# 1.10.1: Gradients

**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/)



The *image gradient* is one of the fundamental building blocks in computer vision image processing.

We use gradients for **detecting edges in images**, which allows us to find **contours** and **outlines** of objects in images. We use them as inputs for quantifying images through feature extraction — in fact, highly successful and well-known image descriptors such as **Histogram of Oriented Gradients** and **SIFT** are built upon image gradient representations. Gradient images are even used to construct **saliency maps**, which highlight the subjects of an image.

Clearly, we use gradients *all the time* in computer vision and image processing. I would go as far as to say they are one of *the most important* building blocks you'll learn about inside this course. While they are not often discussed in detail since other more powerful and interesting methods build on top of them, we are going to take the time and discuss them in detail.

## Objectives:

This lesson has 5 primary objectives. By the time we are complete with this lesson you should be able to:

1. Define what an *image gradient* is.

2. Compute the *gradient orientation* of an input image.
3. Define both **gradient magnitude** and **gradient orientation**.
4. Learn how to compute **gradient magnitude** and **gradient orientation**.
5. Approximate the image gradient using *Sobel* and *Scharr* kernels.
6. Learn how to use the `cv2.Sobel` function to compute image gradient representations in OpenCV.

# Image Gradients

As I mentioned in the introduction, image gradients are used as the basic building blocks in many computer vision and image processing applications.

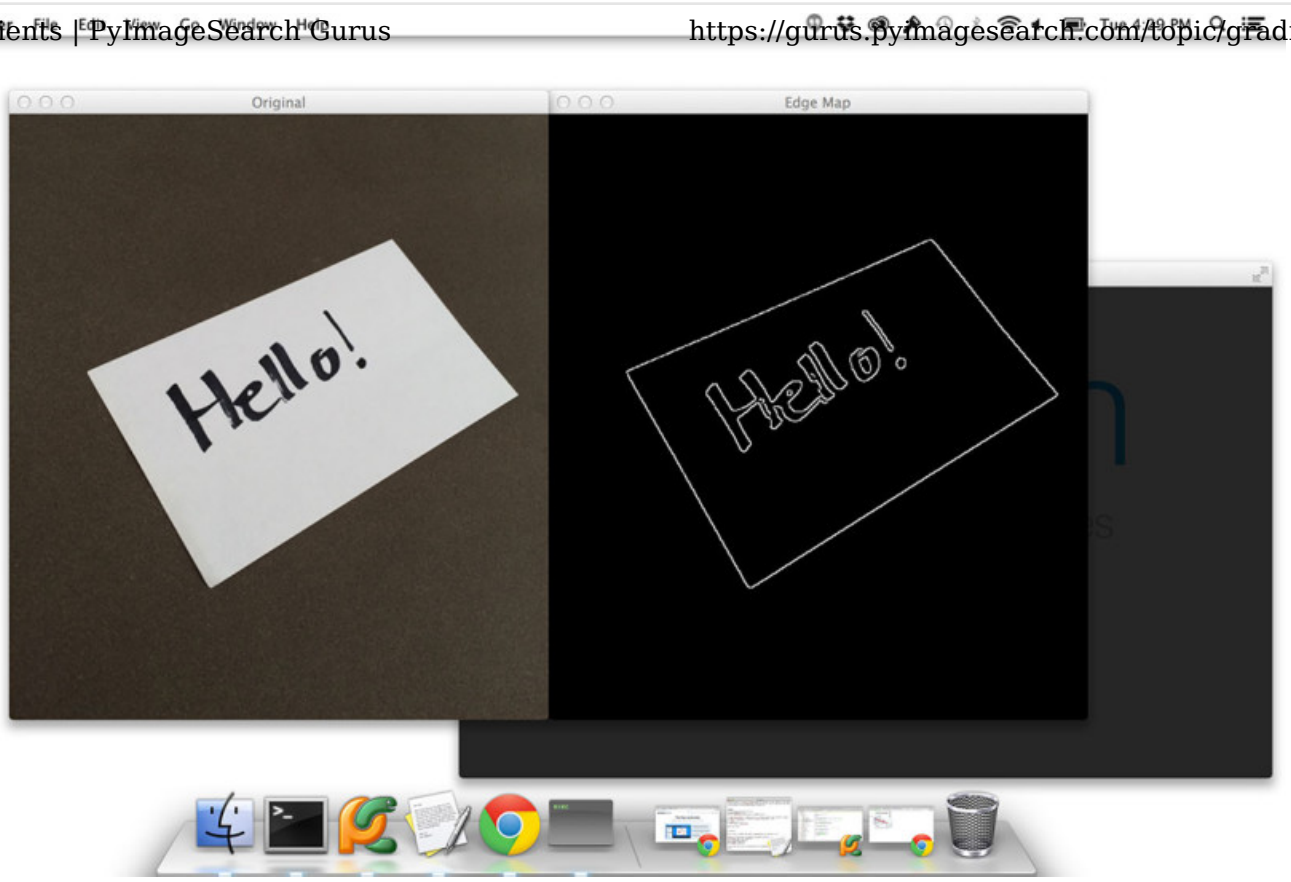
However, the main application of image gradients lies within *edge detection*.

As the name suggests, *edge detection* is the process of finding edges in an image, which reveals *structural information* regarding the objects in an image. Edges could therefore correspond to:

- Boundaries of an object in an image.
- Boundaries of shadowing or lighting conditions in an image.
- Boundaries of "parts" within an object.



Below is an image of edges being detected in an image:




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

**FIGURE 1:** AN EXAMPLE OF EXTRACTING EDGES FROM IMAGES. COMPUTING IMAGE GRADIENTS IS A PRE-PROCESSING STEP TO DETECTING EDGES IN IMAGES.

Feedback

On the *left* we have our original input image. On the *right* we have our image with detected edges — commonly called an **edge map**. The image on the right clearly reveals the structure and outline of the objects in an image. Notice how the outline of the notecard, along with the words written on the notecard, are clearly revealed. Using this outline we could then apply **contours** (**Module 1.11** (https://gurus.pyimagesearch.com/lessons/contours/)) to extract the actual objects from the region or quantify the shapes so we can identify them later. Just as image gradients are building blocks for methods like edge detection, edge detection is also a building block for developing a complete computer vision application.

So how do we go about finding these edges in an image?

The first step is to compute the *gradient* of the image.

Formally, an image gradient is defined as **a *directional* change in image intensity**.

Or put more simply, at each pixel of the input (grayscale) image, a gradient measures the change in pixel intensity in a given direction. By estimating the direction or *orientation* along with the *magnitude* (i.e. how

strong the change in direction is), we are able to detect regions of an image that look like edges.

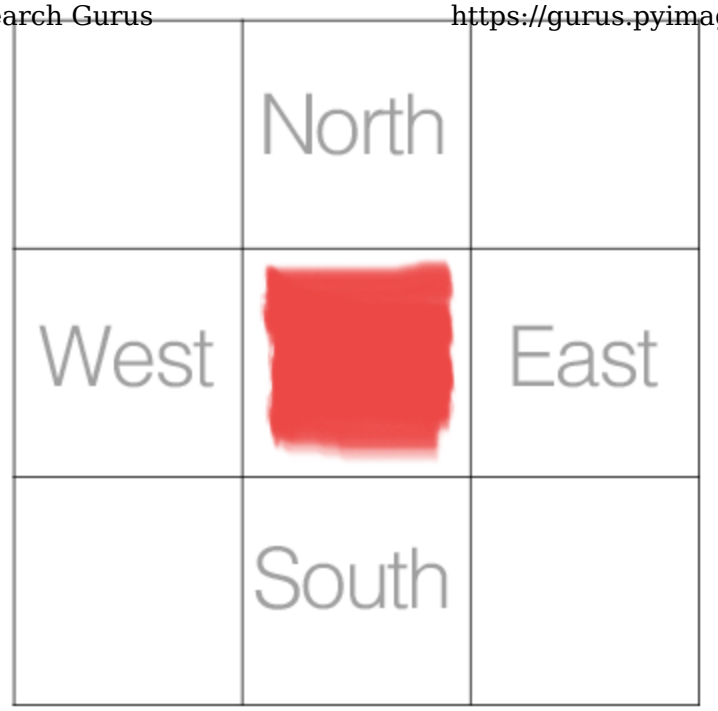
In practice, image gradients are estimated using kernels (**Module 1.5** (https://gurus.pyimagesearch.com/lessons/kernels/)) just like we did using smoothing and blurring — but this time we are trying to find the *structural components* of the image. Our goal here is to find the change in direction to the central pixel marked in *red* in both the *x* and *y* direction:


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

**FIGURE 2:** WHEN COMPUTING IMAGE GRADIENTS, WE NEED TO FIND THE DIFFERENCE IN IMAGE PIXEL INTENSITIES ALONG BOTH THE *X* AND *Y* DIRECTION WITHIN THE GIVEN KERNEL SIZE.

However, before we dive into kernels for gradient estimation, let's actually go through the process of computing the gradient manually. The first step is to simply find and mark the *north*, *south*, *east* and *west* pixels surrounding the center pixel:

Feedback




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

**FIGURE 3:** LABELING THE NORTH, SOUTH, EAST, AND WEST PIXEL SURROUNDING THE CENTER RED PIXEL.

In the image above we examine the $3 \times 3$ neighborhood surrounding the central pixel. Our *x* values run from left to right, and our *y* values from top to bottom. In order to compute any changes in direction we'll need the *north*, *south*, *east*, and *west* pixels, which are marked on **Figure 3**.

If we denote our input image as $I$, then we define the north, south, east, and west pixels using the following notation:

- **North:** $I(x, y - 1)$
- **South:** $I(x, y + 1)$
- **East:** $I(x + 1, y)$
- **West:** $I(x - 1, y)$

Again, these four values are ***critical*** in computing the changes in image intensity in both the *x* and *y* direction.

To demonstrate this, let's compute the **vertical change** or the ***y*-change** by taking the difference between the south and north pixels:

$$G_y = I(x, y + 1) - I(x, y - 1)$$

Similarly, we can compute the **horizontal change** or the ***x*-change** by taking the difference between the east and west pixels:

$$G_x = I(x+1, y) - I(x-1, y)$$

Awesome — so now we have $G_x$ and $G_y$, which represent the change in image intensity for the central pixel in both the *x* and *y* direction.

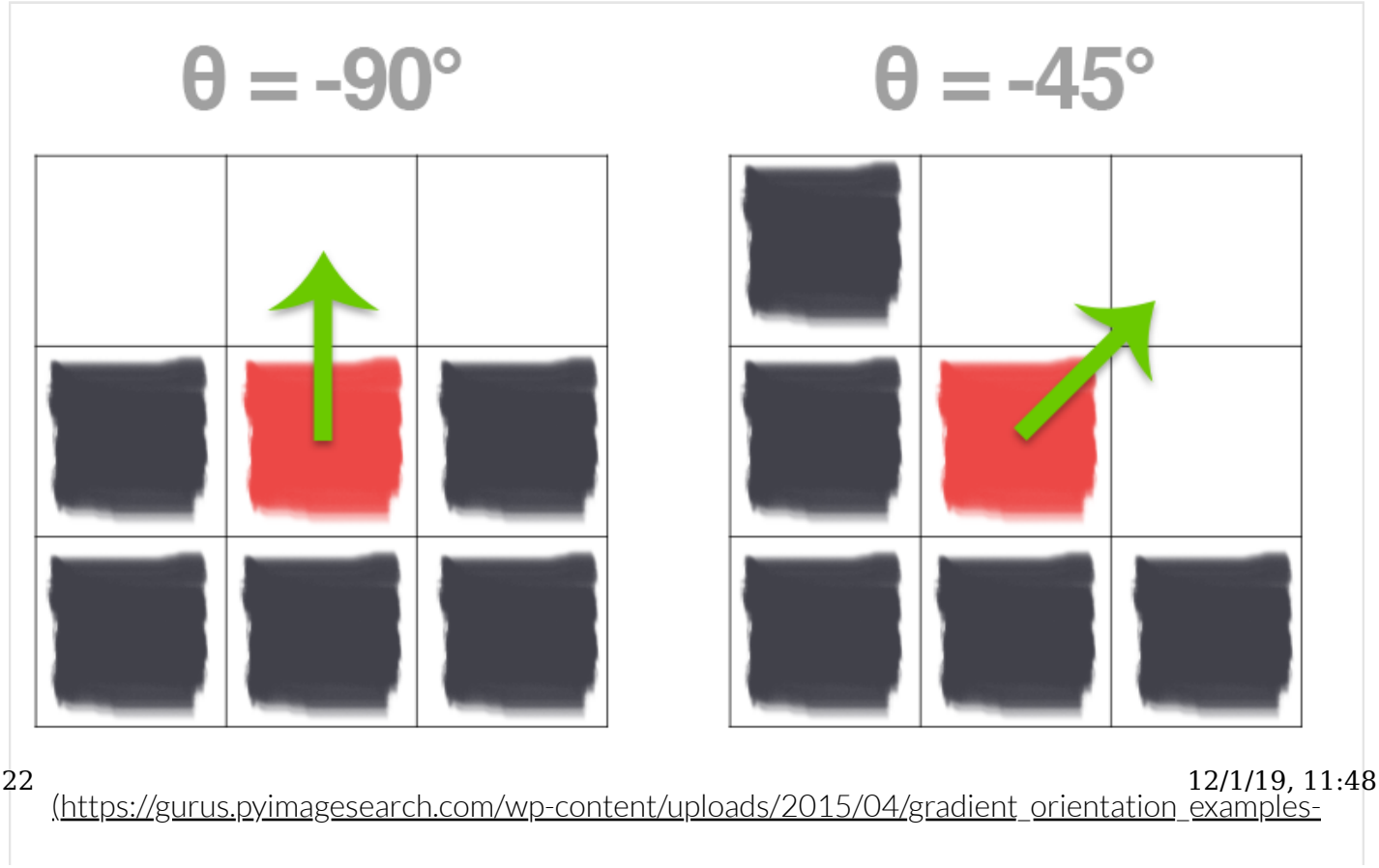
So now the big question becomes: ***what do we do with these values?***

To answer that, we'll need to define two new terms — the **gradient magnitude** and the **gradient orientation**.

The **gradient magnitude** is used to **measure how strong the change in image intensity is**. The gradient magnitude is a real-valued number that quantifies the "strength" of the change in intensity.

While the **gradient orientation** is used to determine **in which direction the change in intensity is pointing.** As the name suggests, the gradient orientation will give us an angle or $\theta$ that we can use to quantify the direction of the change.

For example, take a look at the following visualization of gradient orientation:




(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/gradient_orientation_examples-

Feedback

**FIGURE 4:** COMPUTING CHANGE IN ORIENTATION (IN DEGREES) FROM THE CENTRAL PIXEL. LEFT: THE CHANGE IN ORIENTATION FROM THE CENTRAL PIXEL (-90 DEGREES). RIGHT: NOTICE HOW THE CHANGE THIS TIME IS MORE ANGLED (-45 DEGREES). WE'LL REVIEW HOW TO ACTUALLY PERFORM THIS CALCULATION LATER IN THIS LESSON.

On the *left* we have a $3 \times 3$ region of an image where the top half of the image is *white* and the bottom half of the image is *black*. The gradient orientation is thus equal to $\theta = -90^{\circ}$

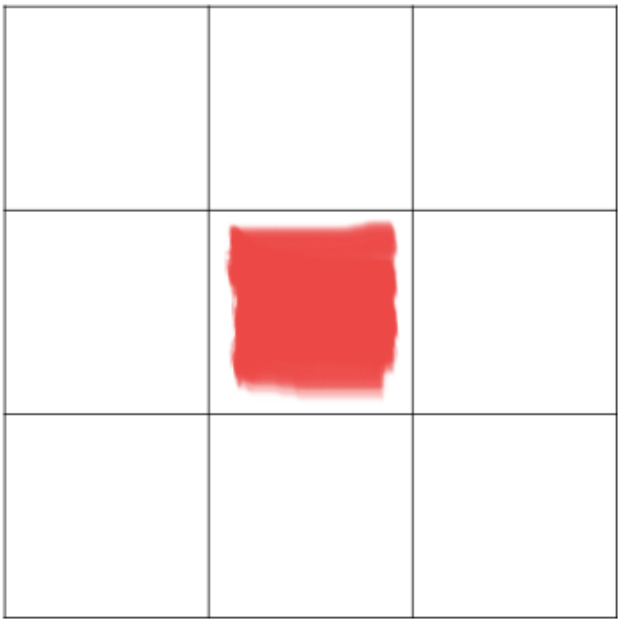
And on the *right* we have another $3 \times 3$ neighborhood of an image, where the upper triangular region is *white* and the lower triangular region is *black*. Here we can see the change in direction is equal to $\theta = -45^{\circ}$.

So I'll be honest — when I was first introduced to computer vision and image gradients, **Figure 4** confused the living hell out of me. I mean, *how in the world* did we arrive at these calculations of $\theta$?! Intuitively, the changes in direction make sense since we can actually see and visualize the result.

**But how do we *actually* go about computing the gradient orientation and magnitude?**

I'm glad you asked. Let's see if we can demystify the gradient orientation and magnitude calculation.

Let's go ahead and start off with an image of our trusty $3 \times 3$ neighborhood of an image:



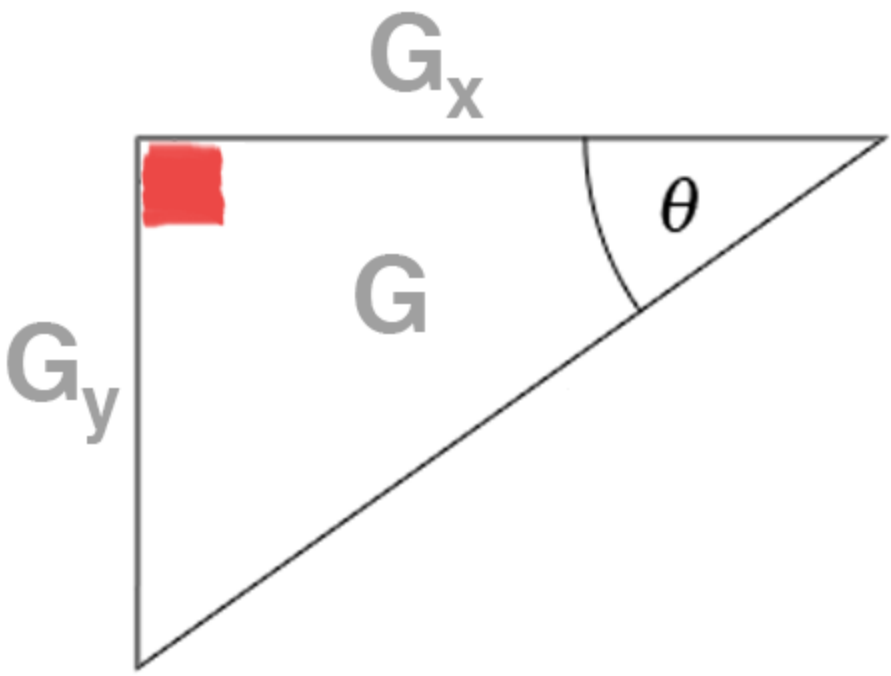
(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/gradient_center_3x3.png)

**FIGURE 5:** OUR STANDARD PIXEL NEIGHBORHOOD REPRESENTATION — THE RED PIXEL IS THE CENTRAL PIXEL, WITH ITS 8-NEIGHBORHOOD SURROUNDING IT.

Feedback

Here we can see that the central pixel is marked in *red*. The next step in determining the gradient orientation and magnitude is actually to compute the changes in gradient in both the *x* and *y* direction. Luckily, we already know how to do this — they are simply the $G_x$ and $G_y$ values that we computed earlier!

Using both $G_x$ and $G_y$, we can apply some basic trigonometry to compute the gradient magnitude $G$, and orientation $\theta$:




(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/gradient_triangle-1.png)

**FIGURE 6:** THE BASIC TRIGONOMETRY OF COMPUTING THE GRADIENT MAGNITUDE AND ORIENTATION.

Seeing this example is what really solidified my understanding of gradient orientation and magnitude. Inspecting this triangle you can see that the gradient magnitude $G$ is the hypotenuse of the triangle. Therefore, all we need to do is apply the Pythagorean theorem and we'll end up with the gradient magnitude:

$$G = \sqrt{G_x^2 + G_y^2}$$

The gradient orientation can then be given as the ratio of $G_y$ to $G_x$. Technically we would use the $tan^{-1}$ to compute the gradient orientation, but this could lead to undefined values — since we are computer scientists, we'll use the $arctan2$ function to account for the different quadrants instead (if you're unfamiliar with the arc-tangent, you can read more about it here (http://en.wikipedia.org/wiki/Atan2)):

The $arctan2$ function gives us the orientation in *radians*, which we then convert to *degrees* by multiplying by the ratio of $180/\pi$.

Let's go ahead and manually compute $G$ and $\theta$ so we can see how the process is done:


(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/gradient_90_degrees.png)

**FIGURE 7:** *(LEFT)* THE PIXEL VALUES OF THE 8-NEIGHBORHOOD. *(RIGHT)* A VISUALIZATION OF THE 90-DEGREE GRADIENT ORIENTATION.

Feedback

In the above image we have an image where the upper-third is *white* and the bottom two-thirds is *black*. Using the equations for $G_x$ and $G_y$, we arrive at:

$$G_x = 0 - 0 = 0$$

and

$$G_y = 0 - 255 = -255$$

Plugging these values into our gradient magnitude equation we get:

$$G = \sqrt{0^2 + (-255)^2} = 255$$

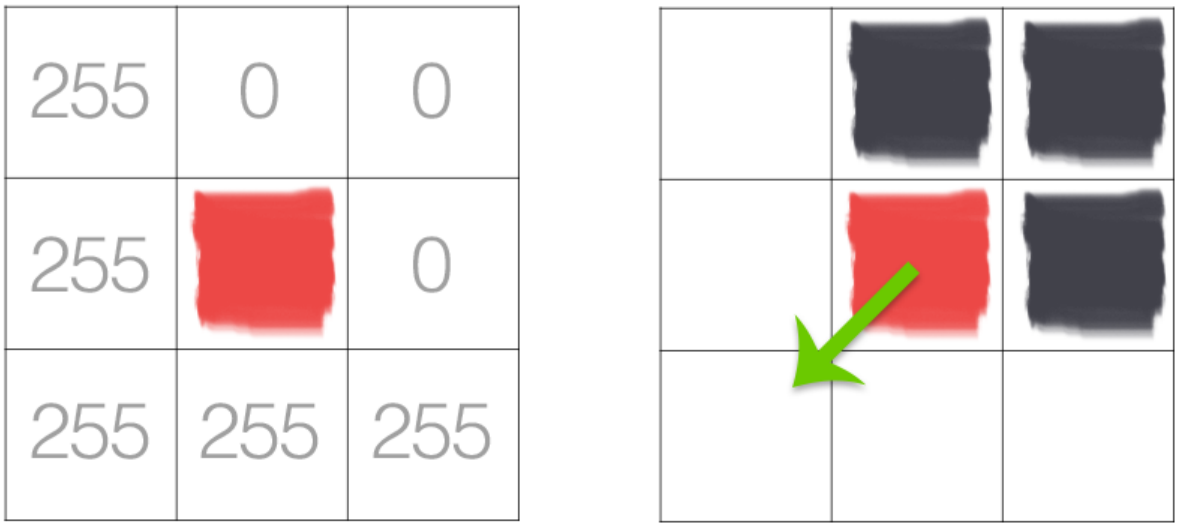
As for our gradient orientation:

$$\theta = arctan2(-255, 0) \times (\tfrac{180}{\pi}) = -90^\circ$$

Sure enough, the gradient of the central pixel is pointing *up* as verified by the $\theta = -90^\circ$.

See, computing the gradient magnitude and orientation isn't too bad!

Let's do another example for fun:




(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/gradient_-135_degrees.png)

**FIGURE 8:** *(LEFT)* OUR RAW PIXEL INTENSITY VALUES. *(RIGHT)* THE GRADIENT ORIENTATION OF -135 DEGREES.

In this particular image we can see that the lower-triangular region of the neighborhood is *white* while the upper-triangular neighborhood is *black*. Computing both $G_x$ and $G_y$ we arrive at:

$G_x = 0 - 255 = -255$ and $G_y = 255 - 0 = 255$

Which leaves us with a gradient magnitude of:

$$G = \sqrt{(-255)^2 + 255^2} = 360.62$$

And a gradient orientation of:

$$\theta = arctan2(255, -255) \times (\frac{180}{\pi}) = 135^\circ$$

Sure enough, our gradient is pointing *down* and to the *left* at an angle of $135^\circ$.

Of course, we have only computed our gradient orientation and magnitude for two unique pixel values: *0* and *255*. Normally you would be computing the orientation and magnitude on a *grayscale* image where the valid range of values would be *[0, 255]*.

# Sobel and Scharr kernels

Now that we have learned how to compute gradients manually, let's look at how we can *approximate* them using kernels, which will give us a tremendous boost in speed. Just like we used **kernels** (https://gurus.pyimagesearch.com/lessons/kernels/) to **smooth and blur**

Feedback

kernels to compute our gradients.

We'll start off with the *Sobel method*, which actually uses two kernels: one for detecting horizontal changes in direction and the other for detecting vertical changes in direction:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \text{and } G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Given an input image neighborhood below, let's compute the Sobel approximation to the gradient:

$$I_{i,j} = \begin{bmatrix} 93 & 139 & 101 \\ 26 & 252 & 196 \\ 135 & 230 & 18 \end{bmatrix}$$

Therefore,

$$G_x = \sum \begin{bmatrix} -1 \times 93 & 0 \times 139 & 1 \times 101 \\ -2 \times 26 & 0 \times 252 & 2 \times 196 \\ -1 \times 135 & 0 \times 230 & 1 \times 18 \end{bmatrix} = \sum \begin{bmatrix} -93 & 0 & 101 \\ -52 & 0 & 392 \\ -135 & 0 & 18 \end{bmatrix} = 231$$

And,

$$G_y = \sum \begin{bmatrix} -1 \times 93 & -2 \times 139 & -1 \times 101 \\ 0 \times 26 & 0 \times 252 & 0 \times 196 \\ 1 \times 135 & 2 \times 230 & 1 \times 18 \end{bmatrix} = \sum \begin{bmatrix} -93 & -278 & -101 \\ 0 & 0 & 0 \\ 135 & 460 & 18 \end{bmatrix} = 141$$

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/04/gradients_gy.gif)

Feedback

Given these values of $G_x$ and $G_y$, it would then be trivial to compute the gradient magnitude $G$ and orientation $\theta$:

$G = \sqrt{231^2 + 141^2} = 270.63$ and $\theta = arctan2(141, 231) \times \frac{180}{\pi} = 31.4^\circ$

We could also use the *Scharr* kernel instead of the *Sobel* kernel which may give us better approximations to the gradient:

$$G_x = \begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix} \text{and } G_y = \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

The exact reasons as to why the Scharr kernel could lead to better approximations are heavily rooted in

mathematical details are well outside our discussion of image gradients. If you're interested in reading more about the Scharr versus Sobel kernels and constructing an optimal image gradient approximation (and can read German), I would suggest taking a look at Scharr's dissertation (http://archiv.ub.uni-heidelberg.de/volltextserver/962/) on the topic.

Overall, gradient magnitude and orientation make for excellent features and image descriptors when quantifying and abstractly representing an image. But for edge detection, the gradient representation is extremely sensitive to local noise. We'll need to add in a few more steps to create an actual robust edge detector — we'll be covering these steps in detail in the next lesson where we review the Canny edge detector.

# Sobel kernels in OpenCV

Up until this point we have been discussing a lot of the theory and mathematical details surrounding image kernels. But how do we actually ***apply*** what we have learned using Python + OpenCV?

I'm so glad you asked.

Open up a new file, name it `sobel.py` , and let's get coding:

Feedback

sobel.py Python

```
# 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, convert it to grayscale, and display the original
# image
image = cv2.imread(args["image"])
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow("Original", image)

# compute gradients along the X and Y axis, respectively
gX = cv2.Sobel(gray, ddepth=cv2.CV_64F, dx=1, dy=0)
gY = cv2.Sobel(gray, ddepth=cv2.CV_64F, dx=0, dy=1)

# the `gX` and `gY` images are now of the floating point data type,
# so we need to take care to convert them back to an unsigned 8-bit
# integer representation so other OpenCV functions can utilize them
gX = cv2.convertScaleAbs(gX)
gY = cv2.convertScaleAbs(gY)

# combine the sobel X and Y representations into a single image
sobelCombined = cv2.addWeighted(gX, 0.5, gY, 0.5, 0)

# show our output images
cv2.imshow("Sobel X", gX)
cv2.imshow("Sobel Y", gY)
cv2.imshow("Sobel Combined", sobelCombined)
cv2.waitKey(0)
```

Feedback

**Lines 1-8** simply handle importing our necessary packages and setting up our argument parser. We'll need only a single switch here, `--image`, which is the image that we want to load from disk.

**Lines 12-14** then load our `image` from disk, convert it to grayscale (since we compute gradient representations on the grayscale version of the image), and display it to our screen.

Computing both the $G_x$ and $G_y$ values is handled on **Lines 17 and 18** by making a call to `cv2.Sobel`. Specifying a value of `dx=1` and `dy=0` indicates that we want to compute the gradient across the *x* direction. And supplying a value of `dx=0` and `dy=1` indicates that we want to compute the gradient across the *y* direction.

***Note:*** While it's not present in the code sample above, computing the Scharr kernel can be done in the exact same manner, only using the `cv2.Scharr` function or supplying an optional parameter value of `ksize=-1` for the `cv2.Sobel` function.

However, both `gX` and `gY` are now of the floating point data type, so if we plan to visualize them on our screen we need to convert them back to 8-bit unsigned integers. **Lines 23 and 24** take the *absolute value* of the gradient images and then squish the values back into the range *[0, 255]*.

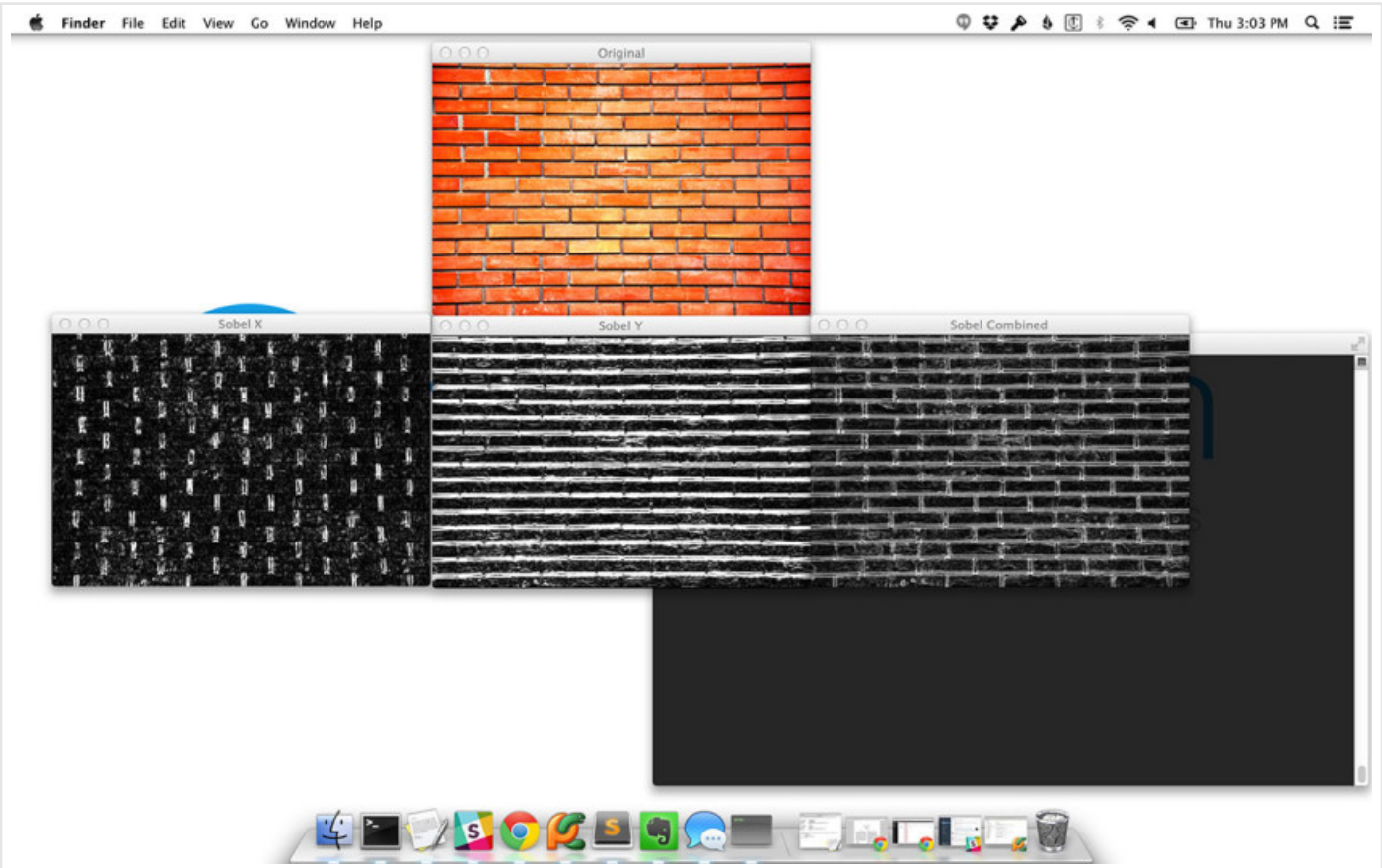
Finally, we combine `gX` and `gY` into a single image using the `cv2.addWeighted` function, weighting each gradient representation equally.

**Lines 30-33** then show our output images to our screen.

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

```
sobel.py                                                    Shell
1 $ python sobel.py --image bricks.png
```

You'll then see the following output image:




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

**FIGURE 9:** *(TOP)* OUR ORIGINAL INPUT IMAGE OF A BRICK WALL. *(BOTTOM LEFT)* COMPUTING THE IMAGE GRADIENT ALONG THE X DIRECTION. *(BOTTOM CENTER)* THE IMAGE GRADIENTS ALONG THE Y DIRECTION. *(BOTTOM RIGHT)* COMBINING THE X AND Y GRADIENT REPRESENTATIONS TOGETHER.

Feedback

At the *top* we have our original image, which is an image of a brick wall. The *bottom-left* then displays the

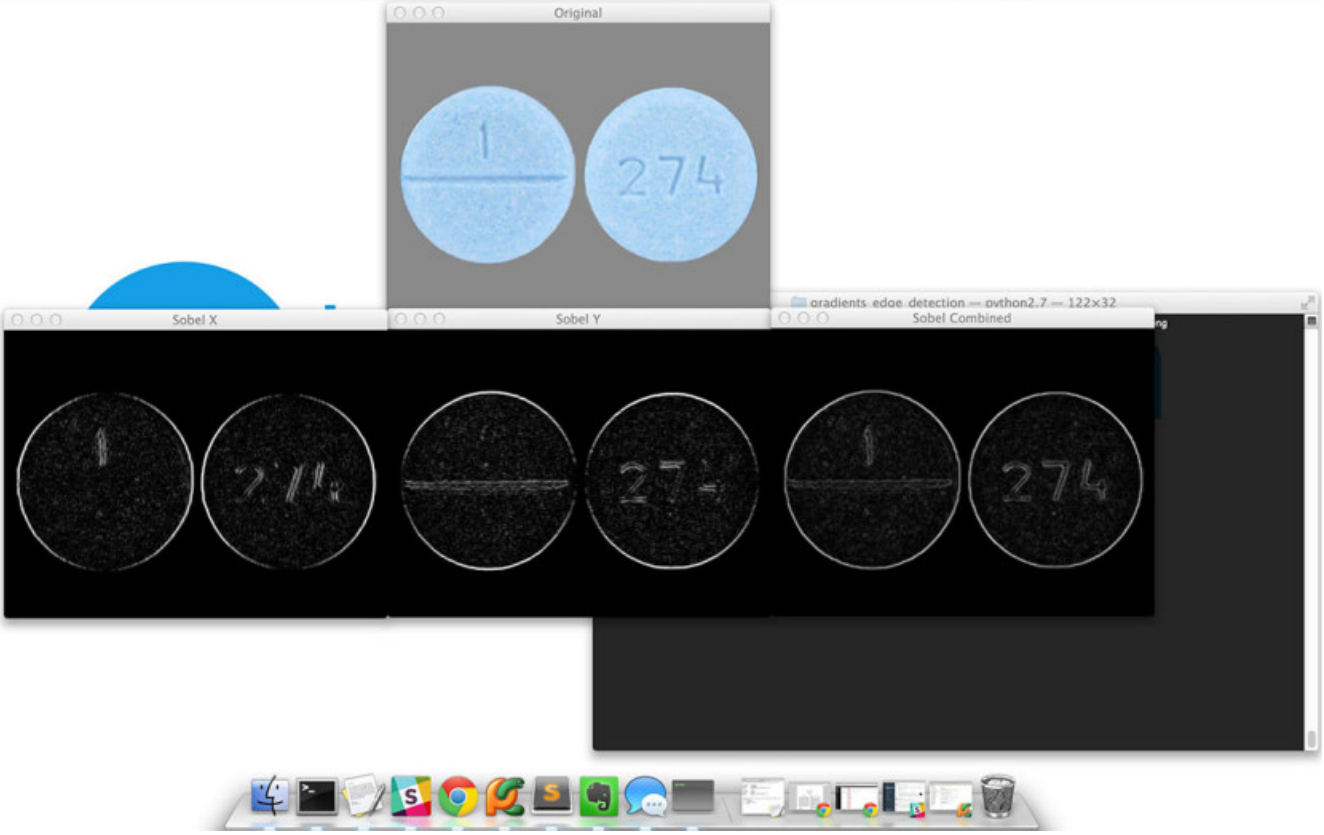
Sobel gradient image along the x direction. Notice how computing the Sobel gradient along the *x* direction reveals the *vertical* mortar regions of the bricks. Similarly, the *bottom-center* shows that Sobel gradient computed along the *y* direction — now we can see the *horizontal* mortar regions of the bricks.

Finally, we can add `gX` and `gY` together and receive our final output image on the *bottom-right*.

Let's look at another example:

```
sobel.py                                                    Shell
1 $ python sobel.py --image clonazepam_1mg.png
```

And here's the output:




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

**FIGURE 10:** *(TOP)* OUR ORIGINAL INPUT IMAGE OF THE FRONTSIDE AND BACKSIDE OF A PILL. *(BOTTOM LEFT)* COMPUTING THE IMAGE GRADIENT ALONG THE X DIRECTION. *(BOTTOM CENTER)* THE IMAGE GRADIENTS ALONG THE Y DIRECTION. *(BOTTOM RIGHT)* COMBINING THE X AND Y GRADIENT REPRESENTATIONS TOGETHER.

Again, just like in the previous example we have the original image at the *top* of our screen — this input image is a photo of a prescription pill.

The *bottom-left* then displays the image gradient along the *x* direction. Notice how the *vertical* regions of the pills are now exposed, such as the left and right boundaries, along with the number *1*.

In the bottom-left we have the image gradient along the y direction. Take special notice of the horizontal imprint that runs the entire length of the pill in the center. In the *bottom-left* image where we computed the gradient across the x direction, this region was not visible. But by computing the gradient along the *y* direction, we were able to reveal it — this is why we compute the image gradient in **both** the *x* and *y* directions.

# Gradient orientation and magnitude in OpenCV

Alright, so now that we have a taste of the Sobel kernel, let's try to compute the gradient orientation and magnitude like we did earlier in this lesson — only instead of doing this computation manually, we are now going to do it with code.

Open up a new file, name it `mag_orientation.py` , and let's see how it's done:

mag_orientation.py Python

```
# import the necessary packages
import numpy as np
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")
ap.add_argument("-l", "--lower-angle", type=float, default=175.0,
    help="Lower orientation angle")
ap.add_argument("-u", "--upper-angle", type=float, default=180.0,
    help="Upper orientation angle")
args = vars(ap.parse_args())
```

I normally don't spend much time explaining command line arguments as they are normally quite simple and intuitive. But let's take a second and define these as I think they are *paramount* in understanding this particular example.

Our script will require three command line arguments. The first is the `--image` , which is the path to where our image resides on disk. The second is the `--lower-angle` , or the *smallest* gradient orientation angle we are interested in detecting. Similarly, we define the final argument as `--upper-angle` , which is the *largest* gradient orientation angle that we want to detect. We default these min and max angles to $175^{\circ}$ and $180^{\circ}$ respectively, but you can change them to whatever you like when executing the script.

The end goal of this program will be to (1) compute the gradient orientation and magnitude, and then (2) only display the pixels in the image that fall within the range $min_{\theta} <= \theta <= max_{\theta}$.

Below follows the remainder of the source code listing:

```
mag_orientation.py                                                    Python
15 # load the image, convert it to grayscale, and display the original
16 # image
17 image = cv2.imread(args["image"])
18 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
19 cv2.imshow("Original", image)
20
21 # compute gradients along the X and Y axis, respectively
22 gX = cv2.Sobel(gray, cv2.CV_64F, 1, 0)
23 gY = cv2.Sobel(gray, cv2.CV_64F, 0, 1)
24
25 # compute the gradient magnitude and orientation respectively
26 mag = np.sqrt((gX ** 2) + (gY ** 2))
27 orientation = np.arctan2(gY, gX) * (180 / np.pi) % 180
28
29 # find all pixels that are within the upper and low angle boundaries
30 idxs = np.where(orientation >= args["lower_angle"], orientation, -1)
31 idxs = np.where(orientation <= args["upper_angle"], idxs, -1)
32 mask = np.zeros(gray.shape, dtype="uint8")
33 mask[idxs > -1] = 255
34
35 # show the images
36 cv2.imshow("Mask", mask)
37 cv2.waitKey(0)
```

As we did in the **Sobel kernels in OpenCV** section above, we load our image of disk and convert it to grayscale on **Lines 17 and 18**. Then using this grayscale representation of the image, we compute the gradients in both the *x* and *y* direction using the Sobel filter on **Lines 22 and 23**.



However, unlike the previous section, we are not going to display the gradient images to our screen, thus we do not have to convert them back into the range *[0, 255]* or use the `cv2.addWeighted` function to combine them together.

Instead, we continue with our gradient magnitude and orientation calculations on **Lines 26 and 27**. Notice how these two lines match our equations above ***exactly.***

The *gradient magnitude* is simply the square-root of the squared gradients in both the *x* and *y* direction added together (**Line 26**).

And the *gradient orientation* is the arc-tangent of the gradients in both the *x* and *y* direction (**Line 27**).

That's really all there is to it! Using these two lines of code we end up with both the gradient magnitude and orientation. While we won't be using the magnitude for the rest of this particular example, we will be discussing it and using it once we get to the next lesson on the Canny edge detector, and later when we

explore image descriptors such as Histogram of Oriented Gradients and SIFT.

Next up, **Lines 30-34** handle selecting the pixels that fall within our lower and upper angles supplied at the command line during runtime. Using the `np.where` function, we are able to select the *indexes* (or simply, the *x* and *y* coordinates) of the NumPy array (i.e our image) that meet our orientation requirements.

**Line 30** handles selecting image coordinates that are *greater than* the lower angle minimum. The first argument to the `np.where` function is the condition that we want to test: again, we are looking for indexes that are greater than the minimum supplied angle. The second argument to `np.where` is the array that we want to check — this is obviously our `orientation` array. And the final argument that we supply is the value if the check *does not* pass. In the case that the orientation is *less than* the minimum angle requirement, we'll set that particular value to *-1*.

In a similar fashion, **Line 31** finds image coordinate that are *less than* the upper angle maximum. Again, we supply our orientations test as the first argument. The second argument is the `idxs` list returned by **Line 30** since we are looking for orientations that pass ***both*** the upper and lower orientation test. Finally, if the orientation is *greater than* the maximum angle requirement, we'll set that particular value to *-1*.

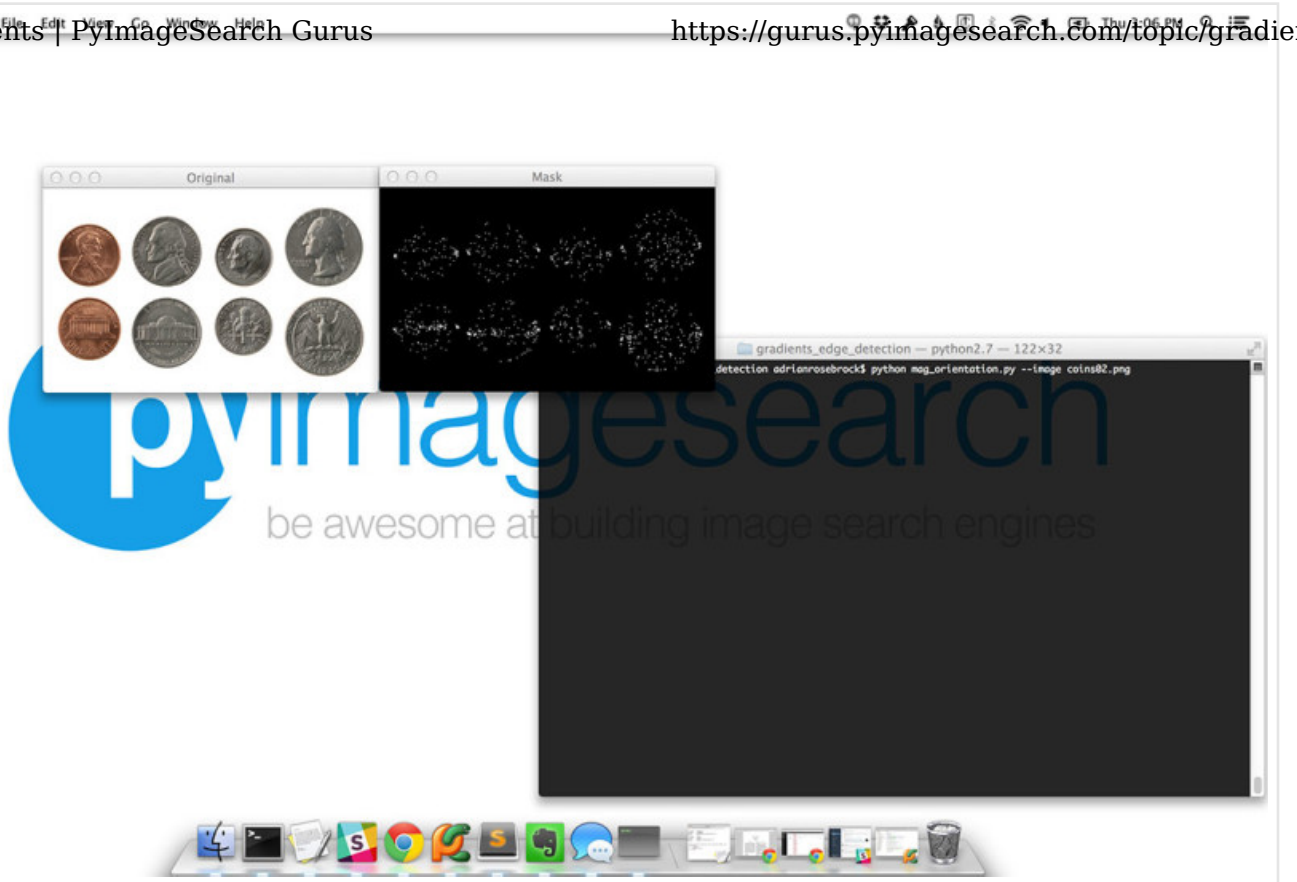


The `idxs` now contains the coordinates of all orientations that are greater than the minimum angle and less than the maximum angle. Using this list, we construct a `mask` on **Lines 32 and 33** — all coordinates that have a corresponding `idxs` value of *> -1* are set to *255* (i.e. foreground). Otherwise, they are left as *0* (i.e. background).

**Lines 36 and 37** then show our output image.

Let's see our code in action:

```
mag_orientation.py                                                    Shell
1 $ python mag_orientation.py --image coins02.png
```




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

**FIGURE 11:** SELECTING THE GRADIENT ORIENTATION PIXEL VALUES WITHIN THE SPECIFIED RANGE.

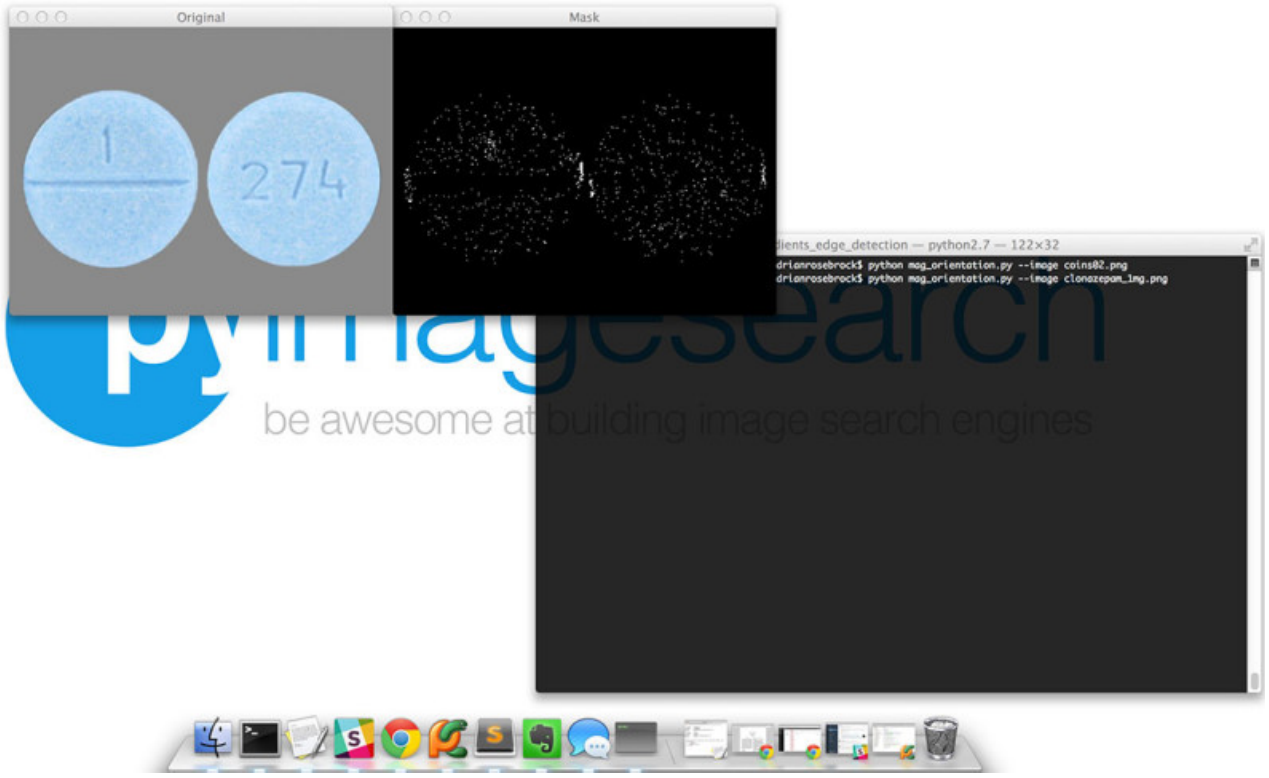
Feedback

On the *left* we have our original input image of coins. And on the *right* we have a mask that displays ***only*** pixels where the gradient orientation falls into the range $175 <= \theta <= 180$. The most dense population of pixels that fall into this range are found along the right and left horizontal boundaries of each coin.

Let's do another example, this time using pills:

may_orientation.py Shell

```
1 $ python mag_orientation.py --image clonazepam_1mg.png
```




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

**FIGURE 12:** AGAIN, ANOTHER EXAMPLE OF SELECTING PIXEL VALUES THAT HAVE CORRESPONDING GRADIENT ORIENTATIONS WITHIN THE SUPPLIED RANGE OF ANGLES.

Feedback

Again, like in **Figure 11** above, our original image is on the *left* and our gradient mask is displayed on the *right*. Similarly, the most dense population of pixels that fall into the range $175 <= \theta <= 180$ are predominately along the left and right horizontal boundaries of each pill.

While you may not realize it yet, this simple exercise of selecting the number of pixels inside an upper and lower boundary region is *exactly* what advanced image descriptors such as Histogram of Oriented Gradients and SIFT are doing! In both cases, a critical step for these two image descriptors is to *count* the number of pixels in each range and then construct a histogram — this histogram is then manipulated in various fashions and returned as a feature vector (i.e. list of numbers) to abstractly quantify and represent the image. But we'll discuss that more once we get to **Module 10** (https://gurus.pyimagesearch.com/lessons/what-are-image-descriptors-feature-descriptors-and-feature-vectors/) on image descriptors and features.

# Summary

In this lesson we defined what an image gradient is: *a directional change in image intensity.*

We also learned how to manually compute the *change in direction* across our image at each pixel using nothing more than the neighborhood of pixel intensity values.

We then used these changes in direction to compute our *gradient orientation* — the direction in which the change in intensity is pointing — and the *gradient magnitude*, which is how strong the change in intensity is.

Of course, computing the gradient orientation and magnitude using our simple method is not the fastest way. Instead, we can rely on the Sobel and Scharr kernels, which allow us to obtain an *approximation* to the image derivative. Similar to smoothing and blurring, our image kernels *convolve* our input image with a kernel that is designed to approximate our gradient.

Finally, we learned how to use the `cv2.Sobel` OpenCV function to compute Sobel and Scharr gradient representations. Using these gradient representations we were able to pinpoint which pixels in the image had an orientation within the range $min_\theta <= \theta <= max_\theta$.

Image gradients are one of *the most important* image processing and computer vision building blocks you'll learn about. Behind the scenes, they are used for powerful image descriptor methods such as Histogram of Oriented Gradients and SIFT. They are used to construct saliency maps to reveal the most "interesting" regions of an image. And as we'll see in the next lesson, we'll see how image gradients are the cornerstone of the Canny edge detector for detecting edges in images.



# Downloads:

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

| Quizzes | | Status |
|---|---|---|
| 1 | Gradients Quiz (https://gurus.pyimagesearch.com/quizzes/gradients-quiz/) | |

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

## 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