1.12: Histograms

Histograms are prevalent in nearly every aspect of computer vision. We use grayscale histograms for thresholding. We use histograms for white balancing. We use color histograms for object tracking in images, such as with the CamShift algorithm. We use color histograms as features — include color histograms in multiple dimensions. And in an abstract sense, we use histograms of image gradients to form the HOG and SIFT descriptors. Even the extremely popular bag-of-visual-words representation used in image search engines and machine learning is a histogram as well! And in all likelihood, I'm sure this is not the first time you have ran across histograms in your studies.

So why are histograms so useful?

Because histograms capture the *frequency distribution* of a set of data. And it turns out that examining these frequency distributions is a very nice way to build simple image processing techniques — along with very powerful machine learning algorithms.

In this lesson we'll be focusing on grayscale and color histograms. But in future lessons we'll be exploring gradient histograms and the bag-of-visual-words model. This is certainly not the last time you will be seeing histograms.

Objectives:

By the end of this lesson you will:

1. Know what a histogram is.
2. Know how to compute a histogram in OpenCV.
3. Understand how to compute a grayscale histogram of an image.

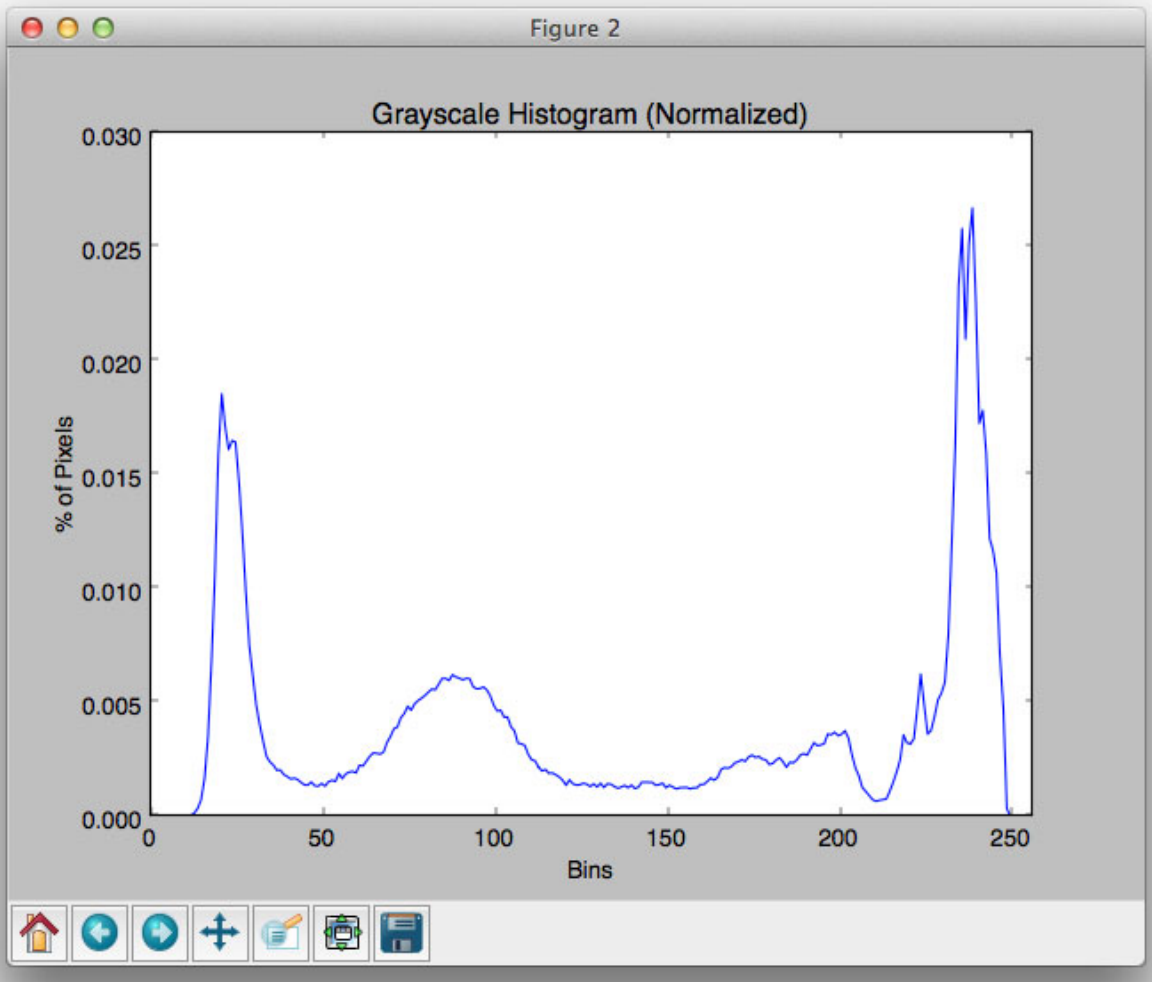
4. Write code to extract a "flattened" RGB histogram from an image.
5. Extract multi-dimensional color histograms from an image.

So, what's a histogram?

So, what exactly is a histogram? A histogram represents the distribution of pixel intensities (whether color or gray- scale) in an image. It can be visualized as a graph (or plot) that gives a high-level intuition of the intensity (pixel value) distribution. We are going to assume a RGB color space in this example, so these pixel values will be in the range of 0 to 255.

When plotting the histogram, the X-axis serves as our "bins." If we construct a histogram with 256 bins, then we are effectively counting the number of times each pixel value occurs. In contrast, if we use only 2 (equally spaced) bins, then we are counting the number of times a pixel is in the range *[0, 128]* or *[128, 255]*. The number of pixels binned to the x-axis value is then plotted on the y-axis.

Let's take at an example image to make this more clear:



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

FIGURE 1: AN EXAMPLE OF AN IMAGE HISTOGRAM.

In **Figure 1** we have plotted a histogram with 256-bins along the *x*-axis and the percentage of pixels falling into the given bins along the *y*-axis. Examining the histogram, note that there are three primary peaks. The first peak in the histogram is around *x=20* where we see a sharp spike in the number of pixels — clearly there is some sort of object in the image that has a very dark value.

We then see a much slower rising peak in the histogram, where we start to ascend around *x=50* and finally end the descent around *x=120*. This region probably refers to a background region of the image.

Finally, we see there is a very large number of pixels in the range *x=220* to *x=245*. It's hard to say exactly what this region is, but it must dominate a large portion of the image.

Note: *I'm intentionally not revealing which image I used to generate this histogram — I'll leave that as an exercise for you. I'm simply demonstrating my thought process as I look at a histogram. Being able to interpret and understand the data you are looking at, without necessarily knowing its source, is a good skill to have.*

By simply examining the histogram of an image, you get a general understanding regarding the contrast, brightness, and intensity distribution. If this concept seems new or foreign to you, don't worry — we'll be examining more examples like this one later in this lesson.

Using OpenCV to Compute Histograms

Now, let's start building some histograms of our own.

We will be using the `cv2.calcHist` function to build our histograms. Before we get into any code examples, let's quickly review the function:

```
cv2.calcHist(images, channels, mask, histSize, ranges)
```

- **images:** This is the image that we want to compute a histogram for. Wrap it as a list: `[myImage]` .
- **channels:** A list of indexes, where we specify the index of the channel we want to compute a histogram for. To compute a histogram of a grayscale image, the list would be `[0]` . To compute a histogram for all three red, green, and blue channels, the channels list would be `[0, 1, 2]` .
- **mask:** Remember learning about masks in **Section 1.4.8** (https://gurus.pyimagesearch.com/topic/masking/)? Well, here we can supply a mask. If a mask is provided, a histogram will be computed for *masked pixels only*. If we do not have a mask or do not want to apply one, we can just provide a value of `None` .

- **histSize:** This is the number of bins we want to use when computing a histogram. Again, this is a list, one for each channel we are computing a histogram for. The bin sizes do not all have to be the same. Here is an example of 32 bins for each channel: `[32, 32, 32]` .
- **ranges:** The range of possible pixel values. Normally, this is *[0, 256]* (this is not a typo — the ending range of the `cv2.calcHist` function is *non-inclusive* so you'll want to provide a value of 256 rather than 255) for each channel, but if you are using a color space other than RGB [such as HSV], the ranges might be different.)

Next up, we'll use the `cv2.calcHist` function to compute our first histogram.

Grayscale Histograms

Now that we have an understanding of the `cv2.calcHist` function, let's write some actual code.

grayscale_histogram.py — Python

```
1  # import the necessary packages
2  from matplotlib import pyplot as plt
3  import argparse
4  import cv2
5
6  # Construct the argument parser and parse the arguments
7  ap = argparse.ArgumentParser()
8  ap.add_argument("-i", "--image", required=True, help="Path to the image")
9  args = vars(ap.parse_args())
10
11 # load the image, convert it to grayscale, and show it
12 image = cv2.imread(args["image"])
```

This code isn't very exciting yet. All we are doing is importing the packages we will need, setting up an argument parser, and loading our image. We'll make use of the `matplotlib` package to make plotting our histograms easier.

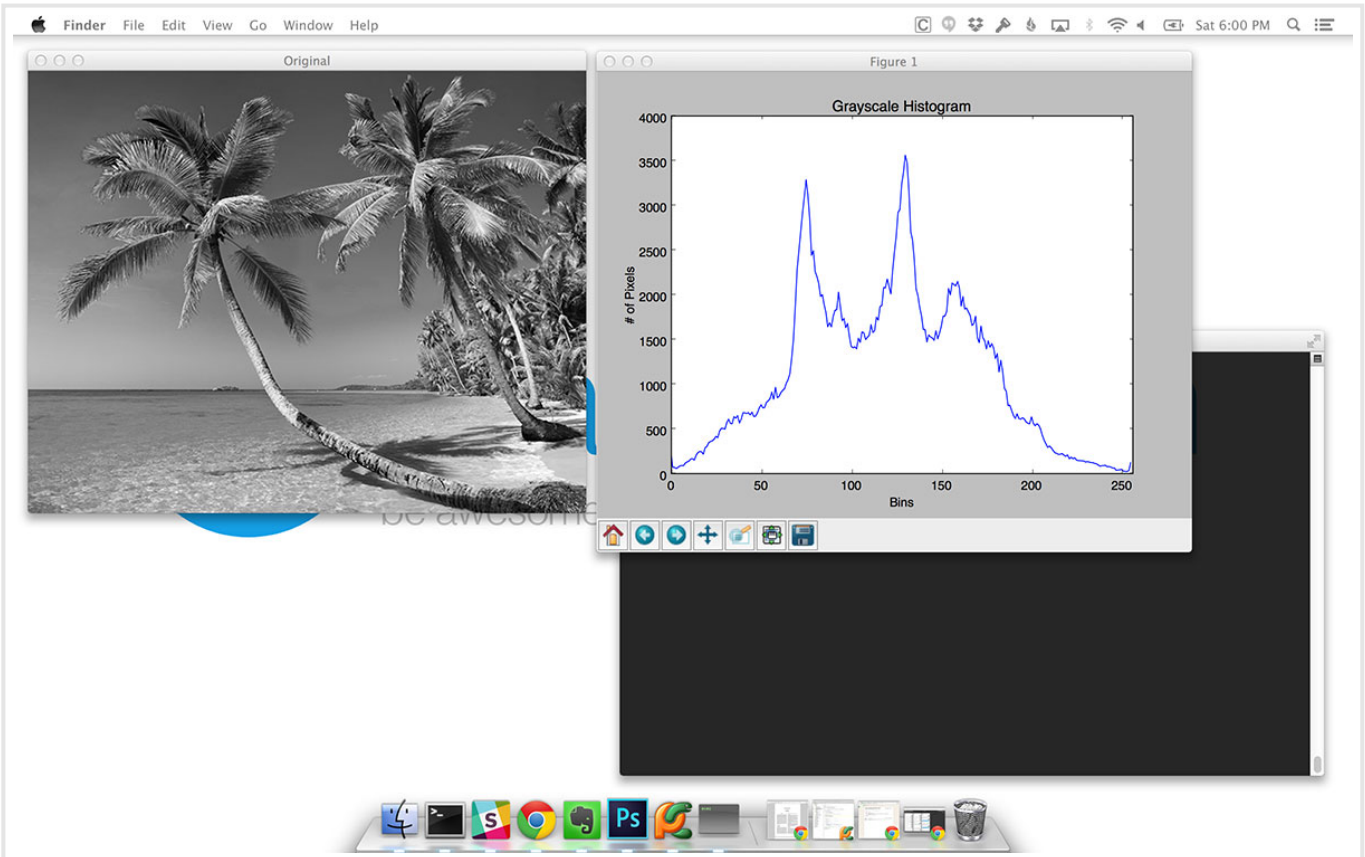
grayscale_histogram.py — Python

```
13 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
14 cv2.imshow("Original", image)
15
16 # construct a grayscale histogram
17 hist = cv2.calcHist([image], [0], None, [256], [0, 256])
18
19 # plot the histogram
20 plt.figure()
21 plt.title("Grayscale Histogram")
22 plt.xlabel("Bins")
23 plt.ylabel("# of Pixels")
24 plt.plot(hist)
25 plt.xlim([0, 256])
```

Now things are getting a little more interesting. On **Line 13**, we convert the image from the RGB colorspace to grayscale. **Line 17** computes the actual histogram. Go ahead and match the arguments of the code up with the function documentation above. We can see that our first parameter is the

grayscale image. A grayscale image has only one channel, so we have a value of `[0]` for `channels`. We don't have a mask, so we set the mask value to `None`. We will use 256 bins in our histogram, and the possible values range from 0 to 255.

Finally, a call to `plt.plot()` plots our grayscale histogram, the results of which can be seen in **Figure 2** below:



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

FIGURE 2: OUR *UNNORMALIZED* GRAYSCALE HISTOGRAM.

Not bad. How do we interpret this histogram? Well, the bins (0-255) are plotted on the x-axis. And the y-axis counts the number of pixels in each bin. The majority of the pixels fall in the range of roughly 60 to 180. Looking at both tails of the histogram, we can see that very few pixels fall in the range [0, 50] and [200, 255] — this implies that there are very few "black" and very few "white" pixels in the image.

An astute eye would notice that I referred to the histogram as *unnormalized* in the caption of **Figure 2**.

Why would I call it that? Why is the histogram "unnormalized"?

An unnormalized histogram counts the raw frequencies of the distribution. Consider counting the number of different colors of M&M's in a bag. We would end up with an integer count for each of the individual colors.

On the other hand, what if I wanted the percentage of each of the colors?

Well that's easy enough to obtain! I would simply divide each of the integer counts by the *total number* of M&M's in the bag. So instead of a *raw frequency* histogram, I would end up with a *normalized* histogram that counts the *percentage* of each color. And by definition, the sum of a normalized histogram is exactly *1*. (provided you are using L1-normalization, but we'll get into that later).

So is there a reason why I would prefer a normalized histogram over an unnormalized one?

It turns out, there is. Let's play a little thought experiment:

And in this thought experiment we want to compare the histograms of two images. These images are identical in every way, shape, and form — with one exception. The first image is *half the size* of the second image. And when we went to compare the histograms, while the shape of the distributions would look identical, we would notice that the pixels counts along the *y*-axis would be dramatically different. In fact, the *y*-axis counts for the first image would be *exactly half* the *y*-axis counts for the second image?

The reason for this being?

We were comparing *raw frequency counts* versus the *percentage counts*!

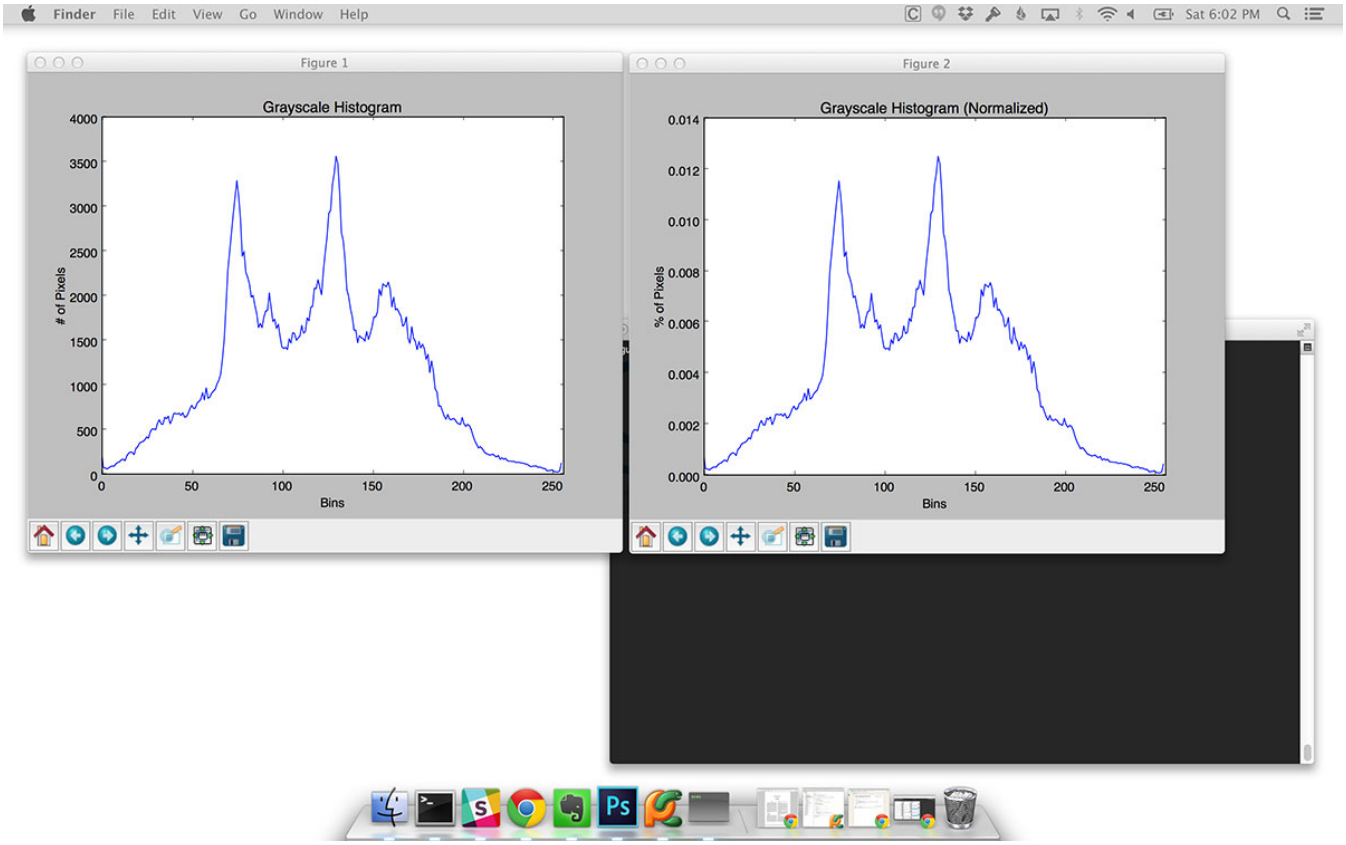
So with this in mind, let's see how we can normalize a histogram and obtain our percentage counts for each pixel:

grayscale_histogram.py — Python

```
27 # normalize the histogram
28 hist /= hist.sum()
29
30 # plot the normalized histogram
31 plt.figure()
32 plt.title("Grayscale Histogram (Normalized)")
33 plt.xlabel("Bins")
34 plt.ylabel("% of Pixels")
35 plt.plot(hist)
36 plt.xlim([0, 256])
37 plt.show()
```

The normalization of the histogram takes only a single line of code, which we can see on **Line 28**: here we are simply dividing the raw frequency counts for each bin of the histogram by the sum of the counts — this leaves us with the *percentage* of each bin rather than the *raw count* of each bin.

Lines 31-37 then plot our normalized histogram, which looks identical to **Figure 2** above, only we can now see that the scale of our histogram along the *y*-axis is measured in percent:



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

FIGURE 3: CONSTRUCTING A NORMALIZED GRAYSCALE HISTOGRAM. NOTICE HOW THE *SHAPE* OF THE DISTRIBUTIONS THEMSELVES ARE IDENTICAL; HOWEVER, THE SCALE ALONG THE *Y*-AXIS IS DIFFERENT.

Throughout the rest of this lesson we'll be using *unnormalized* histograms as a matter of simplicity; however, I wanted to bring attention to this detail now as it will be extremely important in the future once we get to building image classification systems and exploring image search engines.

Color Histograms

In the previous section we explored grayscale histograms. Now let's move on to computing a histogram for each channel of the image.

color_histograms.py — Python

```python
# import the necessary packages
from matplotlib import pyplot as plt
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 and show it
image = cv2.imread(args["image"])
cv2.imshow("Original", image)
```

Again, we'll import the packages that we'll need, utilizing `matplotlib` once more to plot the histograms.

Let's examine some more code:

color_histograms.py Python

```
15 # grab the image channels, initialize the tuple of colors and the
16 # figure
17 chans = cv2.split(image)
18 colors = ("b", "g", "r")
19 plt.figure()
20 plt.title("'Flattened' Color Histogram")
21 plt.xlabel("Bins")
22 plt.ylabel("# of Pixels")
23
24 # loop over the image channels
25 for (chan, color) in zip(chans, colors):
26     # create a histogram for the current channel and plot it
27     hist = cv2.calcHist([chan], [0], None, [256], [0, 256])
28     plt.plot(hist, color = color)
29     plt.xlim([0, 256])
```

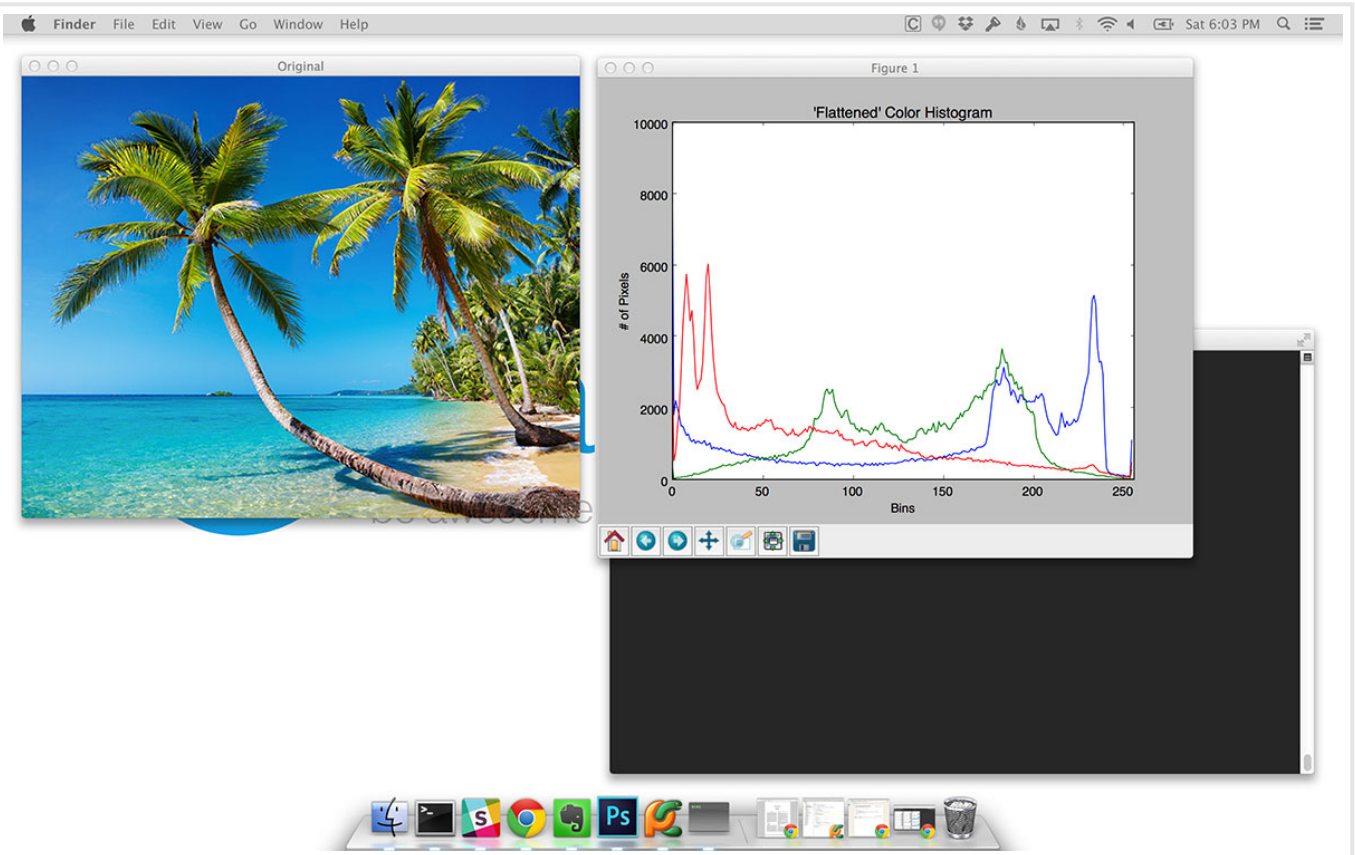
The first thing we are going to do is split the image into its three channels: Blue, Green, and Red. Normally, we read this is a red, green, blue (RGB). However, remember that OpenCV reverses this order to BGR (and I'll keep repeating it until you want to kill me — I can't even begin to tell you the number of times I've forgot about this little caveat when I started learning OpenCV). We then initialize a tuple of strings representing the colors. We take care of all this on **Lines 17 and 18**.

On **Lines 20-22** we set up our PyPlot figure. We'll plot the bins on the x-axis and the number of pixels placed into each bin on the y-axis.

We then reach a `for` loop on **Line 25**: we start looping over each of the channels in the image.

Then, for each channel we compute a histogram on **Line 27**. The code is identical to that of computing a histogram for the grayscale image; however, we are doing it for each Red, Green, and Blue channel, allowing us to characterize the distribution of pixel intensities. We add our histogram to the plot on **Line 29**.

We can examine our color histogram in **Figure 4** below:



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

FIGURE 4: CREATING A COLOR HISTOGRAM FOR EACH CHANNEL OF THE IMAGE.

We see there is a sharp peak in the green histogram around bin 100. We see that most of the green pixels in the image are contained in the range [85, 200] — these regions are a mid-range to light green from the green vegetation and trees in the beach image.

We also see a lot of lighter blue pixels in our image. Considering that we have both views of the crystal clear ocean *and* the cloudless blue sky, this should not be a surprise.

Up until this point, we have computed a histogram for only one channel at a time. Now we move on to multi-dimensional histograms and take into consideration two channels at a time.

The way I like to explain multi-dimensional histograms is to use the word **AND**.

For example, we can ask a question such as: "How many pixels have a Red value of 10 **AND** a Blue value of 30?" "How many pixels have a Green value of 200 **AND** a Red value of 130?" By using the conjunctive **AND**, we are able to construct multi-dimensional histograms.

It's that simple. Let's check out some code to automate the process of building a 2D histogram:

color_histograms.py Python

```
31 # let's move on to 2D histograms -- we need to reduce the
32 # number of bins in the histogram from 256 to 32 so we can
33 # better visualize the results
34 fig = plt.figure()
35
36 # plot a 2D color histogram for green and blue
37 ax = fig.add_subplot(131)
38 hist = cv2.calcHist([chans[1], chans[0]], [0, 1], None, [32, 32],
39     [0, 256, 0, 256])
40 p = ax.imshow(hist, interpolation="nearest")
41 ax.set_title("2D Color Histogram for G and B")
42 plt.colorbar(p)
43
44 # plot a 2D color histogram for green and red
45 ax = fig.add_subplot(132)
46 hist = cv2.calcHist([chans[1], chans[2]], [0, 1], None, [32, 32],
47     [0, 256, 0, 256])
48 p = ax.imshow(hist, interpolation="nearest")
49 ax.set_title("2D Color Histogram for G and R")
50 plt.colorbar(p)
51
52 # plot a 2D color histogram for blue and red
53 ax = fig.add_subplot(133)
54 hist = cv2.calcHist([chans[0], chans[2]], [0, 1], None, [32, 32],
55     [0, 256, 0, 256])
56 p = ax.imshow(hist, interpolation="nearest")
57 ax.set_title("2D Color Histogram for B and R")
58 plt.colorbar(p)
59
60 # finally, let's examine the dimensionality of one of the 2D
61 # histograms
62 print("2D histogram shape: {}, with {} values".format(
63     hist.shape, hist.flatten().shape[0]))
```

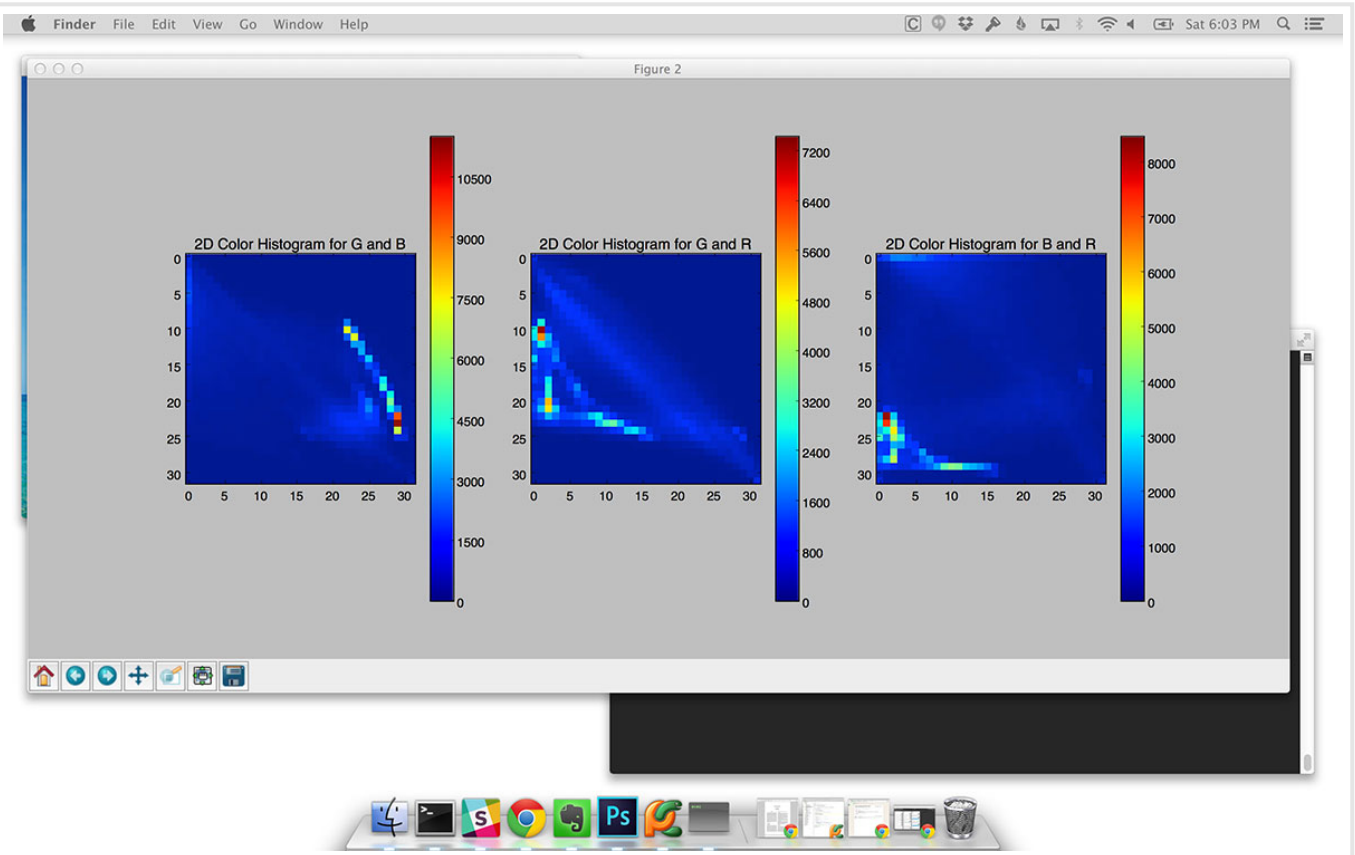
Yes, this is a fair amount of code. But that's only because we are computing a 2D color histogram for each combination of RGB channels: Red and Green, Red and Blue, and Green and Blue.

Now that we are working with multi-dimensional histograms, we need to keep in mind the number of bins we are using. In previous examples, I've used 256 bins for demonstration purposes. However, if we used 256 bins for each dimension in a 2D histogram, our resulting histogram would have 65,536 separate pixel counts. Not only is this wasteful of resources, it's not practical. Most applications use somewhere between 8 and 64 bins when computing multi-dimensional histograms. As **Lines 38-39** show, I am now using 32 bins instead of 256.

The most important take away from this code can be seen by inspecting the first arguments to the `cv2.calcHist` function. Here we see that we are passing in a list of two channels: the Green and Blue channels. And that's all there is to it.

So how is a 2D histogram stored in OpenCV? It's a 2D NumPy array. Since I used 32 bins for each channel, I now have a *32 x 32* histogram.

How do we visualize a 2D histogram? Let's take a look at **Figure 5** where we see three graphs:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/color_histograms_2d.jpg)

FIGURE 5: COMPUTING 2D COLOR HISTOGRAMS FOR EACH COMBINATION OF THE RED, GREEN, AND BLUE CHANNELS.

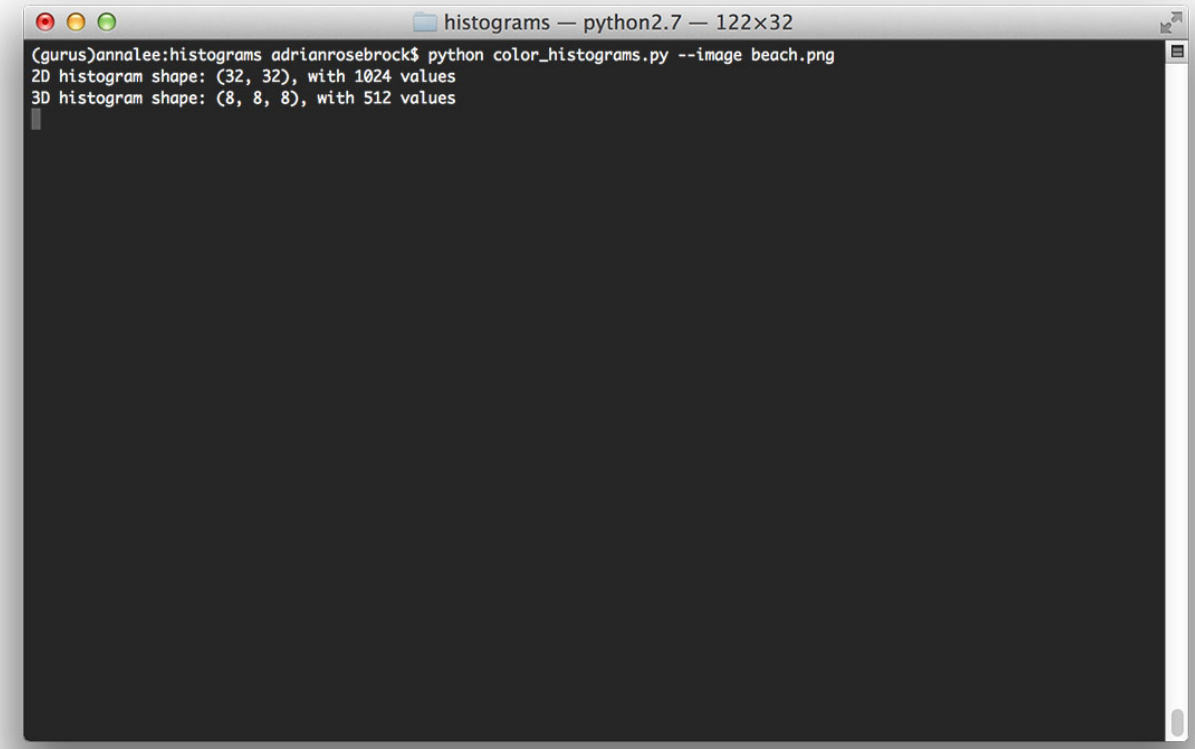
The first is a 2D color histogram for the Green and Blue channels, the second for Green and Red, and the third for Blue and Red. Shades of blue represent low pixel counts, whereas shades of red represent large pixel counts (i.e. peaks in the 2D histogram). We tend to see many peaks in the Green and Blue histogram, where *x=28* and *y=27*. This corresponds to the green pixels of the vegetation and trees and the blue of the sky and ocean.

Using a 2D histogram takes into account two channels at a time. But what if we wanted to account for all *three* RGB channels? You guessed it. We're now going to build a 3D histogram.

color_histograms.py Python

```
65 # our 2D histogram could only take into account 2 out of the 3
66 # channels in the image so now let's build a 3D color histogram
67 # (utilizing all channels) with 8 bins in each direction -- we
68 # can't plot the 3D histogram, but the theory is exactly like
69 # that of a 2D histogram, so we'll just show the shape of the
70 # histogram
71 hist = cv2.calcHist([image], [0, 1, 2],
72     None, [8, 8, 8], [0, 256, 0, 256, 0, 256])
73 print("3D histogram shape: {}, with {} values".format(
74     hist.shape, hist.flatten().shape[0]))
75
76 # Show our plots
77 plt.show()
```

The code here is very simple — it's just an extension from the code above. We are now computing an *8 x 8 x 8* histogram for each of the RGB channels. We can't visualize this histogram, but we can see that the shape is indeed `(8, 8, 8)` with 512 values:



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

FIGURE 6: HERE WE CAN SEE THAT OUR 3D COLOR HISTOGRAM DOES INDEED HAVE *8 X 8 X 8 = 512* BINS.

Feedback

Histogram Equalization

Histogram equalization improves the contrast of an image by "stretching" the distribution of pixels. Consider a histogram with a large peak at the center of it. Applying histogram equalization will stretch the peak out towards the corner of the image, thus improving the global contrast of the image. Histogram equalization is applied to grayscale images.

This method is useful when an image contains foregrounds and backgrounds that are both dark or both light. It tends to produce unrealistic effects in photographs; however, is normally useful when enhancing the contrast of medical or satellite images.

Regardless of whether you are applying histogram equalization to a photograph, a satellite image, or an X-ray, we first need to see some code so we can understand what is going on:

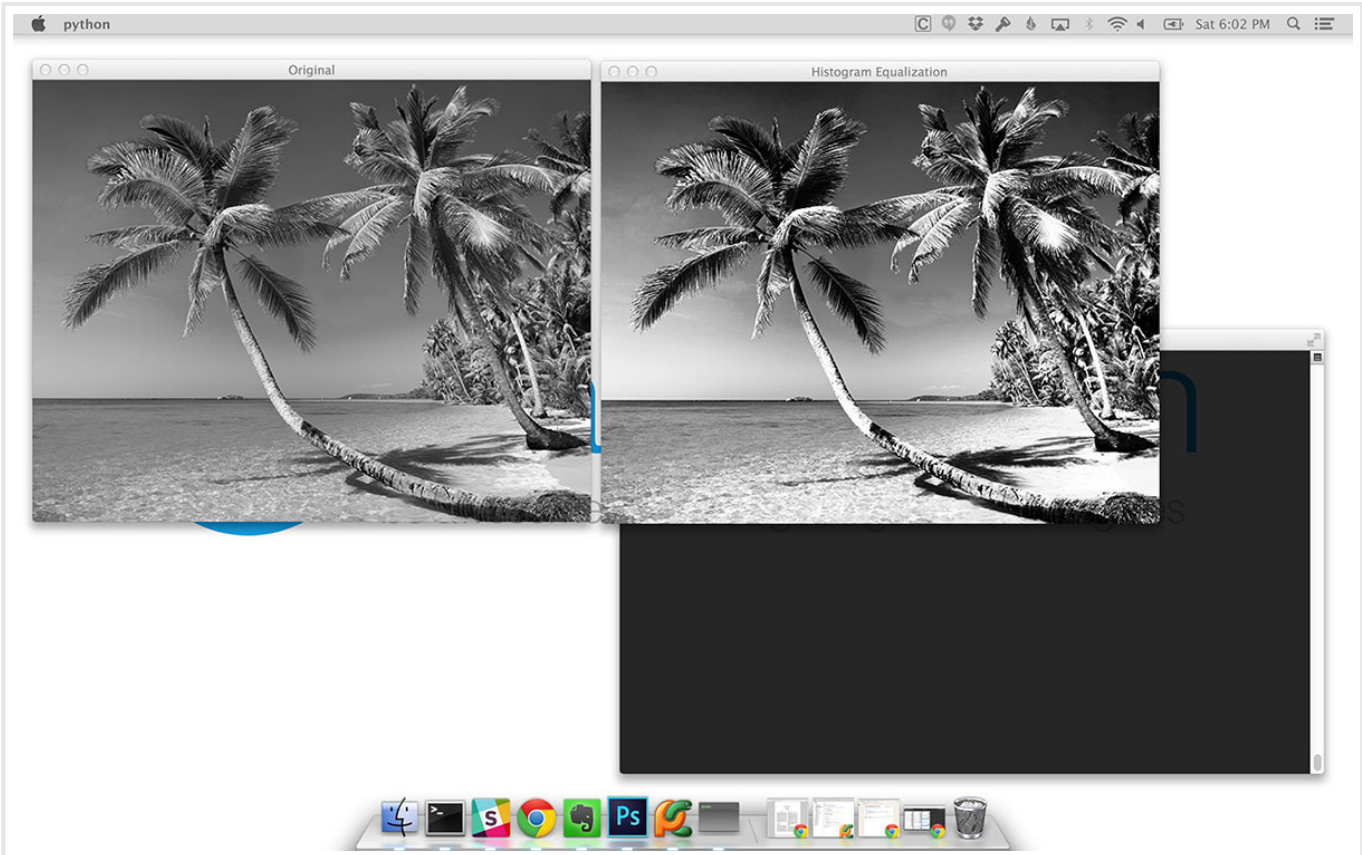
```
equalize.py                                                    Python
```

```
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 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
13
14 # apply histogram equalization to stretch the constrast of our image
15 eq = cv2.equalizeHist(image)
16
17 # show our images -- notice how the constrast of the second image has
18 # been stretched
19 cv2.imshow("Original", image)
20 cv2.imshow("Histogram Equalization", eq)
21 cv2.waitKey(0)
```

Lines 1-11 handle our standard practice of importing packages, parsing arguments, and loading our image. We then convert our image to grayscale on **Line 12**.

Performing histogram equalization is done using just a single function: `cv2.equalizeHist` , which accepts a single parameter, the grayscale image we want to perform histogram equalization on. The last couple lines of code display our histogram equalized image.

The result of applying histogram equalization can be seen in **Figure 7**:



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

FIGURE 7: *LEFT:* THE ORIGINAL BEACH IMAGE. *RIGHT:* THE BEACH IMAGE AFTER APPLYING HISTOGRAM EQUALIZATION.

On the *left*, we have our original beach image. Then, on the *right*, we have our histogram equalized beach image. Notice how the contrast of the image has been radically changed and now spans the entire range of *[0, 255]*.

Histograms and Masks

In **Section 1.4.8** (https://gurus.pyimagesearch.com/topic/masking/), I mentioned that masks can be used to focus on only regions of an image that interest us. We are now going to construct a mask and compute color histograms for only the masked region.

First, we need to define a convenience function to plot our histograms and save us from writing repetitive lines of code:

histogram_with_mask.py — Python

```
1  # import the necessary packages
2  from matplotlib import pyplot as plt
3  import numpy as np
4  import cv2
5
6  def plot_histogram(image, title, mask=None):
7      # grab the image channels, initialize the tuple of colors and
8      # the figure
9      chans = cv2.split(image)
10     colors = ("b", "g", "r")
11     plt.figure()
12     plt.title(title)
13     plt.xlabel("Bins")
14     plt.ylabel("# of Pixels")
15
16     # loop over the image channels
17     for (chan, color) in zip(chans, colors):
18         # create a histogram for the current channel and plot it
19         hist = cv2.calcHist([chan], [0], mask, [256], [0, 256])
20         plt.plot(hist, color=color)
21         plt.xlim([0, 256])
```

On **Lines 1-4** we import our packages, then on **Line 6** we define `plot_histogram`. This function accepts three parameters: an `image`, the `title` of our plot, and a `mask`. The `mask` defaults to `None` if we do not have a mask for the image.

The body of our `plot_histogram` function simply computes a histogram for each channel in the image and plots it, just as in previous examples in this section.

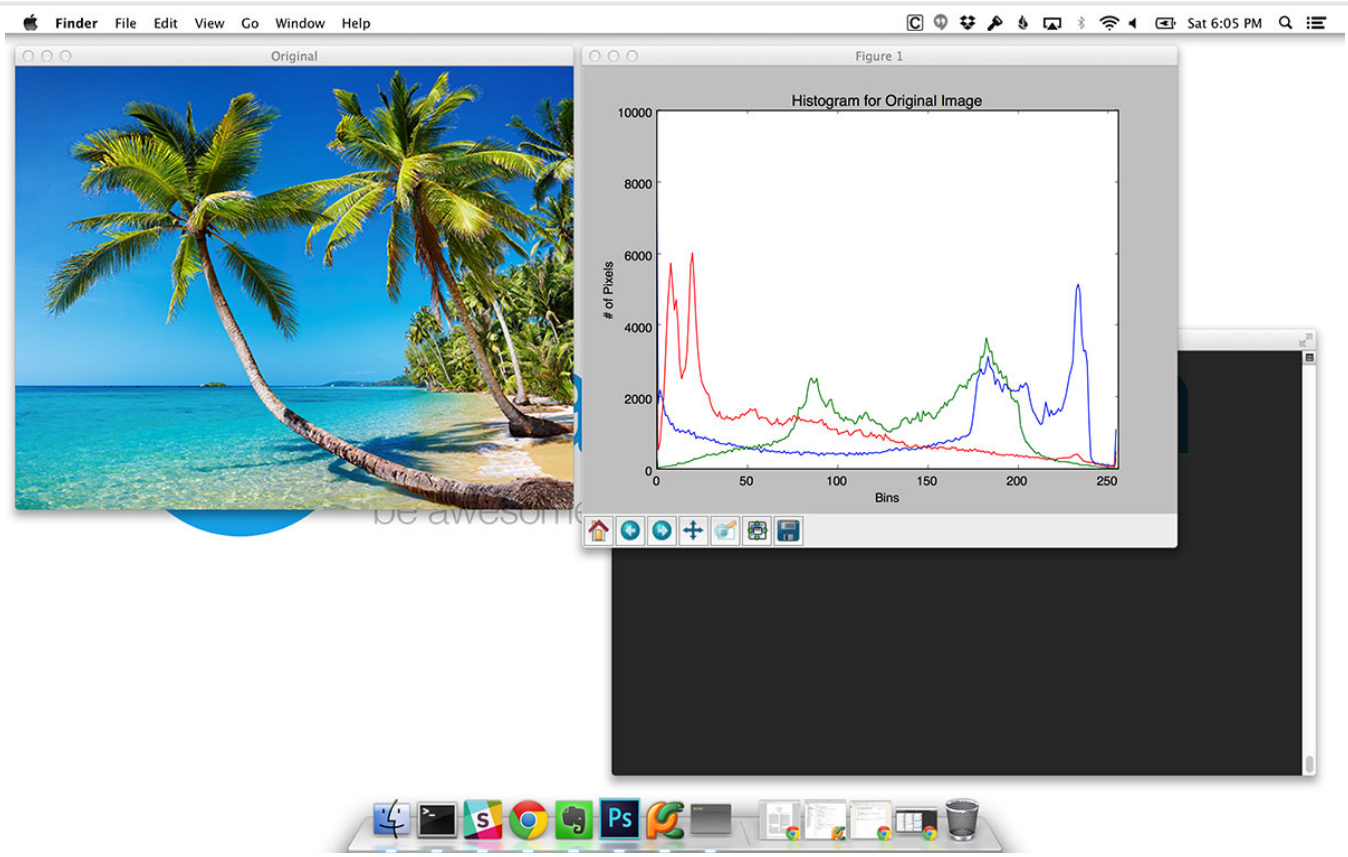
Now that we have a function to help us easily plot histograms, let's move into the bulk of our code:

histogram_with_mask.py — Python

Feedback

```
23 # load the beach image and plot a histogram for it
24 image = cv2.imread("beach.png")
25 cv2.imshow("Original", image)
26 plot_histogram(image, "Histogram for Original Image")
```

We start off by loading our beach image from disk on **Line 24**, displaying it on screen on **Line 25**, and then plotting a color histogram for each channel of the beach image on **Line 26**. The plot for the beach image can be seen in **Figure 8** below:



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

FIGURE 8: THE ORIGINAL RGB HISTOGRAM OF THE BEACH IMAGE WITH NO MASKING APPLIED.

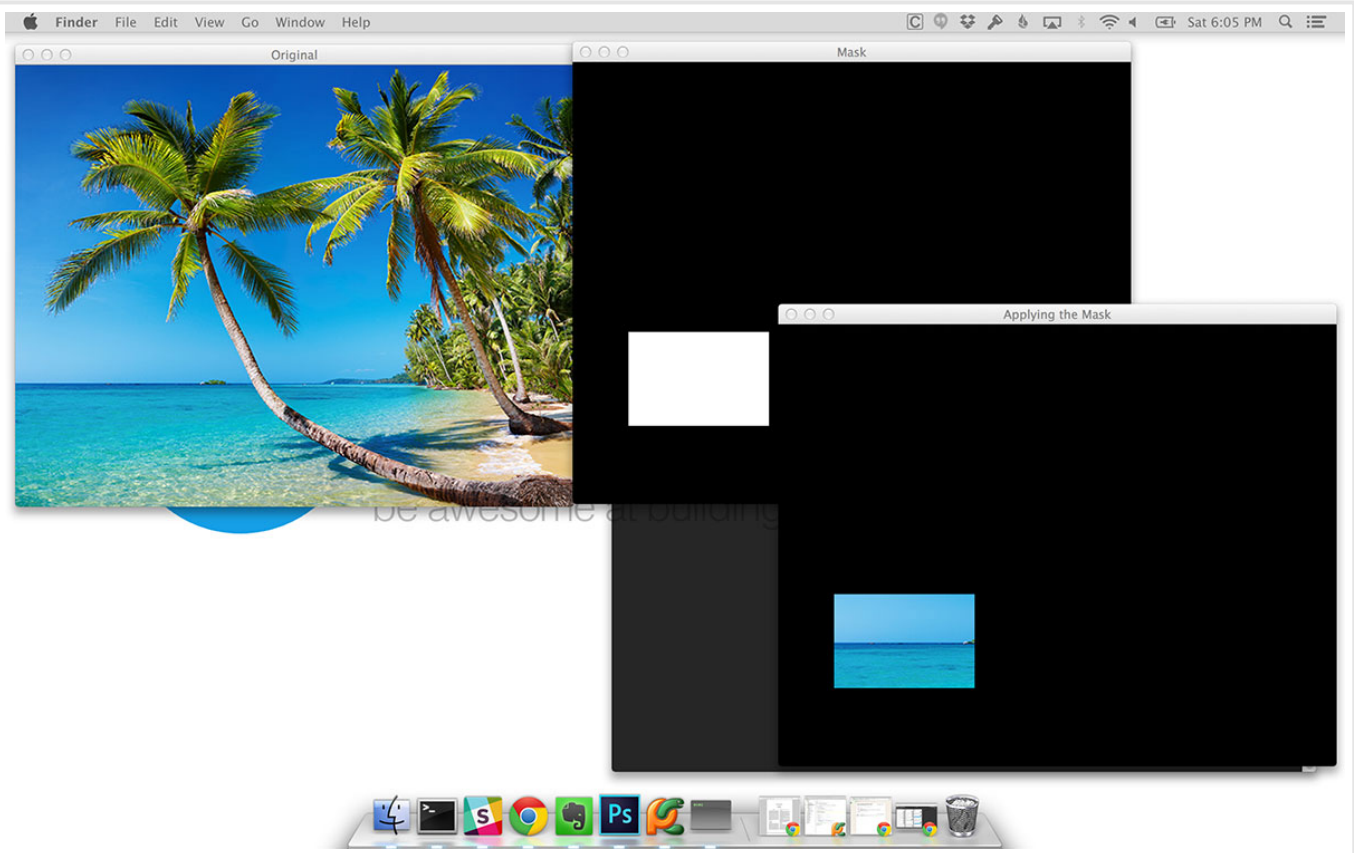
Now we are ready to construct a mask for the image:

histogram_with_mask.py — Python

```
28 # construct a mask for our image -- our mask will be BLACK for regions
29 # we want to IGNORE and WHITE for regions we want to EXAMINE
30 mask = np.zeros(image.shape[:2], dtype="uint8")
31 cv2.rectangle(mask, (60, 290), (210, 390), 255, -1)
32 cv2.imshow("Mask", mask)
33
34 # what does masking our image look like?
35 masked = cv2.bitwise_and(image, image, mask=mask)
36 cv2.imshow("Applying the Mask", masked)
```

We define our `mask` as a NumPy array, with the same width and height as our beach image on **Line 30**. We then draw a white rectangle starting from point *(60, 210)* to point *(290, 390)* on **Line 31**. This rectangle will serve as our mask — **only pixels in our original image belonging to the masked region will be considered in the histogram computation.**

To visualize our mask, we apply a bitwise AND to the beach image (**Line 25**), the results of which can be seen in **Figure 9**:



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

FIGURE 9: *LEFT:* OUR RECTANGULAR MASK. *RIGHT:* APPLYING OUR MASK TO THE BEACH IMAGE USING A BITWISE AND. NOW WE SEE ONLY THE BLUE SKY — THE REST OF THE IMAGE IS IGNORED.

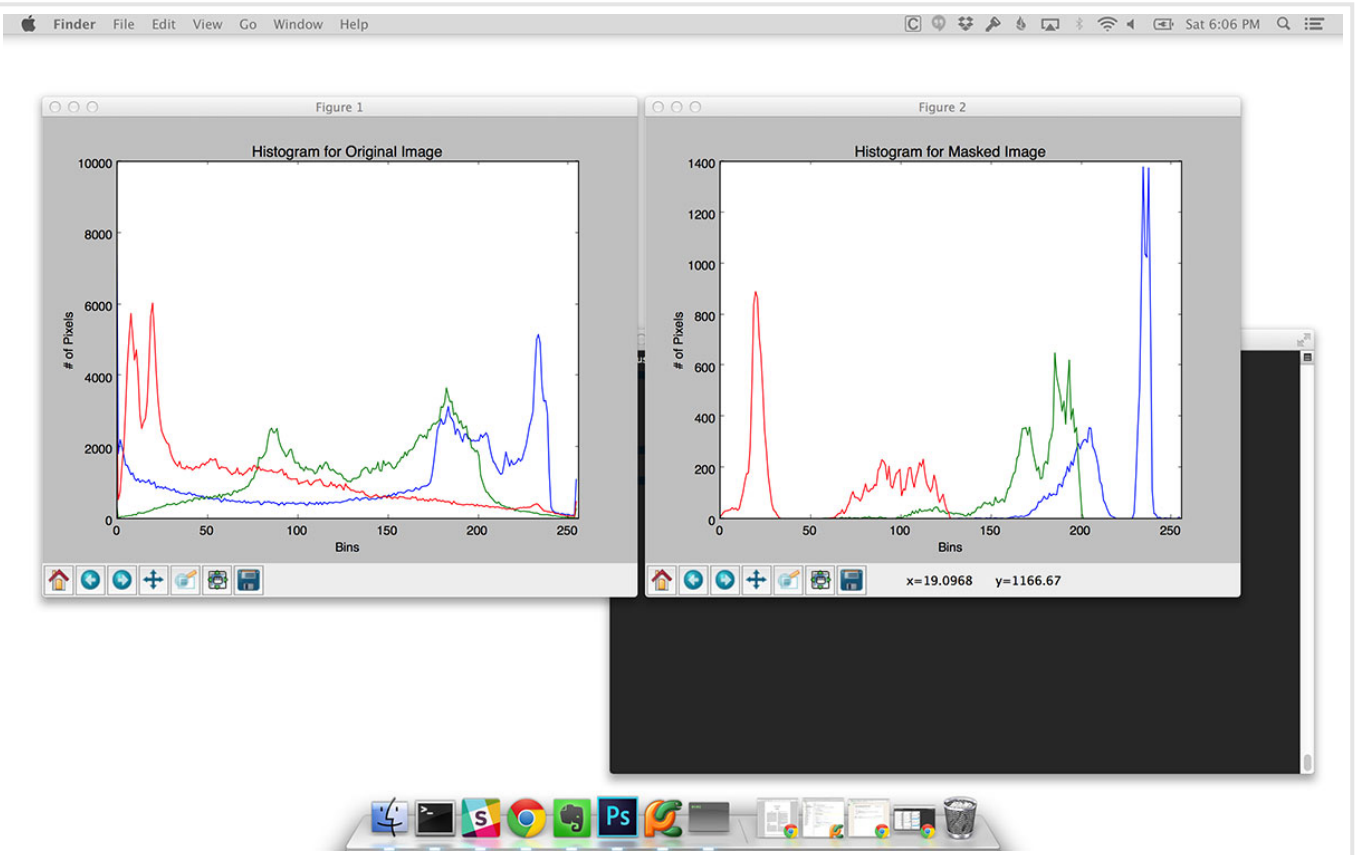
Notice how the image in the *middle* is simply a white rectangle, but when we apply our mask to to beach image, we only see the blue ocean and sky (*right*).

Finally, let's compute a histogram for the image, but only including the pixels in the masked region:

histogram_with_mask.py — Python

```
38 # compute a histogram for our image, but we'll only include pixels in
39 # the masked region
40 plot_histogram(image, "Histogram for Masked Image", mask=mask)
41
42 # show our plots
43 plt.show()
```

We can see our masked histogram in **Figure 10** below:



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

FIGURE 10: THE RESULTING HISTOGRAM OF THE MASKED IMAGE IN FIGURE 9. RED CONTRIBUTES VERY LITTLE IN OUR IMAGE AND IS TOWARDS THE DARKER END OF THE DISTRIBUTION. SOME LIGHTER GREEN VALUES ARE PRESENT, AND MANY LIGHT BLUE COLORS, CORRESPONDING TO THE OCEAN AND SKY IN THE IMAGE.

On the *left* we have our histogram of the **original image**, whereas on the *right* we have our histogram for the **masked image**.

For the masked image, most red pixels fall in the range *[10, 25]*, indicating that red pixels contribute very little to our image. This makes sense, since our ocean and sky are blue. Green pixels are then present, but these are toward the lighter end of the distribution, which corresponds to the green foliage and trees. Finally, our blue pixels fall in the brighter range and are obviously our blue ocean and sky.

Most importantly, compare our masked color histograms in *(right)* to the unmasked color histograms in *(left)* above. Notice how how dramatically different the color histograms are. By utilizing masks, we are able to apply our computation only to the specific regions of the image that interest us — in this example, we simply wanted to examine the distribution of the blue sky and ocean.

Summary

In this lesson you have learned all about histograms. Histograms are very simple, but are very powerful tools to have. They are used extensively for thresholding, color correction, and even image features! Make sure you have a good grasp of histograms, you'll be certainly using them in the future.

Downloads:

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

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

← Previous Lesson (https://gurus.pyimagesearch.com/lessons/contours/) Next Lesson → (https://gurus.pyimagesearch.com/lessons/connected-component-labeling/)

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