1.7: Smoothing and blurring

I'm pretty sure we all know what blurring is. It's what happens when your camera takes a picture out of focus. Sharper regions in the image lose their detail. The goal here is to use a low-pass filter to reduce the amount of noise and detail in an image.

Practically, this means that each pixel in the image is mixed in with its surrounding pixel intensities. This "mixture" of pixels in a neighborhood becomes our blurred pixel.

While this effect is usually unwanted in our photographs, it's actually quite helpful when performing image processing tasks. In fact, smoothing and blurring is one of the most *common* pre-processing steps in computer vision and image processing.

For example, we can see that blurring is applied when building a mobile document scanner (http://www.pyimagesearch.com/2014/09/01/build-kick-ass-mobile-document-scanner-just-5-minutes/) on the PyImageSearch blog. We also apply smoothing to aid us in finding our marker when measuring the distance from an object to our camera (https://gurus.pyimagesearch.com/lessons/measuring-distance-from-camera-to-object-in-image/). In both these examples the smaller details in the image are smoothed out and we are left with more of the *structural aspects* of the image.

As we'll see through this course, many image processing and computer vision functions, such as thresholding and edge detection, perform better if the image is first smoothed or blurred.

Objectives:

By the end of this lesson, you should be able to:

1. Understand the role kernels play in smoothing and blurring.
2. Apply simple average blurring.
3. Apply weighted Gaussian blurring.
4. Understand the importance of the median filter.
5. Utilize bilateral filtering to blur an image while preserving edges.

Smoothing and Blurring

Smoothing and blurring is one of the most important pre-processing steps in all of computer vision and image processing. By smoothing an image prior to applying techniques such as edge detection (https://gurus.pyimagesearch.com/lessons/gradients-and-edge-detection/) or thresholding (https://gurus.pyimagesearch.com/lessons/thresholding/) we are able to reduce the amount of high frequency content, such as noise and edges (i.e. the "detail" of an image).

While this may sound counter-intuitive, by reducing the detail in an image we can more easily find objects that we are interested in.

Furthermore, this allows us to focus on the larger structural objects in the image.

In the rest of this lesson we'll be discussing the four main smoothing and blurring options that we'll use through this course: *averaging*, *Gaussian blurring*, *median filtering*, and *bilateral filtering*.

Let's get started with averaging.

Averaging

The first blurring method we are going to explore is averaging.

An average filter does exactly what you think it might do — takes an area of pixels surrounding a central pixel, averages all these pixels together, and replaces the central pixel with the average.

By taking the average of the region surrounding a pixel, we are smoothing it and replacing it with the value of its local neighborhood. This allows us to reduce noise and the level of detail, simply by relying on the average.

Remember when we discussed kernels in **Module 1.5** (https://gurus.pyimagesearch.com/lessons

Kernels. We'll be using kernels for not only edge detection and gradients, but for averaging as well!

To accomplish our average blur, we'll actually be convolving our image with a $M \times N$ normalized filter where both M and N are both *odd* integers.

This kernel is going to slide from left-to-right and from top-to-bottom for each and every pixel in our input image. The pixel at the center of the kernel (and hence why we have to use an odd number, otherwise there would not be a true "center") is then set to be the *average* of all other pixels surrounding it.

Let's go ahead and define a 3×3 average kernel that can be used to blur the central pixel with a 3 pixel radius:

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Notice how each entry of the kernel matrix is *uniformly weighted* — we are giving equal weight to all pixels in the kernel. An alternative is to give pixels different weights, where pixels *farther from the central pixel* contribute *less* to the average; we'll discuss this method of smoothing in the **Gaussian blurring** section of this lesson.

Feedback

We could also define a 5×5 average kernel:

$$K = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

This kernel takes more pixels into account for the average, and will blur the image more than a 3×3 kernel since the kernel covers more area of the image.

Hence, this brings us to an important rule: **as the size of the kernel increases, so will the amount in which the image is blurred.**

Simply put: the larger your smoothing kernel is, the more blurred your image will look.

To investigate this notion, let's explore some code:

```
# import the necessary packages
import argparse
import cv2

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

# load the image, display it, and initialize the list of kernel sizes
image = cv2.imread(args["image"])
cv2.imshow("Original", image)
kernelSizes = [(3, 3), (9, 9), (15, 15)]

# loop over the kernel sizes and apply an "average" blur to the image
for (kX, kY) in kernelSizes:
    blurred = cv2.blur(image, (kX, kY))
    cv2.imshow("Average ({}, {})".format(kX, kY), blurred)
    cv2.waitKey(0)
```

Lines 1-12 handle our standard process of importing packages, setting up the argument parser, and loading our image from disk.

We then define a list of `kernelSize` on **Line 13** — the sizes of these kernels increase progressively so we'll be able to visualize the impact the kernel size has on the output image.

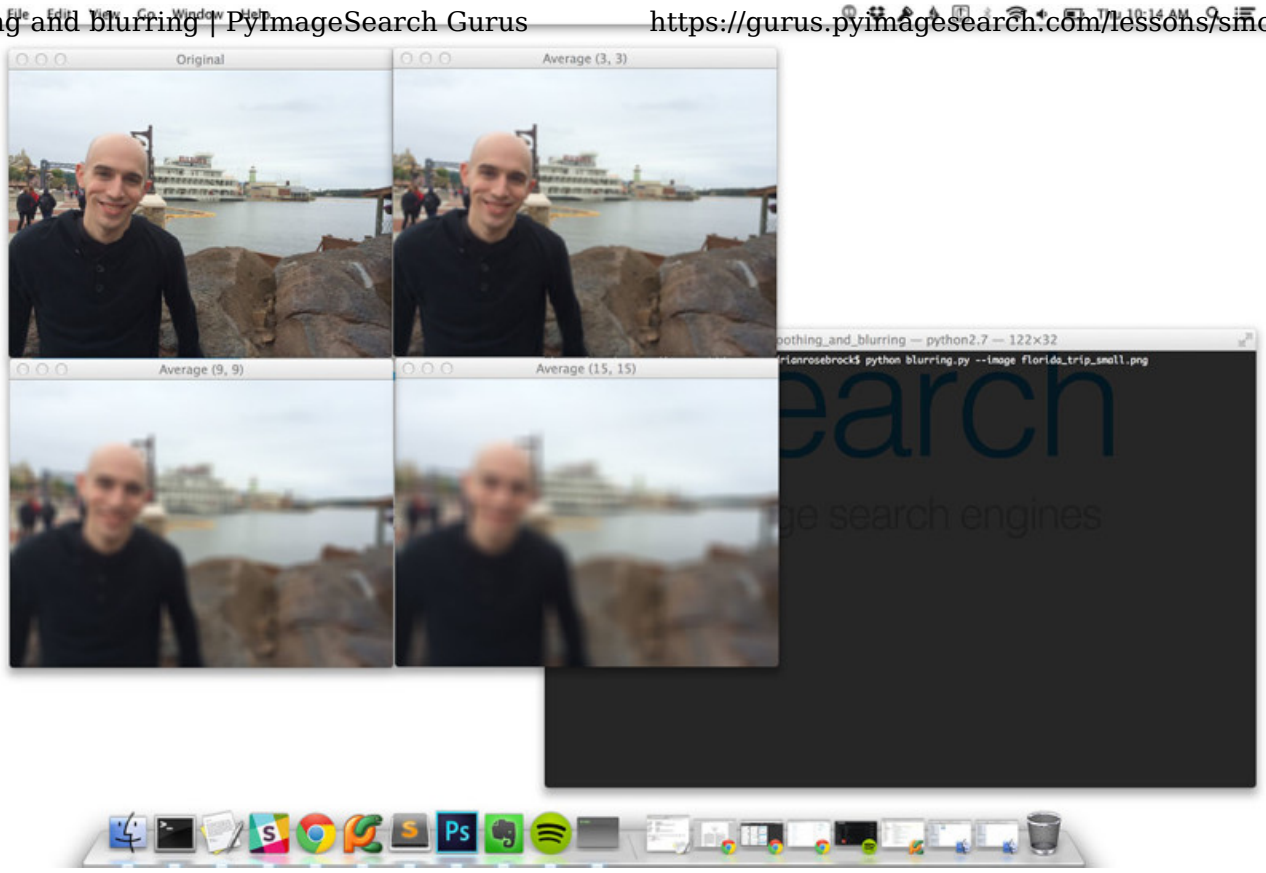
From there, we start looping over each of the kernel sizes on **Line 16**.

In order to average blur an image, we use the `cv2.blur` function. This function requires two arguments: the image we want to blur and the size of the kernel. As **Lines 17-19** show, we blur our image with increasing sizes kernels. The larger our kernel becomes, the more blurred our image will appear.

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

blurring.py — Shell

```
$ python blurring.py --image florida_trip_small.png
```



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

FIGURE 1: SMOOTHING AN IMAGE USING AN AVERAGE BLUR. NOTICE AS HOW THE KERNEL SIZE INCREASES, THE IMAGE BECOMES PROGRESSIVELY MORE BLURRED.

Feedback

On the *top-left* we have our original input image. On the *top-right* we have blurred it using a 3×3 kernel. The image is only slightly more blurred at this point, and the only noticeable area of the blur is around the facial region. However, by the time we get to a kernel size of 9×9 and 15×15, the image becomes practically unrecognizable.

Again, as the size of your kernel increases, your image will become progressively more blurred. This could easily lead to a point where you lose the edges of important structural objects in the image. Choosing the right amount of smoothing is critical when developing your own computer vision applications.

While average smoothing was quite simple to understand, it also weights each pixel inside the kernel area equally — and by doing this it becomes easy to over-blur our image and miss out on important edges. We can remedy this problem by applying Gaussian blurring.

Gaussian

Next up, we are going to review Gaussian blurring. Gaussian blurring is similar to average blurring, but

Instead of using a simple mean, we are now using a weighted mean, where neighborhood pixels that are closer to the central pixel contribute more "weight" to the average. And as the name suggests, Gaussian smoothing is used to remove noise that approximately follows a Gaussian distribution.

The end result is that our image is less blurred, but more **naturally** blurred, than using the average method discussed in the previous section. Furthermore, based on this weighting we'll be able to preserve more of the edges in our image as compared to average smoothing.

Just like an average blurring, Gaussian smoothing also uses a kernel of $M \times N$, where both M and N are odd integers.

However, since we are weighting pixels based on how far they are from the central pixel, we need an equation to construct our kernel. The equation for a Gaussian function in one direction is:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

And it then becomes trivial to extend this equation to two directions, one for the *x*-axis and the other for the *y*-axis, respectively:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where x and y are the respective distances to the horizontal and vertical center of the kernel and σ is the standard deviation of the Gaussian kernel.

Feedback

Again, as we'll see in the code below, when the size of our kernel increases so will the amount of blurring that is applied to our output image. However, the blurring will appear to be more "natural" and will preserve edges in our image better than simple average smoothing:

blurring.py — Python

```
21 # close all windows to cleanup the screen
22 cv2.destroyAllWindows()
23 cv2.imshow("Original", image)
24
25 # loop over the kernel sizes and apply a "Gaussian" blur to the image
26 for (kX, kY) in kernelSizes:
27     blurred = cv2.GaussianBlur(image, (kX, kY), 0)
28     cv2.imshow("Gaussian ({}, {})".format(kX, kY), blurred)
29     cv2.waitKey(0)
```

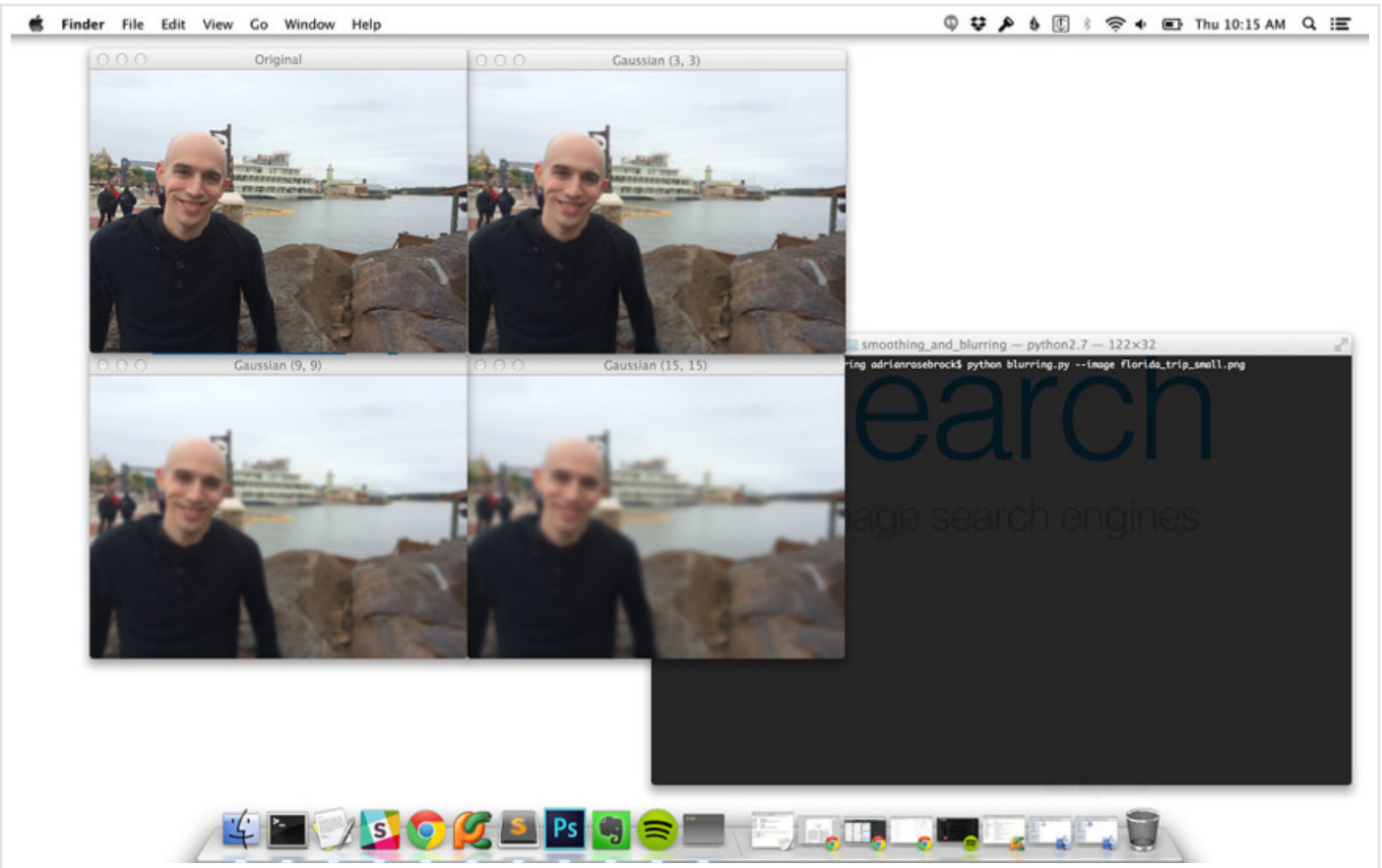
Lines 22 and 23 simply close all open windows and display our original image as a reference point.

The actual Gaussian blur takes place on **Lines 26-29** by using the `cv2.GaussianBlur` function.

arguments to the function is the image we want to blur. Then, similar to `cv2.blur`, we provide a tuple representing our kernel size. Again, we start with a small kernel size of 3×3 and start to increase it.

The last parameter is our σ, the standard deviation of the Gaussian distribution. By setting this value to *0*, we are instructing OpenCV to automatically compute σ based on our kernel size. In most cases, you'll want to let your σ be computed for you. But in the case you want to supply σ for yourself, I would suggest reading through the OpenCV documentation (http://docs.opencv.org/modules/imgproc/doc/filtering.html#gaussianblur) on `cv2.GaussianBlur` to make sure you understand the implications.

We can see the output of our Gaussian blur in **Figure 2** below:



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

FIGURE 2: APPLYING GAUSSIAN BLURRING.

Our images have less of a blur effect than when using the averaging method in **Figure 1**; however, the blur itself is more natural due to the computation of the weighted mean, rather than allowing all pixels in the kernel neighborhood to have equal weight.

In general, I tend to recommend starting with a simple Gaussian blur and tuning your parameters as needed. While the Gaussian blur is slightly slower than a simple average blur (and only by a tiny fraction),

a Gaussian blur tends to give more pleasing results, especially when applied to natural images.

Median

Traditionally, the median blur method has been most effective when removing salt-and-pepper noise. This type of noise is exactly what it sounds like: imagine taking a photograph, putting it on your dining room table, and sprinkling salt and pepper on top of it. Using the median blur method, you could remove the salt and pepper from your image.

When applying a median blur, we first define our kernel size . Then, as in the averaging blurring method, we consider all pixels in the neighborhood of size $K \times K$ where K is an odd integer. Notice how, unlike average blurring and Gaussian blurring where the kernel size could be *rectangular*, the kernel size for the median must be *square*. Furthermore (unlike the averaging method), instead of replacing the central pixel with the average of the neighborhood, we instead replace the central pixel with the median of the neighborhood.

The reason median blurring is more effective at removing salt-and-pepper style noise from an image is that each central pixel is always replaced with a pixel intensity that exists in the image. And since the median is robust to outliers, the salt-and-pepper noise will be less influential to the median than another statistical method, such as the average.

Feedback

Again, methods such as averaging and Gaussian compute means or weighted means for the neighborhood — this average pixel intensity may or may not be present in the neighborhood. But by definition, the median pixel *must* exist in our neighborhood. By replacing our central pixel with a median rather than an average, we can substantially reduce noise.

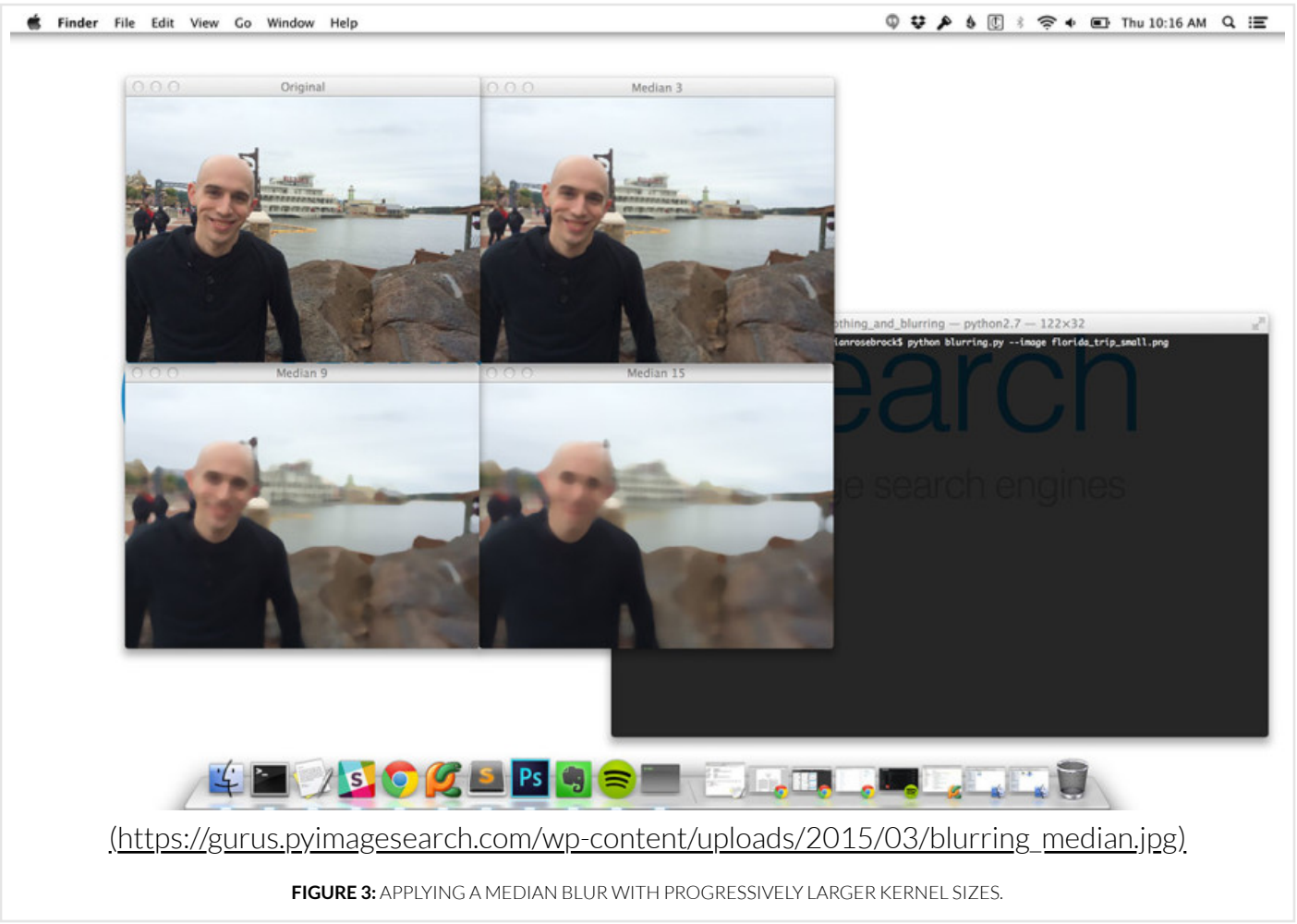
Now, it's time to apply our median blur:

blurring.py Python

```
31 # close all windows to cleanup the screen
32 cv2.destroyAllWindows()
33 cv2.imshow("Original", image)
34
35 # loop over the kernel sizes and apply a "Median" blur to the image
36 for k in (3, 9, 15):
37     blurred = cv2.medianBlur(image, k)
38     cv2.imshow("Median {}".format(k), blurred)
39     cv2.waitKey(0)
```

Applying a median blur is accomplished by making a call to the `cv2.medianBlur` function. This method takes two parameters: the image we want to blur and the size of our kernel.

On **Line 36** we start looping over square kernel sizes. We start off with a kernel size of 9 and then increase it to 9 and 15. The resulting median blurred images are then stacked and displayed to us can be seen in **Figure 3** below:



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

FIGURE 3: APPLYING A MEDIAN BLUR WITH PROGRESSIVELY LARGER KERNEL SIZES.

Notice that we are no longer creating a "motion blur" effect like in averaging and Gaussian blurring — instead, we are removing ***substantially more*** detail and noise.

For example, take a look at the color of the rocks to the right of myself in the image. As our kernel size increases, detail and color on the rocks become substantially less pronounced. By the time we are using a 15×15 kernel the rocks have lost almost all detail and look like a big "blob".

The same can be said for my face in the image — as the kernel size increases, my face rapidly loses detail and practically blends together.

The median blur is by no means a "natural blur" like Gaussian smoothing. However, for damaged images or photos captured under highly sub-optimal conditions, a median blur can really help as a pre-processing step prior to passing the image along to other methods, such as thresholding and edge

Feedback

Bilateral

The last method we are going to explore is bilateral blurring.

Thus far, the intention of our blurring methods have been to reduce noise and detail in an image; however, as a side effect we have tended to lose edges in the image.

In order to reduce noise while still maintaining edges, we can use bilateral blurring. Bilateral blurring accomplishes this by introducing two Gaussian distributions.

The first Gaussian function only considers spatial neighbors. That is, pixels that appear close together in the *(x, y)*-coordinate space of the image. The second Gaussian then models the pixel intensity of the neighborhood, ensuring that only pixels with similar intensity are included in the actual computation of the blur.

Intuitively, this makes sense. If pixels in the same (small) neighborhood have a similar pixel value, then they likely represent the same object. But if two pixels in the same neighborhood have contrasting values, then we could be examining the edge or boundary of an object — and we would like to preserve this edge.

Overall, this method is able to preserve edges of an image, while still reducing noise. The largest downside to this method is that it is considerably slower than its averaging, Gaussian, and median blurring counterparts.

Let's create a new file, name it `bilateral.py` , and look at some code to accomplish bilateral filtering:

bilateral.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, display it, and construct the list of bilateral
11 # filtering parameters that we are going to explore
12 image = cv2.imread(args["image"])
13 cv2.imshow("Original", image)

14 params = [(11, 21, 7), (11, 41, 21), (11, 61, 39)]
15
16 # loop over the diameter, sigma color, and sigma space
17 for (diameter, sigmaColor, sigmaSpace) in params:
18     # apply bilateral filtering and display the image
19     blurred = cv2.bilateralFilter(image, diameter, sigmaColor, sigmaSpace)
20     title = "Blurred d={}, sc={}, ss={}".format(diameter, sigmaColor, sigmaSpace)
21     cv2.imshow(title, blurred)
22     cv2.waitKey(0)
```

We'll start off by importing our packages, setting up our argument parser, and loading our image from disk.

We then define a list of blurring parameters on **Line 14**. These parameters correspond to the *diameter*, σ_{color}, and σ_{space} of the bilateral filter, respectively.

From there, we loop over each of these parameter sets on **Line 17** and apply bilateral filtering by making a call to the `cv2.bilateralFilter` on **Line 19**. Finally, **Lines 20-22** display our blurred image to our screen.

Let's take a second and review the parameters we supply to `cv2.bilateralFilter`. The first parameter we supply is the image we want to blur. Then, we need to define the *diameter* of our pixel neighborhood — the larger this diameter is, the more pixels will be included in the blurring computation. Think of this parameter as a square kernel size.

The third argument is our color standard deviation, denoted as σ_{color}. A larger value for σ_{color} means that more colors in the neighborhood will be considered when computing the blur. If we let σ_{color} get too large in respect to the *diameter*, then we essentially have broken the assumption of bilateral filtering — that only pixels of similar color should contribute significantly to the blur.

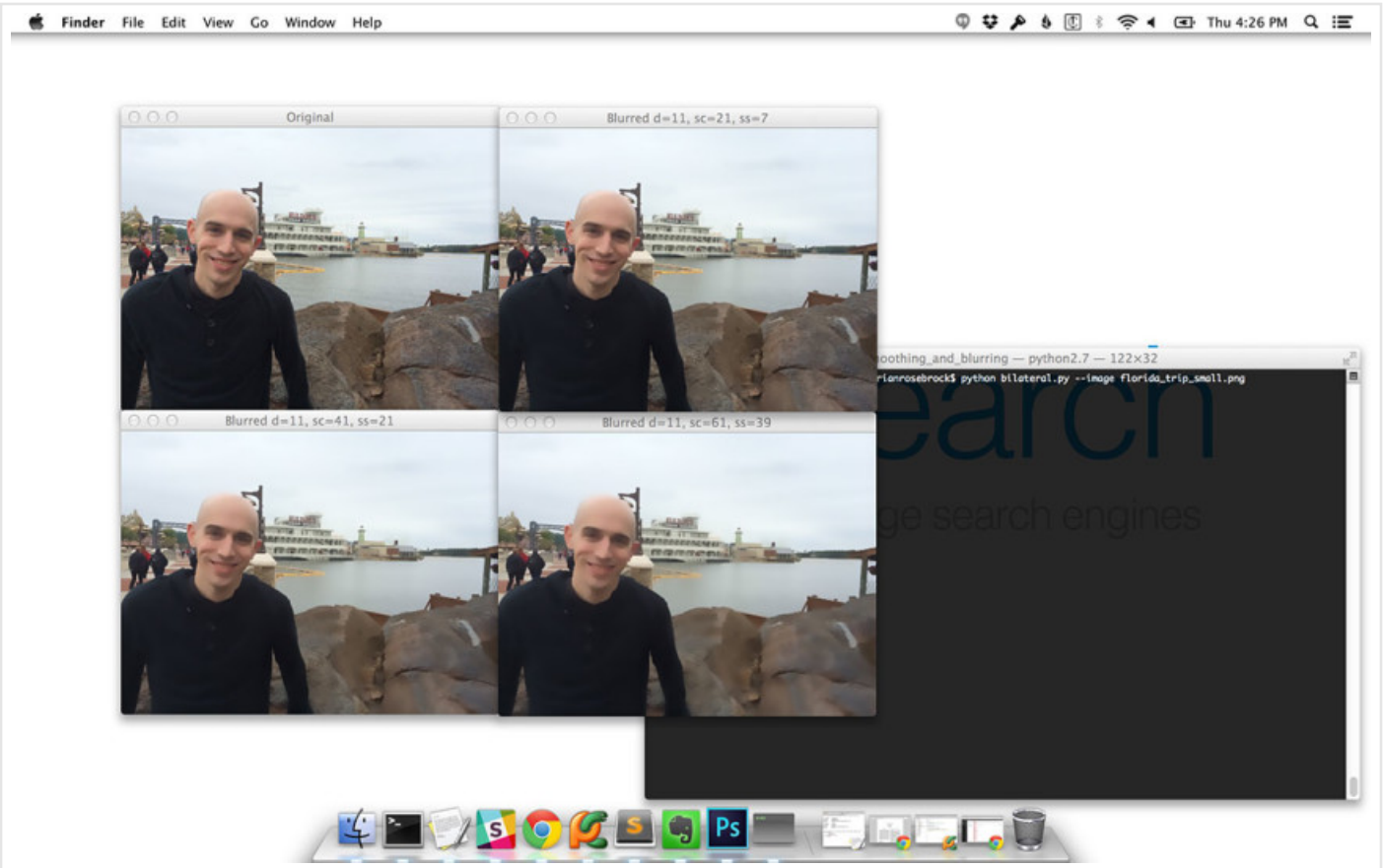
Finally, we need to supply the space standard deviation, which we call σ_{space}. A larger value of σ_{space} means that pixels farther out from the central pixel *diameter* will influence the blurring calculation.

Feedback

To view the results of our bilateral blurring, just execute the following command:

```
bilateral.py                                                    Shell
1 $ python bilateral.py --image florida_trip_small.png
```

And you'll see the following output:



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

FIGURE 5: APPLYING BILATERAL FILTERING TO OUR IMAGE.

On the *top-left* we have our original input image. And on the *top-right* we start off with a diameter of $d = 11$ pixels, $\sigma_{color} = 21$, and $\sigma_{space} = 7$. The effects of our blurring are not fully apparent yet, but if you zoom in on the rocks and compare them to our original image, you'll notice that much of the texture has disappeared! The rocks appear much smoother, as if they have been eroded and smoothed over years and years of rushing water. However, the edges and boundaries between the lake and the rocks is *clearly* maintained.

Now, take a look at the *bottom-left* where we have increased both σ_{color} and σ_{space} jointly. At this point we can really see the effects of bilateral filtering. The buttons on my black hoodie have practically disappeared and nearly all detail and wrinkles on my skin have been removed. Yet at the same time there is *still* a clear boundary between myself and the background of the image. If we were using an average or

Gaussian blur, the background would be mixing with the foreground.

Finally, we have the *bottom-right* where I have increased σ_{color} and σ_{space} yet again, just to demonstrate how powerful of a technique bilateral filtering is. Now nearly all detail and texture from the rocks, water, sky, and my skin and hoodie are gone. It's also starting to look as if the number of colors in the image have been reduced. Again, this is an exaggerated example and you likely wouldn't be applying *this* much blurring to an image, but it does demonstrate the effect bilateral filtering will have on your edges: dramatically smoothed detail and texture, while still preserving boundaries and edges.

So there you have it — an overview of blurring techniques! If it's not entirely clear when you will use each blurring or smoothing method just yet, that's okay. We will be building on these blurring techniques substantially in this course and you'll see plenty of examples on when to apply each type of blurring. For the time being, just try to digest the material and store blurring and smoothing as yet another tool in your toolkit.

Summary

In this lesson we reviewed how to smooth and blur images. We started by discussing the role kernels play in smoothing and blurring.

We then reviewed the four primary methods to smooth an image in OpenCV: *averaging*, *Gaussian blurring*, *median filtering*, and *bilateral filtering*.

The simple average method is fast, but may not preserve edges in images.

Applying a Gaussian blur is better at preserving edges, but is slightly slower than the average method.

A median filter is primarily used to reduce salt-and-pepper style noise as the median statistic is much more robust and less sensitive to outliers than other statistical methods such as the mean.

Finally, the bilateral filter preserves edges, but is *substantially* slower than the other methods. Bilateral filtering also boasts the most parameters to tune which can become a nuisance to tune correctly.

In general, I recommend starting with a simple Gaussian blur to obtain a baseline and then going from there.

Downloads:

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

	Quizzes	Status
1	Smoothing and Blurring Quiz (https://gurus.pyimagesearch.com/quizzes/smoothing-and-blurring-quiz/)	

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

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