1.10.2: Edge detection

Topic Progress: (https://gurus.pyimagesearch.com/topic/gradients/)

(https://gurus.pyimagesearch.com/topic/edge-detection/)

← Back to Lesson (https://gurus.pyimagesearch.com/lessons/gradients-and-edge-detection/)

In the previous lesson (https://gurus.pyimagesearch.com/topic/gradients/) we discussed *image gradients* and how they are one of the fundamental building blocks of computer vision and image processing.

Today, we are going to see just how important image gradients are; specifically, by examining the *Canny edge detector*.

The Canny edge detector is arguably the *most well known* ***and*** *the most used* edge detector in all of computer vision and image processing. While the Canny edge detector is not exactly "trivial" to understand, we'll break down the steps into bite-sized pieces so we can understand what is going on under the hood.

Fortunately for us, since the Canny edge detector is so widely used in almost all computer vision applications, OpenCV has already implemented it for us in the `cv2.Canny` function. We'll also explore how to use this function to detect edges in images of our own.

Objectives:

By the end of this lesson you will understand:

1. What the Canny edge detector is and how it is used.

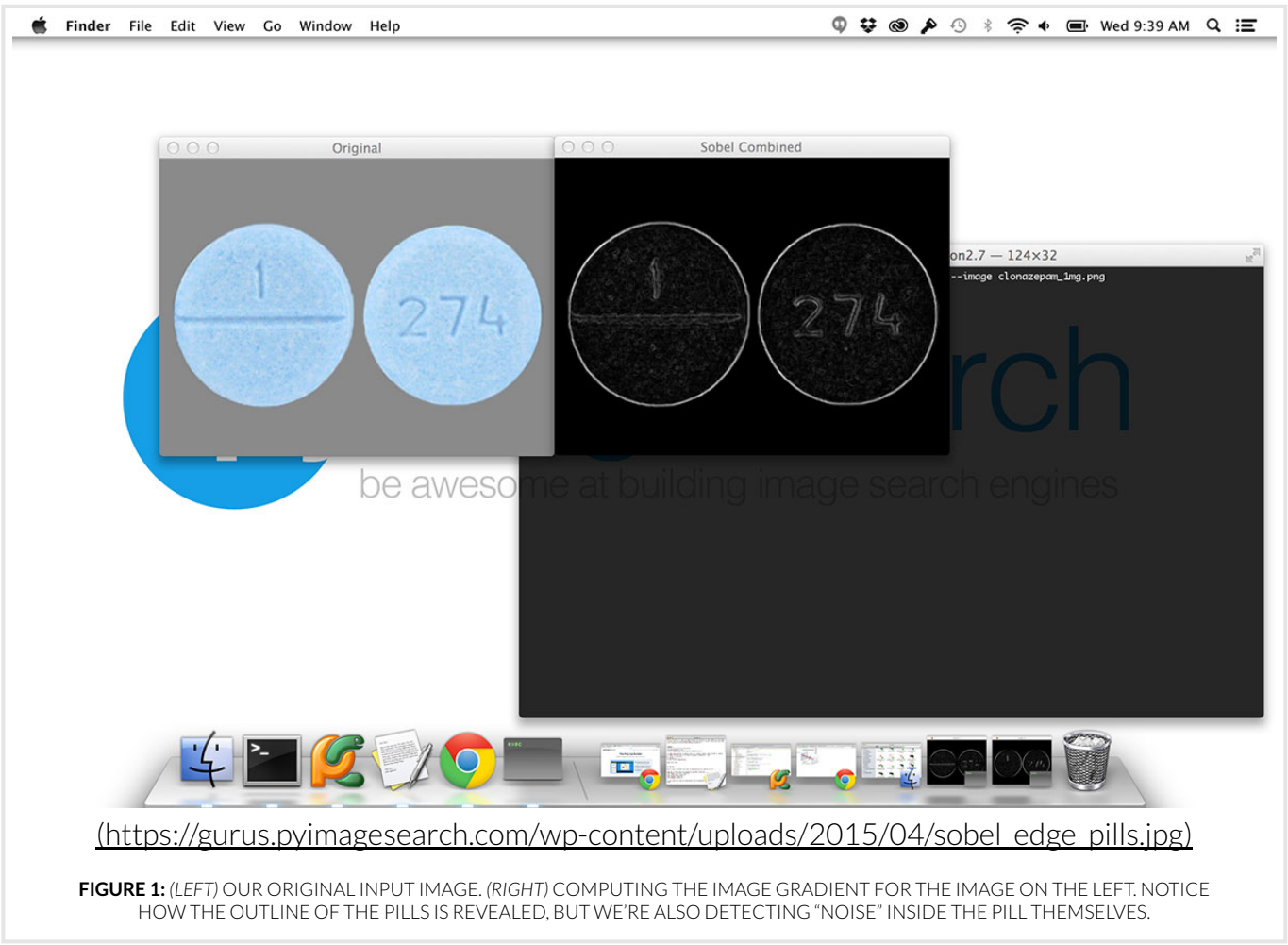
2. The basic steps of the Canny edge detector.
3. How to use the `cv2.Canny` function to detect edges in images.
4. How to extend the Canny edge detector to create the `auto_canny` , a zero parameter edge detector which is much easier to use.

Edge detection

As we discovered in the previous lesson, the gradient magnitude and orientation allow us to reveal the *structure* of objects in an image. In later lessons, we'll see both the gradient orientation and magnitude make for excellent image features when quantifying an image.

However, for the process of edge detection, the gradient magnitude is extremely sensitive to noise.

For example, let's examine the gradient representation of the following image:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/sobel_edge_pills.jpg)

FIGURE 1: *(LEFT)* OUR ORIGINAL INPUT IMAGE. *(RIGHT)* COMPUTING THE IMAGE GRADIENT FOR THE IMAGE ON THE LEFT. NOTICE HOW THE OUTLINE OF THE PILLS IS REVEALED, BUT WE'RE ALSO DETECTING "NOISE" INSIDE THE PILL THEMSELVES.

On the *left* we have our original input image of a frontside and backside. And on the *right* we have the image gradient representation.

As you can see, the gradient representation is a bit *noisy*. Sure, we have been able to detect the actual outline of the pills. But we're also left with a lot of "noise" inside the pills itself representing the pill imprint.

So what if we wanted to detect *just the outline* of the pills?

That way, if we had *just* the outline, we could extract the pills from the image using something like **contours** (https://gurus.pyimagesearch.com/lessons/contours/). Wouldn't that be nice?

Unfortunately, simple image gradients are not going to allow us to (easily) achieve our goal.

Instead, we'll have to use the image gradients as building blocks to create a more robust method to detect edges — the Canny edge detector.

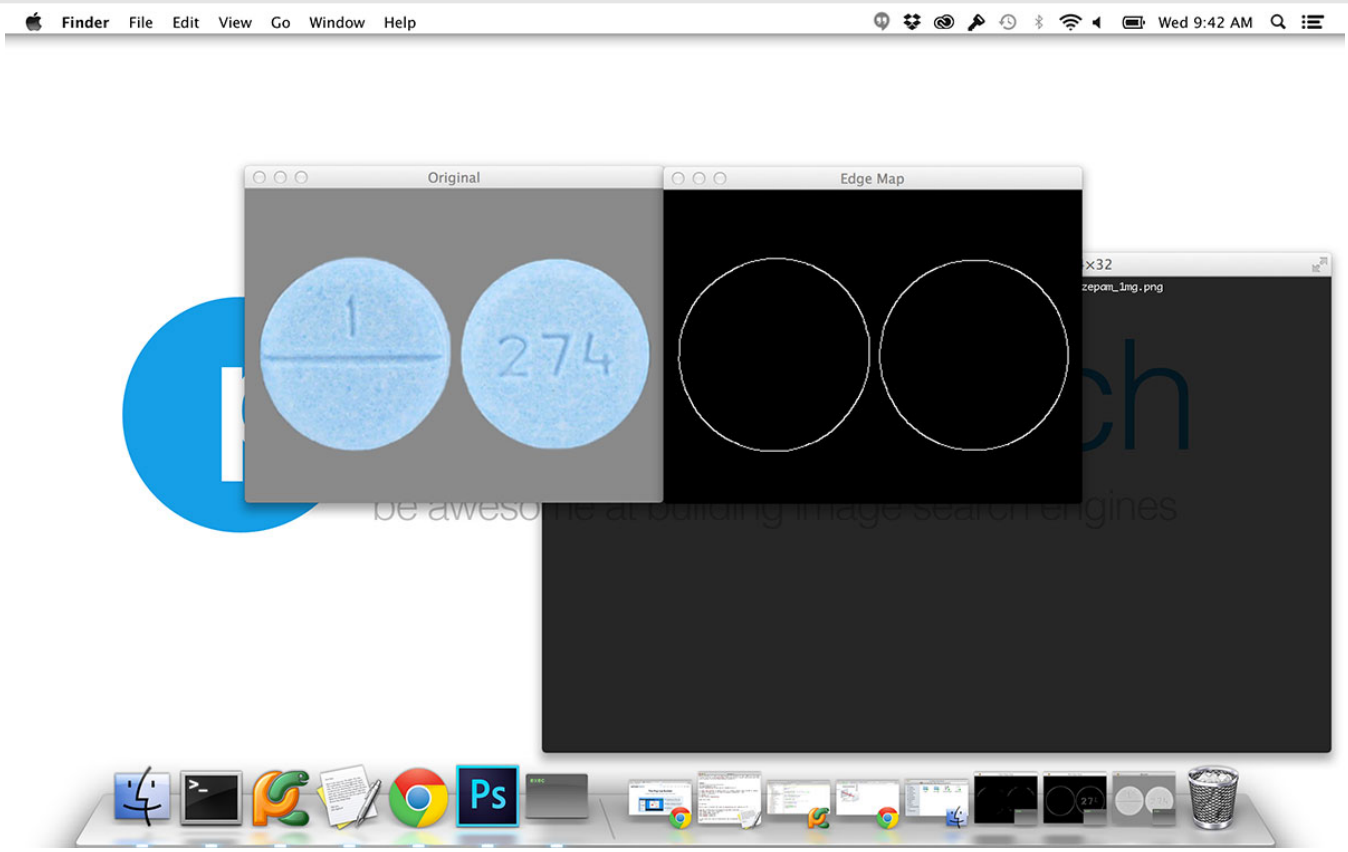
The Canny edge detector

The Canny edge detector is a multi-step algorithm used to detect a wide range of edges in images. The algorithm itself was introduced by John F. Canny in his 1986 paper, *A Computational Approach To Edge Detection*.

If you look inside many image processing projects, you'll most likely see the Canny edge detector being called somewhere in the source code. While image gradients tend to be important building blocks for more advanced techniques like the Canny edge detector, the Canny edge detector is also a building block for methods such as object detection in images. Whether we are finding the distance from our camera to an object (https://gurus.pyimagesearch.com/lessons/measuring-distance-from-camera-to-object-in-image/), building a mobile document scanner (http://www.pyimagesearch.com/2014/09/01/build-kick-ass-mobile-document-scanner-just-5-minutes/), or finding a Game Boy screen in an image (http://www.pyimagesearch.com/2014/04/21/building-pokedex-python-finding-game-boy-screen-step-4-6/), the Canny edge detector will often be found as an important pre-processing step.

More formally, an ***edge*** is defined as **discontinuities in pixel intensity**, or more simply, **a *sharp* difference and change in pixel values.**

Here is an example of applying the Canny edge detector to detect edges in our pill image from above:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/canny_edge_map.jpg)

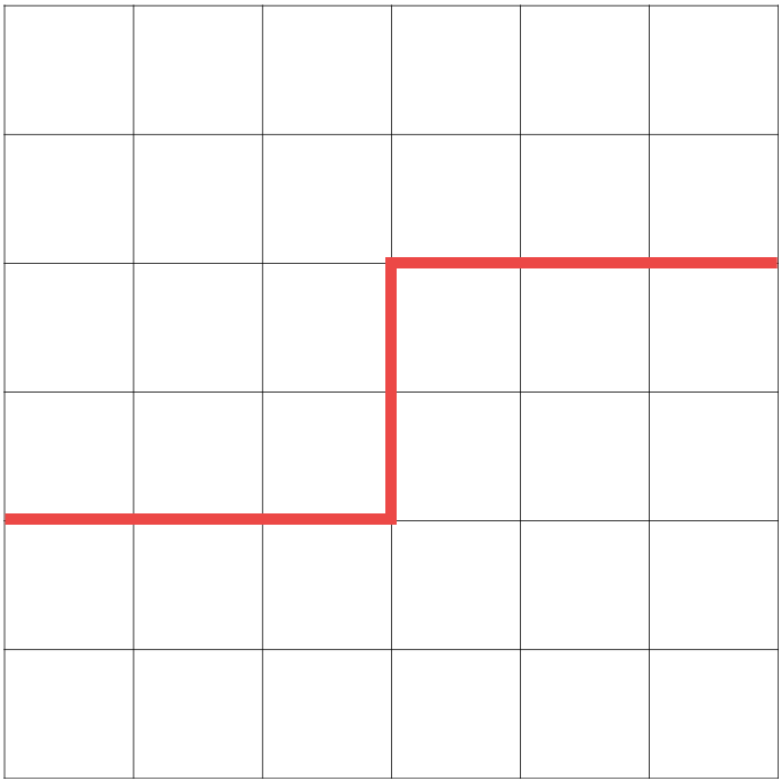
FIGURE 2: COMPUTING A CANNY EDGE MAP FOR OUR PILL IMAGE.

On the *left* we have our original input image. And on the *right* we have the edged image, or what is commonly called the **edge map**. Notice how we have *only* the outlines of the pill as a clear, thin white line — there is no longer any "noise" inside the pills themselves.

Before we dive deep into the Canny edge detection algorithm, let's start by looking what *types of edges* there are in images:

Step Edge:

A *step edge* forms when there is an abrupt change in pixel intensity from one side of the discontinuity to the other. Take a look at the following graph for an example of a step edge:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/step_edge.png)

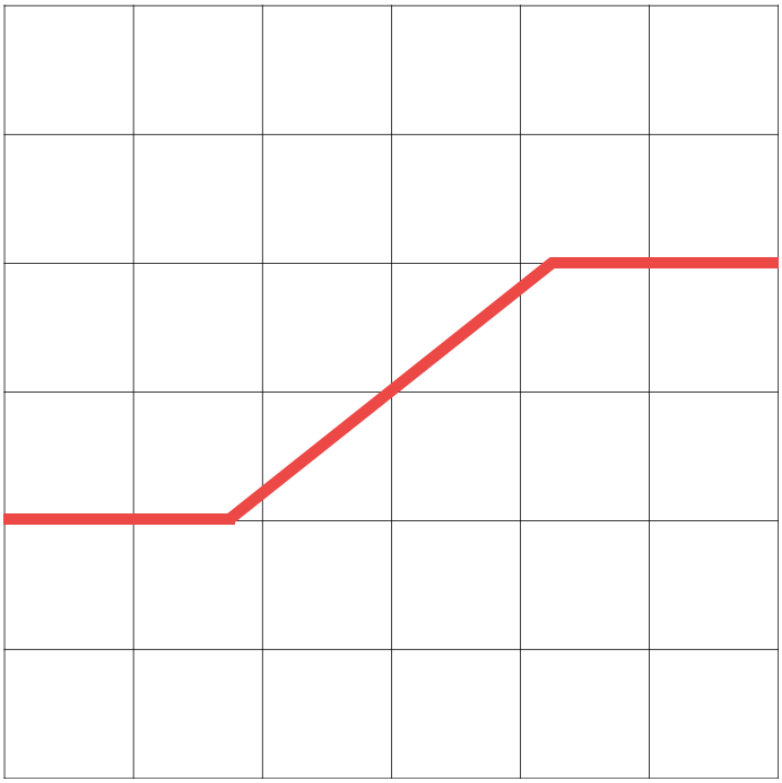
FIGURE 3: AN EXAMPLE OF A STEP EDGE, AS IF WE ARE TAKING THE NEXT STEP ON A FLIGHT OF STAIRS.

As the name suggests, the graph actually looks like a step — there is a *sharp* θ° step in the graph, indicating an abrupt change in pixel value. These types of edges tend to be easy to detect.

Ramp Edge:

A *ramp edge* is like a *step edge*, only the change in pixel intensity is not instantaneous. Instead, the change in pixel value occurs a short, but finite distance.

Here we can see an edge that is slowly "ramping" up in change, but the change in intensity is not immediate like in a *step edge*:

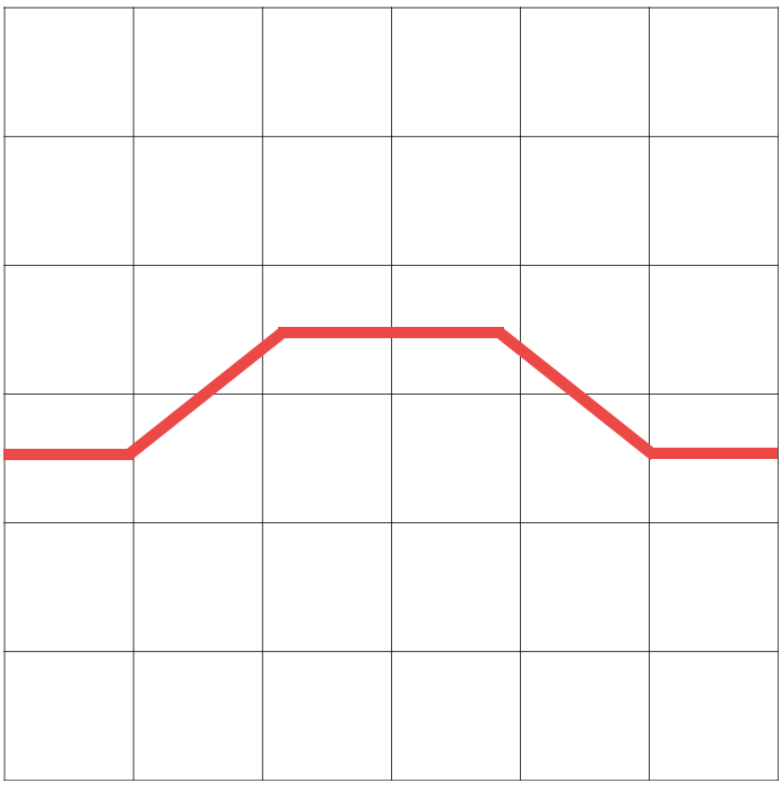


(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/ramp_edge.png)

FIGURE 4: AN EXAMPLE OF A RAMP EDGE, AS IF WE ARE DRIVING OUR CAR UP A STEEP HILL.

Ridge Edge:

A *ridge edge* is similar to combining two *ramp edges*, one bumped right against another. I like to think of *ramp edges* as driving up and down a large hill or mountain:



(https://gurus.pyimagesearch.com/wp-

content/uploads/2015/04/ridge_edge.png)

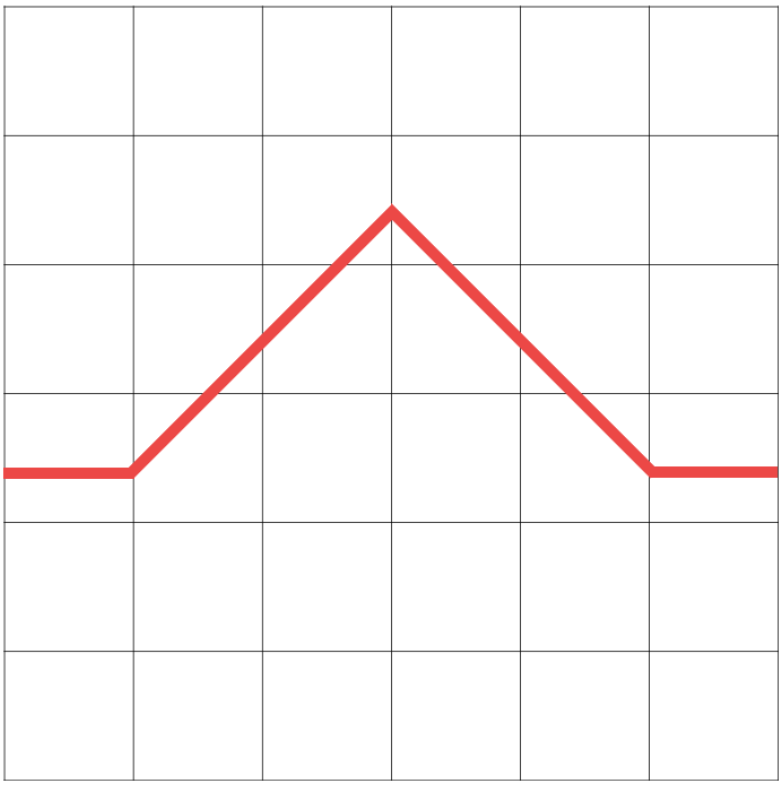
FIGURE 4: A RIDGE EDGE IS LIKE A RAMP EDGE, ONLY AS OUR CAR GETS TO THE TOP OF THE HILL, IT DRIVES ALONG THE PLATEAU A BIT, BEFORE DRIVING BACK DOWN THE HILL.

First, you slowly ascend the mountain. Then you reach the top where it levels out for a short period. And then you're riding back down the mountain.

In the context of edge detection, a *ridge edge* occurs when image intensity abruptly changes, but then returns to the initial value after a short distance.

Roof Edge:

Lastly we have the *roof edge*, which is a type of *ridge edge*:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/roof_edge.png)

FIGURE 5: IN A ROOF EDGE, THERE IS NO PLATEAU AT THE TOP. WE SIMPLY DRIVE TO THE TOP OF THE HILL AND IMMEDIATELY HEAD BACK DOWN.

Unlike the *ridge edge* where there is a short, finite plateau at the top of the edge, the *roof edge* has no such plateau. Instead, we slowly ramp up on either side of the edge, but the very top is a pinnacle and we simply fall back down the bottom.

Canny in a nutshell

Now that we have reviewed the various types of edges in an image, let's discuss the actual Canny edge detection algorithm, which is a multi-step process consisting of:

1. Applying Gaussian smoothing to the image to help reduce noise.
2. Computing the G_x and G_y image gradients using the Sobel kernel.
3. Applying non-maxima suppression to keep only the local maxima of gradient magnitude pixels that are pointing in the direction of the gradient.
4. Defining and applying the T_{upper} and T_{lower} thresholds for Hysteresis thresholding.

Let's discuss each of these steps.

Step 1: Gaussian smoothing

This step is fairly intuitive and straightforward. As we learned from our lesson on **smoothing and blurring** (https://gurus.pyimagesearch.com/lessons/smoothing-and-blurring/), smoothing an image allows us to ignore much of the *detail* and instead focus on the actual *structure*.

This also makes sense in the context of edge detection — we are not interested in the actual *detail* of the image. Instead, we want to apply edge detection to find the *structure* and *outline* of the objects in the image so we can further process them.

Step 2: Gradient orientation and magnitude

Now that we have a smoothed image, we can compute the gradient orientation and magnitude, just like we did in the previous lesson (https://gurus.pyimagesearch.com/topic/gradients/).

However, as we have seen, the gradient magnitude is quite susceptible to noise and does not make for the best edge detector. We need to add two more steps on to the process to extract better edges.

Step 3: Non-maxima Suppression

Non-maxima suppression sounds like a complicated process, but it's really not — it's simply an **edge thinning** process.

After computing our gradient magnitude representation, the edges themselves are still quite noisy and blurred, but in reality there should only be *one* edge response for a given region, not a whole clump of pixels reporting themselves as edges.

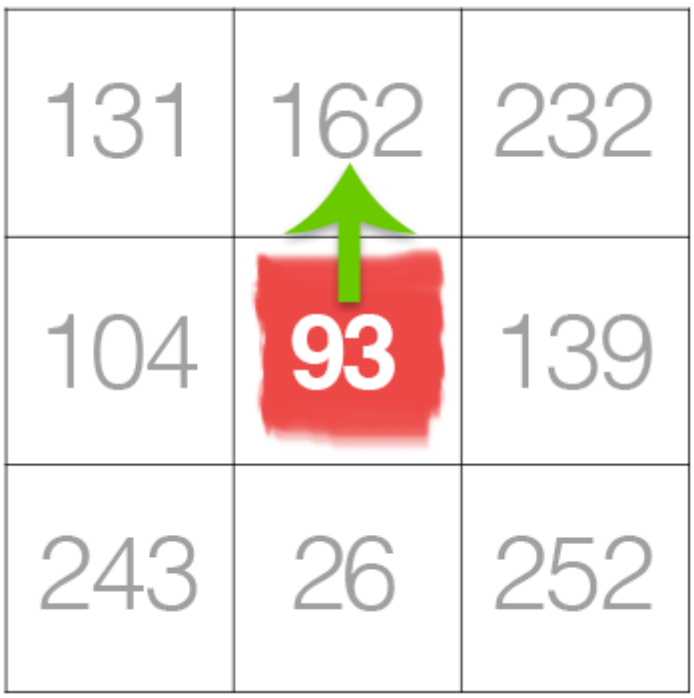
To remedy this, we can apply edge thinning using non-maxima suppression. To apply non-maxima suppression we need to examine the gradient magnitude G and orientation θ at each pixel in the image and:

1. Compare the current pixel to the 3×3 neighborhood surrounding it.
2. Determine in which direction the orientation is pointing:
    1. If it's pointing towards the north or south, then examine the north and south magnitude.
    2. If the orientation is pointing towards the east or west, then examine the east and west pixels.

3. If the center pixel magnitude is ***greater than*** both the pixels it is being compared to, then preserve the magnitude. Otherwise, discard it.

Some implementations of the Canny edge detector *round* the value of θ to either $0°$, $45°$, $90°$, or $135°$, and then use the rounded angle to compare not only the north, south, east, and west pixels, but also the corner top-left, top-right, bottom-right, and bottom-left pixels as well.

But, let's keep things simple and view an example of applying non-maxima suppression for an angle of $\theta = 90°$ degrees:



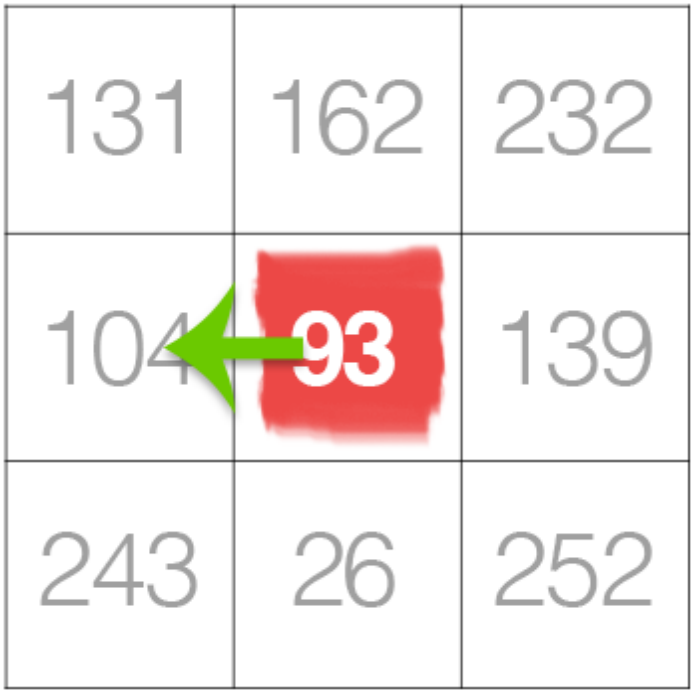
(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/canny_suppress_90_degrees.png)

FIGURE 6: LET'S PRETEND THE GRADIENT ORIENTATION FOR THIS IMAGE IS 90 DEGREES. APPLYING NON-MAXIMA SUPPRESSION, WE WOULD NEED TO EXAMINE THE NORTH AND SOUTH PIXELS AND TAKE THE MAXIMUM VALUE — WHICH IS 162.

In the example above we are going to pretend that the gradient orientation is $\theta = 90°$ (it's actually not, but that's okay, this is only an example).

Given that our gradient orientation is pointing north, we need to examine *both* the north and south pixels. The central pixel value of *93* is greater than the south value of *26*, so we'll discard the *26*. However, examining the north pixel we see that the value is *162* — we'll keep this value of *162* and suppress (i.e. set to *0*) the value of *93* since *93* < *162*.

Here's another example of applying non-maxima suppression for when $\theta = 180°$:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/canny_suppress_180_degrees.png)

FIGURE 7: APPLYING NON-MAXIMA SUPPRESSION IN THIS IMAGE ENDS IN US DISCARDING THE *93* AND KEEPING THE *104* AND *139* VALUES.

Notice how the central pixel is *less than both* the east and west pixels. According to our non-maxima suppression rules above (rule #3), we need to discard the pixel value of *93* and keep the east and west values of *104* and *139*, respectively.

As you can see, non-maxima suppression for edge detection is not as hard as it seems!

Step 4: Hysteresis thresholding

Finally, we have the hysteresis thresholding step. Just like non-maxima suppression, it's actually much easier than it sounds.

Even after applying non-maxima suppression, we may need to remove regions of an image that are not technically *edges*, but still responded as edges after computing the gradient magnitude and applying non-maximum suppression.

To ignore these regions of an image, we need to define two thresholds: T_{upper} and T_{lower}.

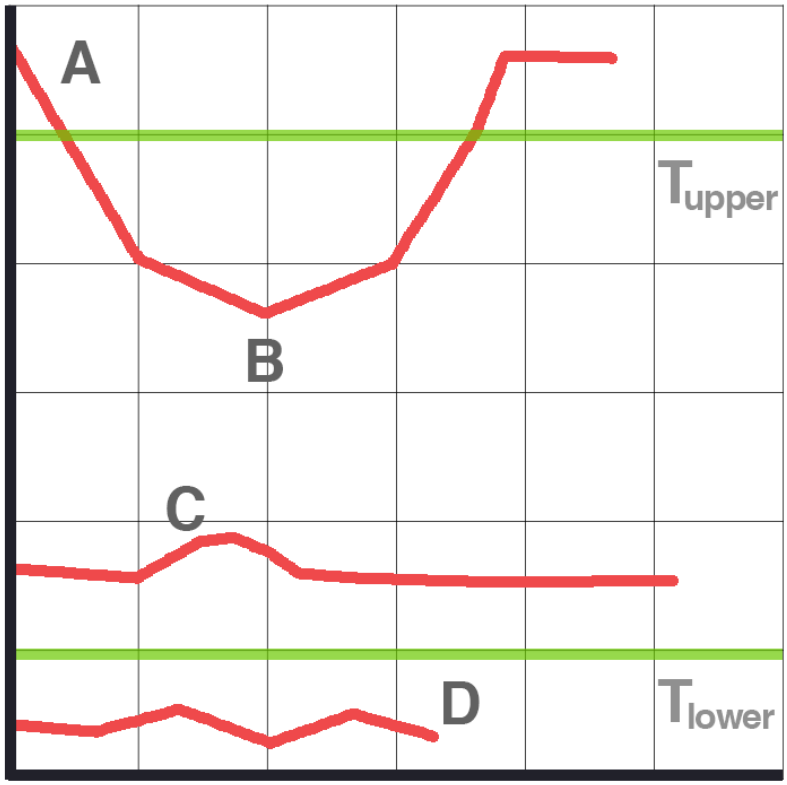
Any gradient value $G > T_{upper}$ is ***sure to be an edge***.

Any gradient value $G < T_{lower}$ is ***definitely not an edge***, so immediately discard these regions.

And any gradient value that falls into the range $T_{lower} < G < T_{upper}$ needs to undergo additional tests:

1. If the particular gradient value is connected to a **strong edge** (i.e. $G > T_{upper}$), then mark the pixel as an edge.
2. If the gradient pixel is *not* connected to a strong edge, then discard it.

Hysteresis thresholding is actually better explained visually:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/hysteresis.png)

FIGURE 8: AN EXAMPLE OF APPLYING HYSTERESIS TO A SET OF EDGES.

At the top of the graph we can see that **A** is a sure edge, since $A > T_{upper}$.

B is also an edge, even though $B < T_{upper}$ since it is connected to a sure edge, **A**.

C is not an edge since $C < T_{upper}$ and is not connected to a strong edge.

Finally, **D** is not an edge since $D < T_{lower}$ and is automatically discarded.

Setting these threshold ranges is not always a trivial process.

If the threshold range is **too wide** then we'll get many false edges instead of being about to find *just* the structure and outline of an object in an image.

Similarly, if the threshold range is **too tight**, we won't find many edges at all and could be at risk of missing the structure/outline of the object entirely!

Later on in this section I'll demonstrate how we can **automatically** tune these threshold ranges with practically zero effort. But for the time being, let's see how edge detection is actually performed inside OpenCV.

Open up a new file, name it `canny.py` , and insert the following code:

canny.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, 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, (5, 5), 0)
14
15 # show the original and blurred images
16 cv2.imshow("Original", image)
17 cv2.imshow("Blurred", blurred)
18
19 # compute a "wide", "mid-range", and "tight" threshold for the edges
20 wide = cv2.Canny(blurred, 10, 200)
21 mid = cv2.Canny(blurred, 30, 150)
22 tight = cv2.Canny(blurred, 240, 250)
23
24 # show the edge maps
25 cv2.imshow("Wide Edge Map", wide)
26 cv2.imshow("Mid Edge Map", mid)
27 cv2.imshow("Tight Edge Map", tight)
28 cv2.waitKey(0)
```


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

While Canny edge detection can be applied to an RGB image by detecting edges in each of the separate Red, Green, and Blue channels separately and combining the results back together, we *almost always* want to apply edge detection to a single channel, grayscale image (**Line 12**) — this ensures that there will be less noise during the edge detection process.

While the Canny edge detector does apply blurring prior to edge detection, we'll also want to (normally) apply extra blurring prior to the edge detector to further reduce noise and allow us to find the objects in an image (**Line 13**).

Lines 16 and 17 then display our original and blurred images to our screen.

Applying the `cv2.Canny` function to detect edges in then performed on **Lines 20-22**.

The first parameter to `cv2.Canny` is the image we want to detect edges in — in this case, our grayscale, blurred image. We then supply the T_{lower} and T_{upper} thresholds, respectively.

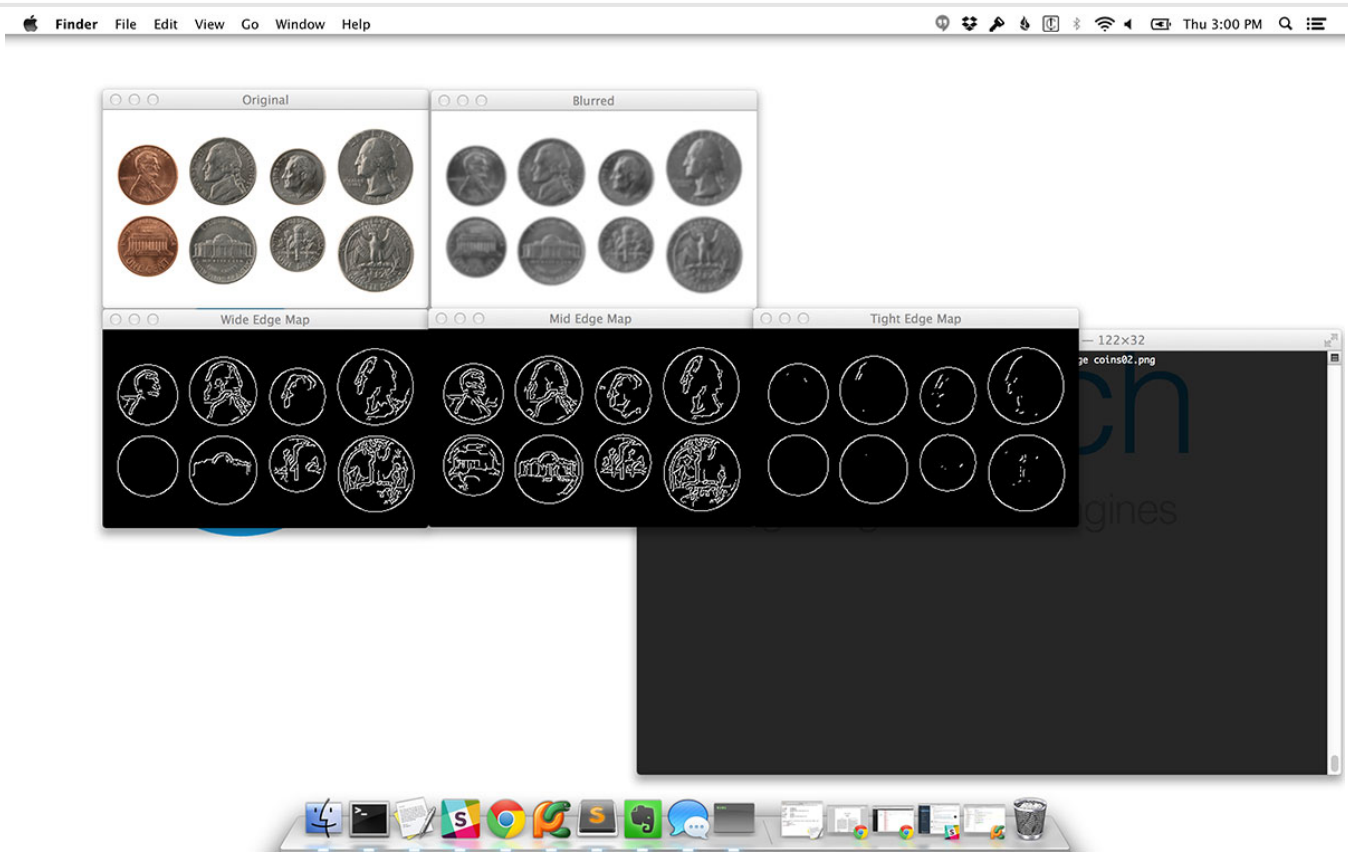
On **Line 20** we are applying a *wide* threshold, a *mid-range* threshold on **Line 21**, and a *tight* threshold on **Line 22**. You can convince yourself that these are *wide*, *mid-range*, and *tight* thresholds by plotting the threshold values on **Figure 9** below (in fact, you'll want to get some practice doing the plotting as I'll be asking you to identify various types of threshold types in the quiz at the bottom of the section).

Finally, **Lines 25-28** display the output edge maps to our screen.

To see our Canny edge detector in action, open up a terminal and issue the following command:

canny.py Shell

```
1 $ python canny.py --image coins02.png
```



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/canny_coins.jpg)

FIGURE 9: APPLYING CANNY EDGE DETECTION TO A COLLECTION OF COINS USING A WIDE RANGE, MID RANGE, AND TIGHT RANGE RANGE OF THRESHOLDS.

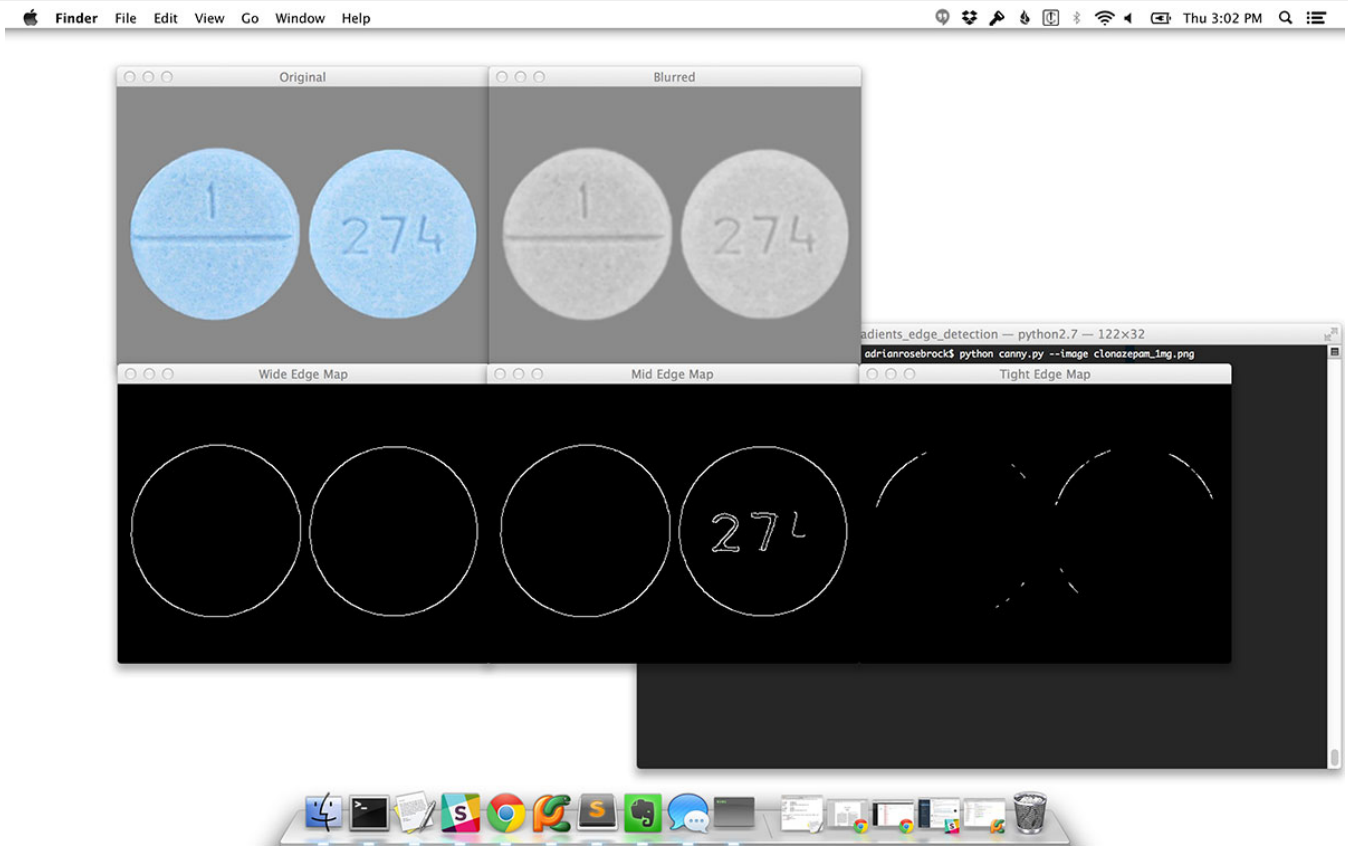
In the above figure, the *top-left* image is our input image of coins. We then blur the image slightly to help smooth details and aide in edge detection on the *top-right*.

Feedback

The wide range, mid-range range, and tight range edge maps are then displayed on the *bottom*, respectively. Using a wide edge map captures the outlines of the coins, but also captures many of the edges of faces and symbols inside the coins. The mid-range edge map also performs similarly. Finally, the tight range edge map is able to capture *just* the outline of the coins while discarding the rest.

Let's try another example:

```
canny.py                                                    Python
1 $ python canny.py --image clonazepam_1mg.png
```



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/canny_pill.jpg)

FIGURE 9: APPLYING CANNY EDGE DETECTION WITH A VARIETY OF HYSTERESIS THRESHOLDS TO AN IMAGE OF PILLS.

Unlike **Figure 8** above, the Canny thresholds for **Figure 9** gives us nearly reversed results. Using the wide range edge map, we are able to find the outlines of the pills. The mid-range edge map also gives us the outlines of the pills, but also some of the digits imprinted on the pill. Finally, the tight edge map does not help us at all — the outline of the pills is nearly completely lost.

As you can tell, depending on your input image you'll need dramatically different hysteresis threshold values — and tuning these values can be a real pain. You might be wondering, is there a way to *reliably* tune these parameters without simply guessing, checking, and viewing the results?

The answer is yes!

Keep reading the rest of this lesson to learn how we can *automatically* tune the hysteresis threshold parameters of the Canny edge detector.

Automatically tuning edge detection parameters

As we saw in the section above, the Canny edge detector requires two parameters: an *upper* and *lower* threshold used during the hysteresis step.

The problem becomes determining these lower and upper thresholds.

You may find your asking, *"What are the optimal values for the thresholds?"*

This question is especially important when you are processing multiple images with different contents captured under varying lighting conditions.

In the remainder of this lesson I'll show you a little trick that relies on ***basic statistics*** that you can apply to remove the manual tuning of the thresholds to Canny edge detection — this little trick will save you time in parameter tuning, and you'll still get a nice Canny edge map after applying the function.

The actual `auto_canny` function is already defined for us inside my imutils (https://github.com/jrosebr1/imutils/blob/master/imutils/convenience.py) package, but let's take a look at the function so we can understand what's going on:

Feedback

imutils.py — Python

```
110 def auto_canny(image, sigma=0.33):
111     # compute the median of the single channel pixel intensities
112     v = np.median(image)
113 
114     # apply automatic Canny edge detection using the computed median
115     lower = int(max(0, (1.0 - sigma) * v))
116     upper = int(min(255, (1.0 + sigma) * v))
117     edged = cv2.Canny(image, lower, upper)
118 
119     # return the edged image
120     return edged
```

We start by defining our `auto_canny` `function.` This function requires a single argument, `image`, which is the single-channel image that we want to detect images in. An optional argument, `sigma`, can be used to vary the percentage thresholds that are determined based on simple statistics.

Next up, we compute the median of the pixel intensities in the image. Unlike the mean, the median is less sensitive to outlier pixel values inside the image, thus making it a more stable and reliable statistic for automatically tuning threshold values.

We then take this median value and construct two thresholds, `lower` and `upper`. These thresholds are constructed based on the +/- percentages controlled by the `sigma` argument.

A lower value of `sigma` indicates a tighter threshold, whereas a larger value of `sigma` gives a wider threshold. In general, you will not have to change this `sigma` value often. Simply select a single, default `sigma` value and apply it to your entire dataset of images.

Note: *In practice,* `sigma=0.33` *tends to give good results on most of the dataset I'm working with, so I choose to supply 33% as the default sigma value.*

Now that we have our lower and upper thresholds, we then apply the Canny edge detector and return it to the calling function.

Let's keep moving with this example and see how we can apply our `auto_canny` function. Open up a new file, name it `auto_canny.py`, and insert the following code:

auto_canny.py — Python

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

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

# load the image, convert it to grayscale, and blur it slightly
image = cv2.imread(args["image"])
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(gray, (3, 3), 0)

# apply Canny edge detection using a wide threshold, tight
# threshold, and automatically determined threshold
wide = cv2.Canny(blurred, 10, 200)
tight = cv2.Canny(blurred, 225, 250)
auto = imutils.auto_canny(blurred)

# show the images
cv2.imshow("Original", image)
cv2.imshow("Wide", wide)
cv2.imshow("Tight", tight)
cv2.imshow("Auto", auto)
cv2.waitKey(0)
```

We start off by importing our packages on **Lines 1-4** and then parsing our command line arguments on **Lines 7-8**. We'll be accepting a single command line argument here, `--image`, which is the path to where our image resides on disk.

Just like in the previous edge detection example, we'll need to do a little pre-processing. After loading our image from disk, we convert it to grayscale, and then apply a little bit of Gaussian blurring to help reduce high frequency noise (**Lines 12-14**).

Lines 18-20 then apply Canny edge detection using three methods:

1. A *wide* threshold.
2. A *tight* threshold.
3. A threshold determined *automatically* using our `auto_canny` function.

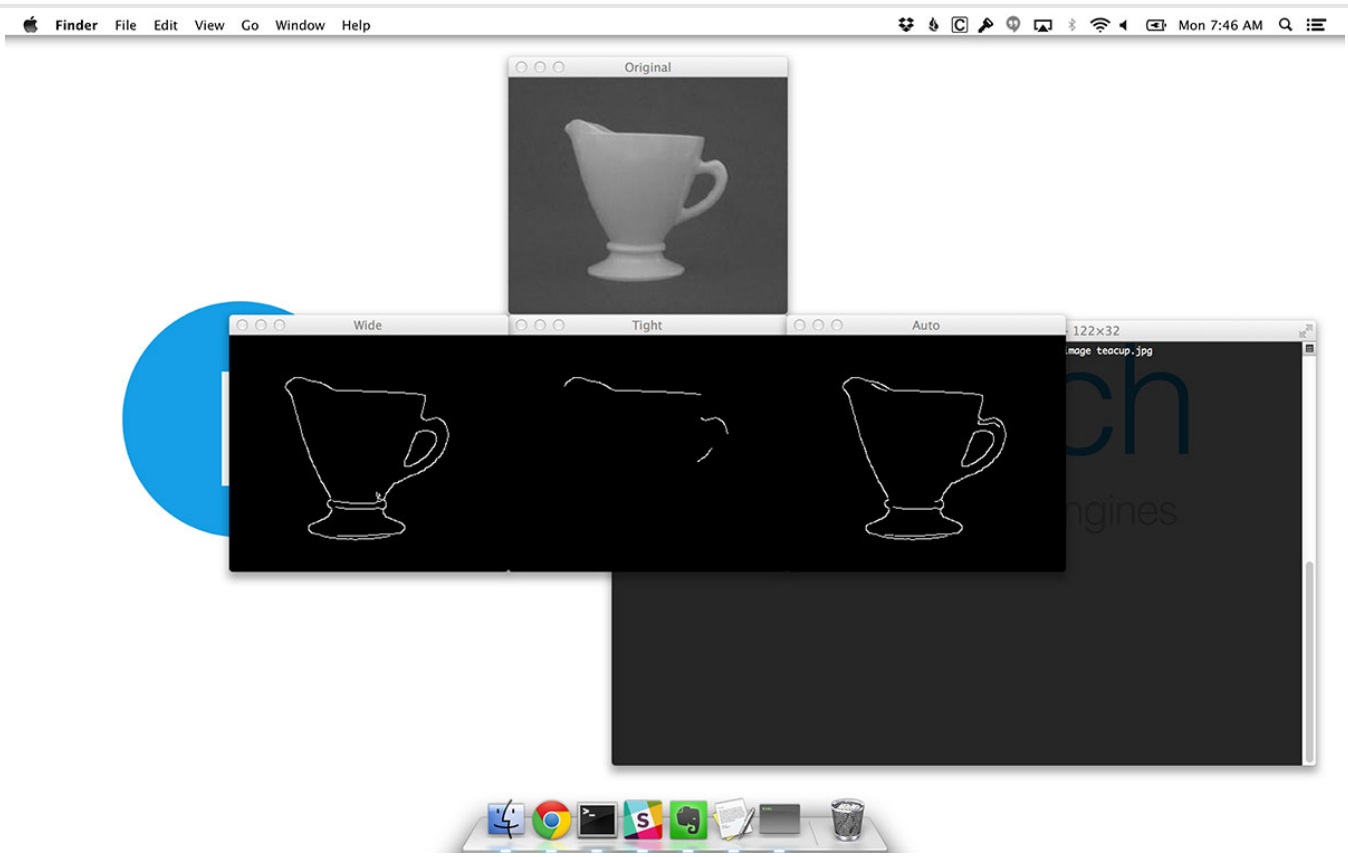
Finally, we can see our resulting edge maps on **Lines 23-27**.

To see our automatic Canny edge detector in action, open up a terminal and execute the following command:

auto_canny.py — Shell

```
$ python auto_canny.py --image teacup.jpg
```

Followed by the output:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/auto_canny_example.jpg)

FIGURE 10: APPLYING AUTOMATIC CANNY EDGE DETECTION TO A PICTURE OF A CUP. *(LEFT)* WIDE CANNY EDGE THRESHOLD. *(CENTER)* TIGHT CANNY EDGE THRESHOLD. *(RIGHT)* AUTOMATIC CANNY EDGE THRESHOLD.

The results here are fairly dramatic. While both the wide *(left)* and the automatic *(right)* Canny edge detection methods perform similarly, the tight threshold *(center)* misses out on almost all of the structural edges of the cup.

Again, this is just a simple trick to (reliably) automatically detect edges in images using the Canny edge detector, ***without providing thresholds to the function***. We were able to detect edges in our image with *literally zero effort!*

This `auto_canny` method simply takes the median of the image and then constructs upper and lower thresholds based on a percentage of this median. In practice, `sigma=0.33` tends to obtain good results; however, you'll want to test this parameter yourself before unleashing it on your entire dataset.

In general, you'll find that the automatic, zero-parameter version of the Canny edge detection is able to obtain fairly decent results with little to no effort on your part.

Summary

In this lesson we learned how to use image gradients, one of the most fundamental building blocks of computer vision and image processing, to create an edge detector.

Specifically, we focused on the Canny edge detector, the most well known and most used edge detector in the computer vision community.

From there, we examined the steps of the Canny edge detector including *smoothing*, *the computation of image gradients*, *non-maxima suppression*, and *hysteresis thresholding*.

We then took our knowledge of the Canny edge detector and used it to apply OpenCV's `cv2.Canny` function to detect edges in images.

However, one of the biggest drawbacks of the Canny edge detector is tuning the upper and lower thresholds for the hysteresis step. If our threshold was too wide, we would get too many edges. And if our threshold was too tight, we would not detect many edges at all!

To aide us in this parameter tuning problem we defined the `auto_canny` function, part of the imutils (https://github.com/jrosebr1/imutils) package which can be used to *automatically* tune our parameters to the Canny edge detector, leaving us with good results in most situations with practically zero effort.

Downloads:

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

	Quizzes	Status
1	Edge Detection Quiz (https://gurus.pyimagesearch.com/quizzes/edge-detection-quiz/)	

← Previous Topic (https://gurus.pyimagesearch.com/topic/gradients/)

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