanup the screen and initialize the list
32 # of kernels sizes that will be applied to the image
33 cv2.destroyAllWindows()
34 cv2.imshow("Original", image)
35 kernelSizes = [(3, 3), (5, 5), (7, 7)]
36
37 # loop over the kernels and apply an "opening" operation to the image
38 for kernelSize in kernelSizes:
39     kernel = cv2.getStructuringElement(cv2.MORPH_RECT, kernelSize)
40     opening = cv2.morphologyEx(gray, cv2.MORPH_OPEN, kernel)
41     cv2.imshow("Opening: ({}, {})".format(kernelSize[0], kernelSize[1]), opening)
42     cv2.waitKey(0)
```

Lines 33 and 34 perform cleanup by closing all open windows and re-displaying our original image. But take a look at the new variable we are defining on **Line 35**, `kernelSizes`. This variable defines the width and height respectively of the structuring element we are going to apply.

We loop over each of these `kernelSizes` on **Line 38** and then make a call to `cv2.getStructuringElement` on **Line 39** to build our structuring element. The `cv2.getStructuringElement` function requires two arguments: the first is the *type* of structuring element we want, and the second is the *size* of the structuring element (which we grab from the `for` loop on **Line 38**).

We pass in a value of `cv2.MORPH_RECT` to indicate that we want a *rectangular* structuring element. But you could also pass in a value of `cv2.MORPH_CROSS` to get a *cross shape* structuring element (a cross is like a 4-neighborhood structuring element, but can be of any size), or `cv2.MORPH_ELLIPSE` to get a circular structuring element. Exactly which structuring element you use is dependent upon your application — and I'll leave it as an exercise to the reader to play with each of these structuring elements.

The actual *opening* operation is performed on **Line 40** by making a call to the `cv2.morphologyEx` function. This function is abstract in a sense — it allows us to pass in whichever morphological operation we want, followed by our kernel/structuring element.

The first required argument of `cv2.morphologyEx` is the image we want to apply the morphological operation to. The second argument is the actual *type* of morphological operation — in this case, it's an *opening* operation. The last required argument is the kernel/structuring element that we are using.

Finally, **Lines 41 and 42** display the output of applying our opening.

As mentioned above, an opening operation allows us to remove small blobs from an image. I went ahead and added some blobs to the PyImageSearch logo:

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

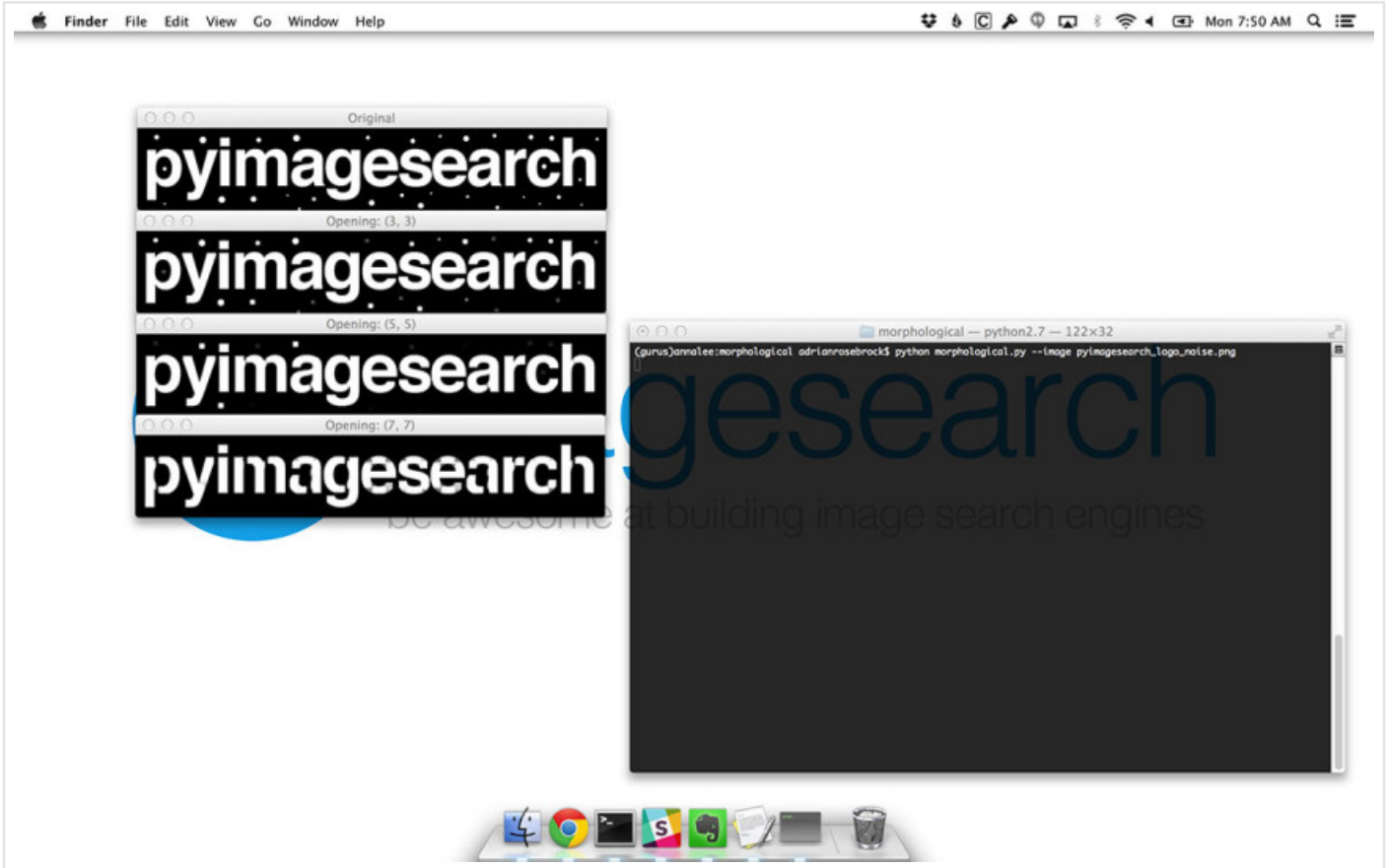
FIGURE 5: THE PYIMAGESEARCH LOGO WITH RANDOM BLOBS ADDED TO IT.

And let's go ahead and execute our script using the new image:

morphological.py Python

```
1 $ python morphological.py --image pyimagesearch_logo_noise.png
```

Here we can see the output of applying the opening morphological operation:



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

FIGURE 6: THE RESULTS OF APPLYING AN OPENING OPERATION TO OUR IMAGE — NOTICE HOW THE SMALL, RANDOM BLOBS HAVE BEEN REMOVED.

Notice how by the time we are using a kernel of size 5×5, the small, random blobs are nearly completely

gone. Finally, the third iteration has a kernel of size 7×7, our opening operation has not only removed all the random blobs, but also "opened" holes in the letter *p* and the letter *a*.

Closing

The exact opposite to an *opening* would be a *closing*. A closing is a ***dilation followed by an erosion***.

As the name suggests, a *closing* is used to close holes inside of objects or for connecting components together.

morphological.py Python

```
44 # close all windows to cleanup the screen
45 cv2.destroyAllWindows()
46 cv2.imshow("Original", image)
47
48 # loop over the kernels and apply a "closing" operation to the image
49 for kernelSize in kernelSizes:
50     kernel = cv2.getStructuringElement(cv2.MORPH_RECT, kernelSize)
51     closing = cv2.morphologyEx(gray, cv2.MORPH_CLOSE, kernel)
52     cv2.imshow("Closing: ({}, {})".format(kernelSize[0], kernelSize[1]), closing)
53     cv2.waitKey(0)
```

Performing the *closing* operation is again accomplished by making a call to `cv2.morphologyEx`, but this time we are going to indicate that our morphological operation is a closing by specifying the `cv2.MORPH_CLOSE` flag.

Feedback

We'll go back to using our original image (without the random blobs). The output for applying a closing operation with increasing structuring element sizes can be seen below:

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

FIGURE 7: APPLYING A MORPHOLOGICAL CLOSING OPERATION TO OUR INPUT IMAGE.

Notice how the closing operation is starting to bridge the gap between letters in the logo. Furthermore, letters such as *e*, *s*, and *a* are practically filled in.

Morphological Gradient

A morphological gradient is the ***difference between the dilation and erosion***. It is useful for determining the outline of a particular object of an image:

Python

```
55 # close all windows to cleanup the screen
56 cv2.destroyAllWindows()
57 cv2.imshow("Original", image)
58
59 # loop over the kernels and apply a "morphological gradient" operation
60 # to the image
61 for kernelSize in kernelSizes:
62     kernel = cv2.getStructuringElement(cv2.MORPH_RECT, kernelSize)
63     gradient = cv2.morphologyEx(gray, cv2.MORPH_GRADIENT, kernel)
64     cv2.imshow("Gradient: ({}, {})".format(kernelSize[0], kernelSize[1]), gradient)
65     cv2.waitKey(0)
```

The most important line to pay attention to is **Line 63,** where we make a call to `cv2.morphologyEx` — but this time we supply the `cv2.MORPH_GRADIENT` flag to indicate that we want to apply the morphological

gradient operation to reveal the outline of our logo:

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

FIGURE 8: A MORPHOLOGICAL GRADIENT CAN BE USED TO FIND THE OUTLINE OF AN OBJECT IN AN IMAGE.

Feedback

Notice how the outline of the PyImageSearch logo has been clearly revealed after applying the morphological gradient operation.

Top Hat/White Hat

A *top hat* (also known as a *white hat*) morphological operation is the ***difference between the original (grayscale/single channel) input image*** and the ***opening***.

A top hat operation is used to reveal **bright regions** of an image on **dark backgrounds**.

Up until this point we have only applied morphological operations to *binary* images. But we can also apply morphological operations to *grayscale* images as well. In fact, both the *top hat/white hat* and the *black hat* operators are more suited for grayscale images rather than binary ones.

To demonstrate applying morphological operations, let's take a look at the following image where our goal is to detect the license plate region of the car:

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

FIGURE 9: OUR GOAL IS TO APPLY MORPHOLOGICAL OPERATIONS TO FIND THE LICENSE PLATE REGION OF THE CAR.

So how are we going to go about doing this?

Well, taking a look at the example image above, we see that the license plate is **bright** since it's a white region against a **dark background** of the car itself. An excellent starting point to finding the region of a license plate would be to use the *top hat* operator.

To test out the top hat operator, create a new file, name it `hats.py` , and insert the following code:

hats.py 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 and convert it to grayscale
11 image = cv2.imread(args["image"])
12 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
13
14 # construct a rectangular kernel and apply a blackhat operation which
15 # enables us to find dark regions on a light background
16 rectKernel = cv2.getStructuringElement(cv2.MORPH_RECT, (13, 5))
17 blackhat = cv2.morphologyEx(gray, cv2.MORPH_BLACKHAT, rectKernel)
18
19 # similarly, a tophat (also called a "whitehat") operation will enable
20 # us to find light regions on a dark background
21 tophat = cv2.morphologyEx(gray, cv2.MORPH_TOPHAT, rectKernel)
22
23 # show the output images
24 cv2.imshow("Original", image)
25 cv2.imshow("Blackhat", blackhat)
26 cv2.imshow("Tophat", tophat)
27 cv2.waitKey(0)
```

Lines 1-12 are not very exciting. We are simply importing packages, parsing command line arguments, loading our image off disk, and converting our input image to grayscale.

Feedback

Line 16 then defines a *rectangular* structuring element with a width of *13* pixels and a height of *5* pixels. As I mentioned earlier in this lesson, structuring elements can be of arbitrary size. And in this case, we are applying a rectangular element that is almost 3x wider than it is tall.

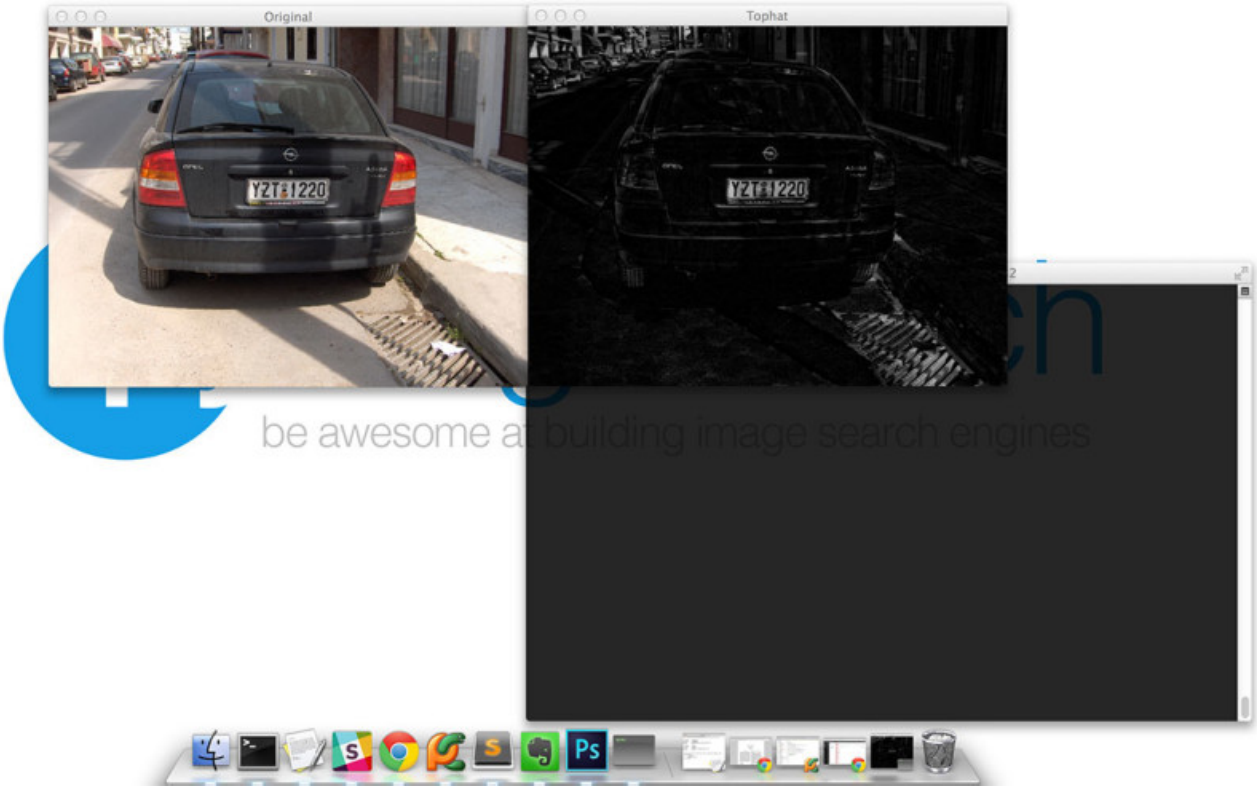
And why is this?

Because a license plate is roughly 3x wider than it is tall!

By having some basic *a priori* knowledge of the objects you want to detect in images, we can construct structuring elements to better aide us in finding them.

Line 17 applies the *black hat* operator (which we'll see the output of in the following section) and **Line 21** applies the *top hat* operator by supplying the `cv2.MORPH_TOPHAT` flag.

Here we can see the output of applying the top hat:



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

FIGURE 10: APPLYING A TOP HAT OPERATION REVEALS LIGHT REGIONS ON A DARK BACKGROUND.

Feedback

Notice how only regions that are *light* against a *dark background* are clearly displayed — in this case, we can clearly see that the license plate region of the car has been revealed.

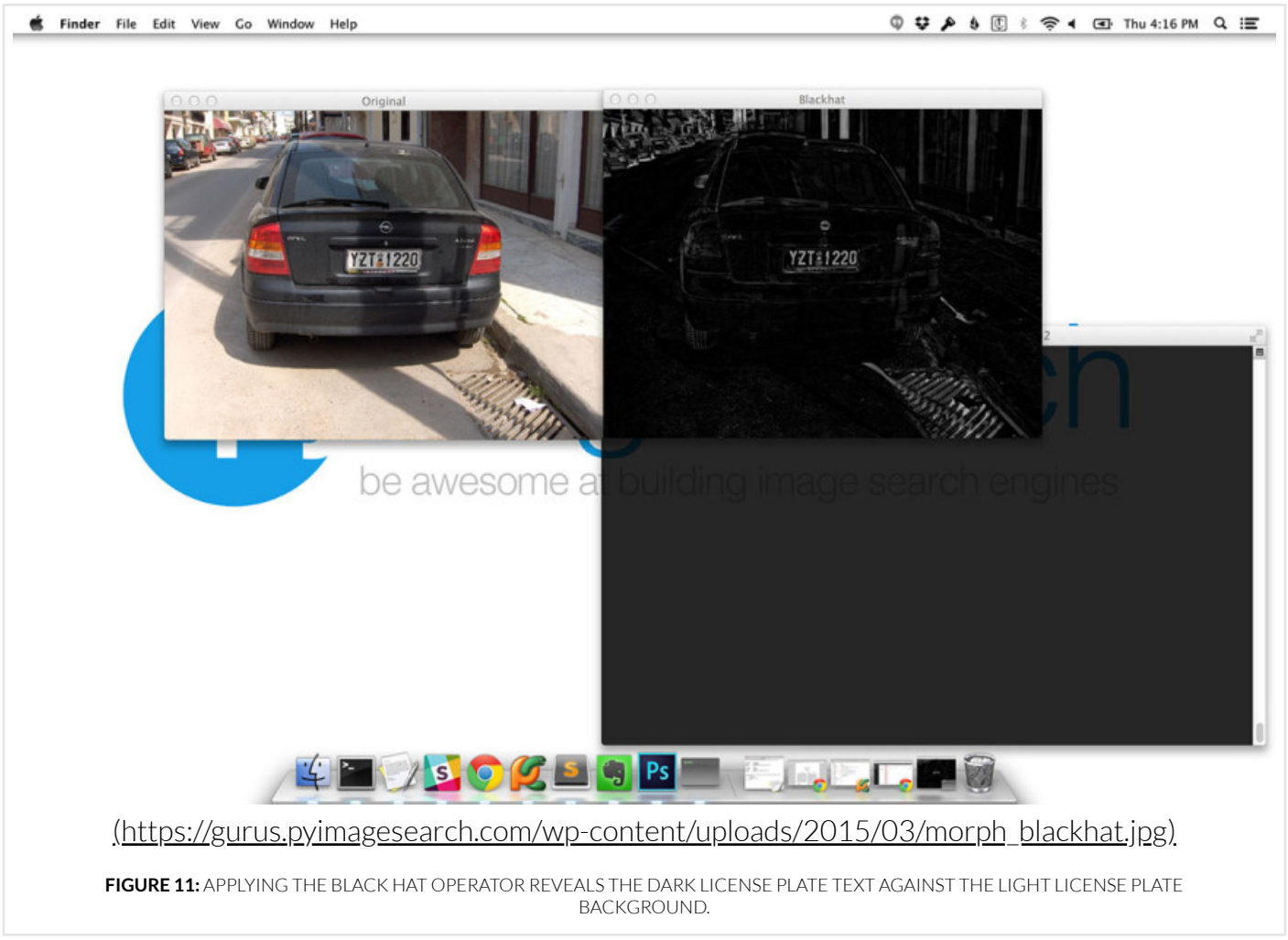
But also note that the license plate characters themselves have not been included. This is because the license plate characters are *dark* against a *light background*. And to reveal our license plate characters we'll need the *black hat* operator.

Black Hat

Of course, the color of a license plate is a lot more static and reliable than the color of a car. In the previous example, we lucked out since our car was a dark color, allowing us to illuminate the license plate region. But if the car had been a *lighter color*, this task would have not been as easy.

One thing that we can (almost always) rely on is that the license plate text itself being darker than the license plate background. To this end, we can apply a *black hat* operator (**Line 17**) by specifying the `cv2.MORPH_BLACKHAT` flag. The black hat operation is the ***difference between the closing of the input image*** and the ***input image itself***. In fact, the black hat operator is simply the *opposite* of the white hat operator!

We apply the black hat operator to reveal dark regions (i.e. the license plate text) against light backgrounds (i.e. the license plate itself).



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

FIGURE 11: APPLYING THE BLACK HAT OPERATOR REVEALS THE DARK LICENSE PLATE TEXT AGAINST THE LIGHT LICENSE PLATE BACKGROUND.

As you can see above, the license plate text is clearly visible after applying our black hat operation!

In fact, we'll be using the *black hat* operator along with *closing* morphological operation to close the gaps in between the license plate characters when building our own automatic license plate detection and recognition system in **Module 6**. (https://gurus.pyimagesearch.com/lessons/what-is-anpr/) As I said, morphological operations are quite the powerful technique when applied thoughtfully!

Summary

In this lesson we learned that morphological operations are image processing transformations applied to either grayscale or binary images. These operations require a *structuring element*, which is used to define the neighborhood of pixels the operation is applied to.

We also reviewed the most important morphological operations that you'll use inside your own

applications: erosion, dilation, opening, closing, ... *morphological gradient, top-hat/white-hat, and black-hat.*

Morphological operations are commonly used as pre-processing steps to more powerful computer vision solutions such as OCR, Automatic Number Plate Recognition (ANPR), and barcode detection.

While these techniques are simple, they are actually extremely powerful and tend to be highly useful when pre-processing your data. *Do not overlook them*.

Downloads:

Download the Code (https://gurus.pyimagesearch.com/protected/code/computer_vision_basics/morphological.zip)

Quizzes	Status
1	Morphological Operations Quiz (https://gurus.pyimagesearch.com/quizzes/morphological-operations-quiz/)

Feedback

← Previous Lesson (https://gurus.pyimagesearch.com/lessons/kernels/) Next Lesson → (https://gurus.pyimagesearch.com/lessons/smoothing-and-blurring/)

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