Comp Photography
Final Project

Osman Shawkat

Fall 2016

oshawkat@gatech.edu

Automated Brightness Correction

Correcting for saturation and brightness in over or underexposed images

The Goal

With the increasing ubiquity of portable cameras and cheap storage, the time required for constructing a scene and calibrating the camera for casual photos is reduced. Unfortunately, cameras are still unable to capture the full brightness range of the eye and so it is easy to produce over or underexposed photos, particularly when shooting around bright light sources (eg sun, street lamps).

This project aims to develop a fast, automated method for correcting overly bright or dark regions of a photo, improving contrast and outputting a wider brightness range to better capture the subject of the photo

Scope Changes

Did you run into issues that required you to change project scope from your proposal?

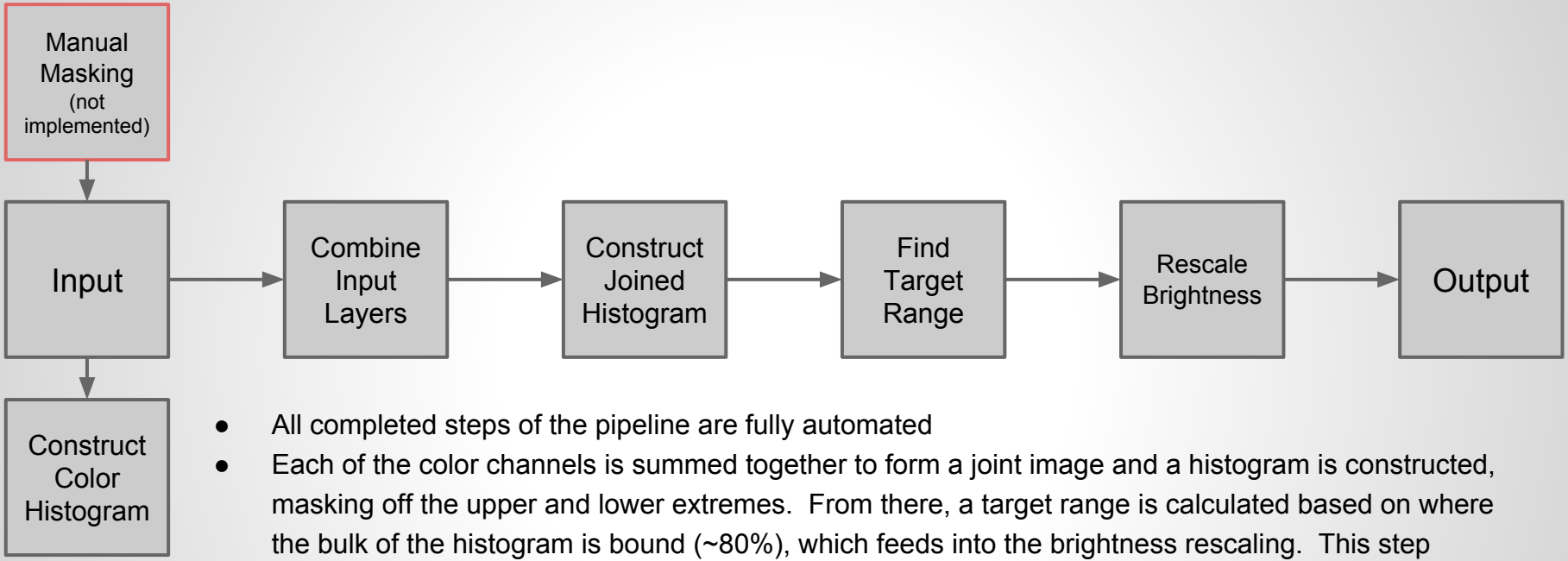
- The scope of the project did not change exactly so much as new challenges were discovered, as will be explained in the coming sections, and there are some areas for further processing and improvement that could not be adequately addressed due to time constraints

Showcase what you did on **This One Single Slide**. That might be challenging. You may use several images and format how you wish; but this single slide should be a good pictorial representation of your work. Be creative.

Input

Output

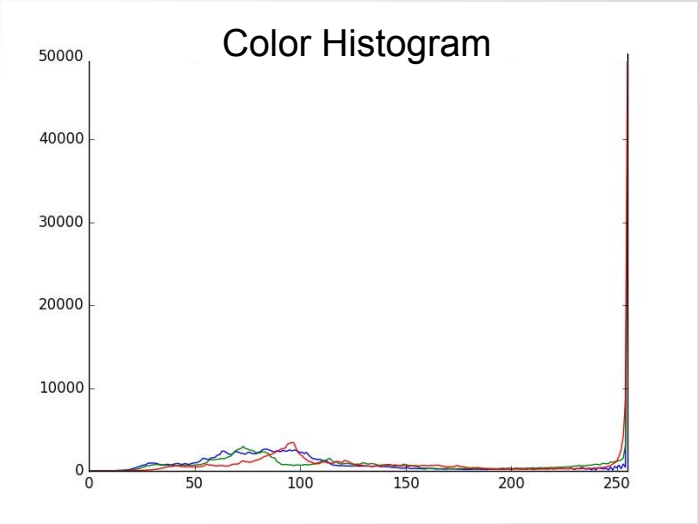
Pipeline



- All completed steps of the pipeline are fully automated
- Each of the color channels is summed together to form a joint image and a histogram is constructed, masking off the upper and lower extremes. From there, a target range is calculated based on where the bulk of the histogram is bound (~80%), which feeds into the brightness rescaling. This step effectively shifts and expands the histogram range to better showcase the original image
- Histogram creation relies heavily on the CV2 calcHist() function
- A color histogram is generated for the input image for reference purposes
- For better results, a mask can be manually created to highlight just the subject, thereby reducing the artifacts in the rest of the image. This portion was not implemented here

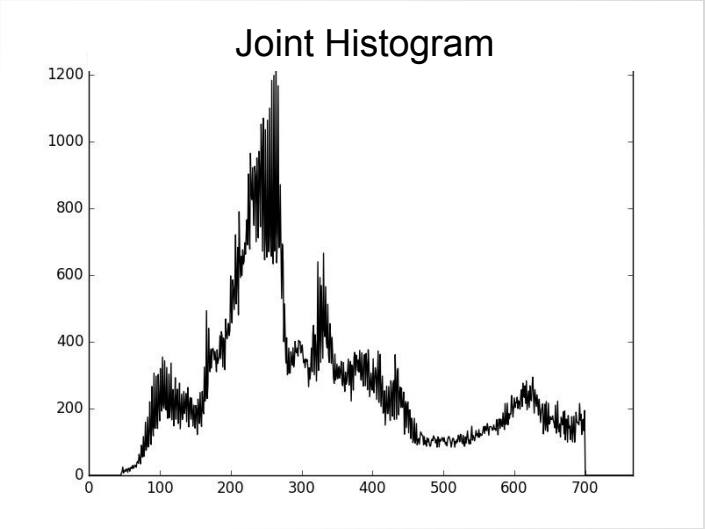
Demonstration

Input



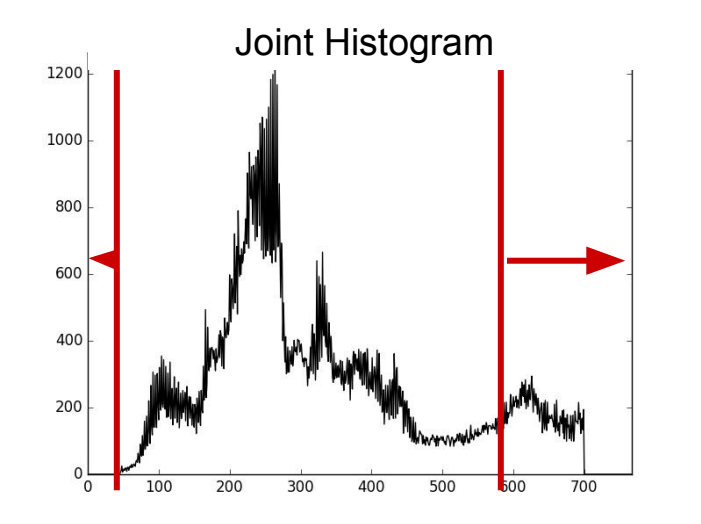
- A visual inspection of the image shows the subject is situated between the sun and the camera, resulting in an overly bright background and a darker subject
- Constructing a histogram from the original image, we see that there is a tall peak at the upper extreme of the spectrum, confirming the saturation seen in the input. The remaining pixels generally fall between values of ~50-150, leaving much of the spectrum unutilized

Demonstration



- To ensure color and brightness accuracy in this process, the channels of the input image are summed together to form a single 2D array. With three channels (of uint8 input), the expected range grows from [0, 255] to [0, 765]
- With this joint image, we can mask out the brightest and darkest regions, thereby concentrating the resulting histogram away from the extremes
    - A joint image is necessary here as masking the extremes for each color channel could change the visible color of a pixel that has high intensity in only some channels. The goal is to correct for overexposure, which saturates all channels, rather than dimming a vivid blue sky
    - The mask was created to only keep joint pixel values between 45 and 700

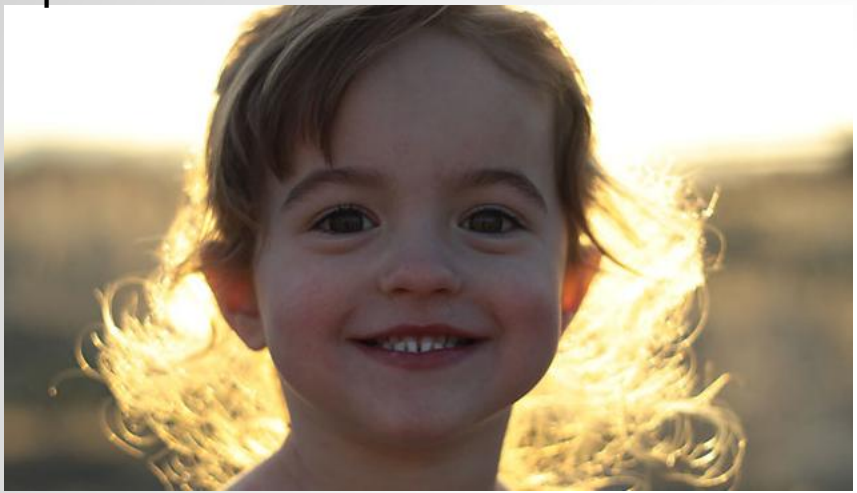
Demonstration



- From this joint histogram, the program will automatically find the interval that contains a certain percentage of the pixels. Here, it is set to exclude the top 10%. This produces a range from 46 to 586
- The pixel values in the original image are then linearly scaled to match the corresponding transformation from this interval to the full joint image range. In this example, most pixels increase in intensity and more of the range is used.
    - Colors are largely preserved by keeping the same pixel intensity ratios across channels
- Note that some of the above histogram is outside the interval and there will still be some values at the extreme if only the transformation above is applied
    - To address the pixel values pushed to the extremes in the transformation, the lightest and darkest of these have their values multiplied by a constant factor
    - One correction is to copy over the input image values to replace the masked areas and lightening/darkening these extremes as needed

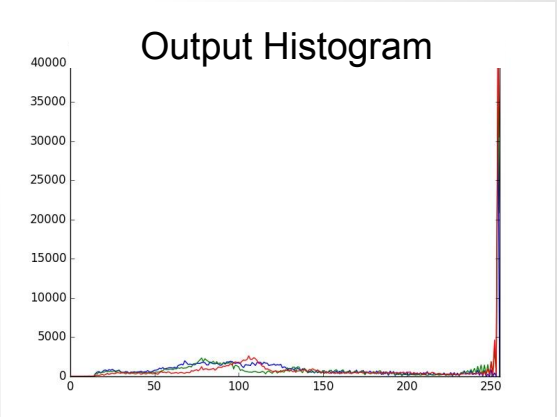
Demonstration

Input

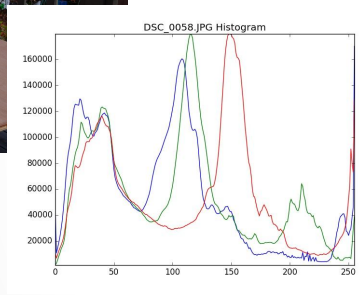


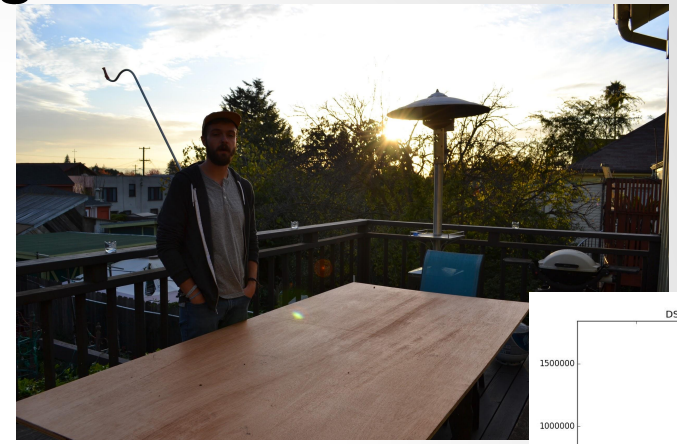
Output

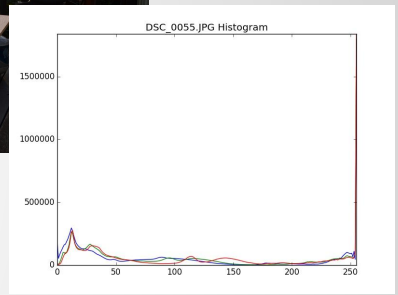
- The output image has appreciably better color, contrast, and depth on the subject's face though there are some artifacts in the background, specifically around the borders of the saturated areas
- Note that the histogram of the output image still shows a peak close to 255 but it is reduced in height. The output also makes better use of the full range, relative to the input
- Additional examples can be found here



Project Development - Histograms







- Histograms are one of the primary tools used in this program, both for representing data as well as for automating the brightness correction in a way that can be generalized to a variety of images
- On the left we see a sample image and its corresponding histogram. This represents a ‘good’ picture, with significant pixel counts across the intensity ranges and peaks away from the edges. There is still some saturation (eg the upper portion of the window frame) but it is relatively small and contained
- On the other hand, the example on the right is a good candidate for brightness correction as the histogram is largely flat throughout the range but has a very tall peak at 255. This is easily corroborated by the picture as much of the sky is saturated and the background and subject are otherwise overly dark, making it difficult to see much detail

Project Development - Calculating Histograms

- The first histogram, produced by the createHistogram() function, generates a list of histograms, one entry for each layer in the image. This helps to showcase where exactly the various peaks are and how much of the intensity range is being used
  - The OpenCV library function, cv2.calcHist(), is used to generate each individual histogram array
  - Channels are iterated over and corresponding histograms are appended to the output list
  - The function assumes that the input image is of type uint8 and therefore segments into 256 bins
- As will become apparent in when producing a histogram for the joint image, the histogram functions accept a mask to exclude certain areas of the image from the calculation. This functionality is natively supported by calcHist() but, as the function accepts a mask of the same size as the input image, the mask must also be sliced to correspond with the channel
- There is also a similar function, createJoinedHistogram(), that produces a histogram for a single 2D array with values up to 768. This bound was chosen as it represents the max of a 3 layer uint8 image. This function is used for calculating the histogram on the joint image

```
def createHistogram(image, mask):
    histogram = []
    for channel in range(image.shape[2]):
        hist = cv2.calcHist([image.astype(np.uint8)], [channel], mask[:,:,channel], [256], [0, 256])
        histogram.append(hist)

    return histogram
```

```
def createJoinedHistogram(image, mask):
    histogram = cv2.calcHist([image.astype(np.uint16)], [0], mask, [768], [0, 768])

    return histogram
```

Project Development - Visualizing Histograms

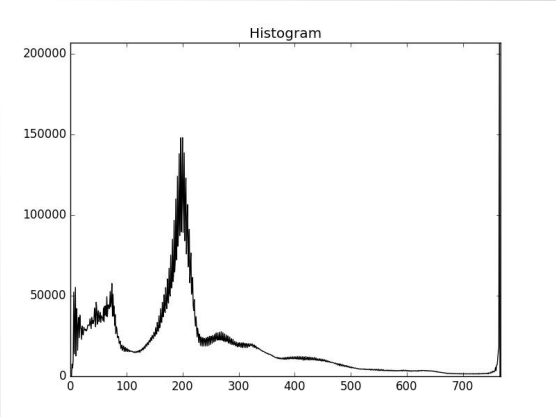
- While the 1D arrays representing the histograms are useful for calculation, it is also helpful to visualize the data. For this, I wrote the drawHistograms() function that takes a list of histograms and displays them on a single chart using PyPlot
  - The function adds a line on the joint plot for each element in the histogram list
  - To better convey the information, the plotted lines are colored black in the case of a single histogram and blue, green, and red otherwise
  - It is critical that any prior plots be cleared before trying to plot any additional information or they may conflict, resulting in nonsensical output

```
def drawHistogram(histograms):

    colors = ('b', 'g', 'r')

    # Clear current plot before making any new ones
    plt.cla()
    plt.clf()

    # B&W vs color images
    if len(histograms) == 1:
        plt.plot(histograms[0], color='k')  # plot the single graph with a black line
    else:
        for index in range(len(histograms)):
            plt.plot(histograms[index], color=colors[index])

    plt.axis('tight')
    plt.title("Histogram")

    return plt
```

Project Development - Collapsing Image Layers

- In adjusting the image brightness, particularly for colored images, it is important to be able to differentiate between areas that are over/underexposed and those that are just a vivid color. For example, simply checking for pixel values above 250 in the original image does not differentiate between a bright red shirt and a bright, white light
- To address this issue, each of the layers of the image is summed together to produce a 2D array of the same height and width as the original but, for a colored image, with a pixel intensity range of [0, 768). This way, if any area of the image is truly over or underexposed, it will be at the extremes of the histogram
    - The implementation of the joinLayers() function is straightforward: an output array of the same height and width of the input image is created and then each layer is summed together into the output
- While this joint image is not displayed directly to the user, the histogram is calculated using the createJoinedHistogram() function and displayed using the drawHistogram() function, both described earlier. A sample graph is shown below

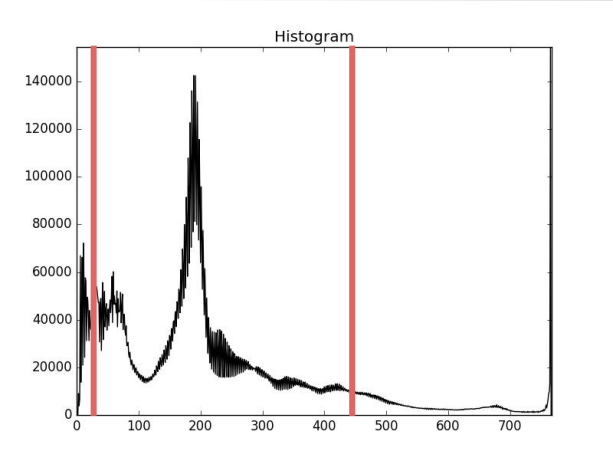
```
def joinLayers(image):

    output = np.zeros((image.shape[0], image.shape[1]), dtype=np.float)

    for channel in range(image.shape[2]):
        output += image[:, :, channel]

    return output
```



Project Development - Finding Appropriate Bounds

- The crux of automating this process lies in finding some appropriate pixel bound for the subsequent rescaling. Here, I leverage the generated joint histogram to calculate the interval by excluding some percentage of pixels from the extremes
  - The threshold percentages were found through empirical testing
- The findRange() function takes in a histogram as well as upper and lower percentage thresholds. It first calculates the total number of pixels in the histogram then iterates from the bottom of the intensity range, calculating the point at which the lower threshold is crossed, to find the lower bound. The upper bound is similarly derived but from the upper extreme of the spectrum
- Sample bounds are drawn for the given joint histogram with a lower threshold of 5% and an upper threshold of 10%

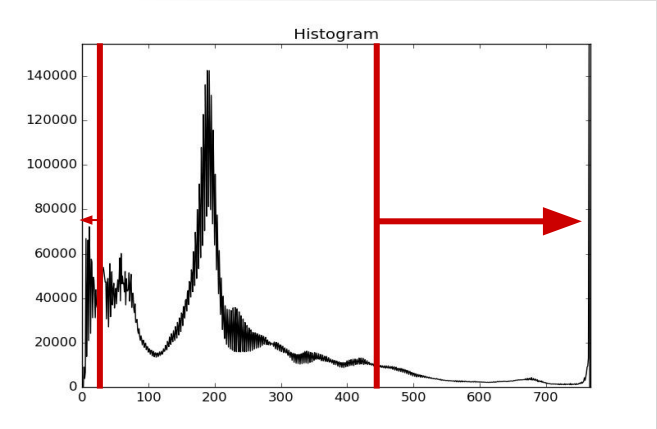
```
def findRange(histogram, lower_threshold, upper_threshold):

    # Calculate total number of pixels in the histogram (if used
    # with joinHistLayers, will count total for each channel)
    total_pixels = np.sum(histogram)

    # Starting from the bottom of the range, 0, find the intensity
    # value for which a threshold percent of pixels are excluded
    total = 0
    i = 0
    while np.sum(histogram[:i]) <= total_pixels * lower_threshold:
        i += 1
    start = i - 1

    # Also find upper bound
    total = 0
    i = histogram.shape[0]
    while np.sum(histogram[i:]) <= total_pixels * upper_threshold:
        i -= 1
    end = i + 1

    return start, end
```



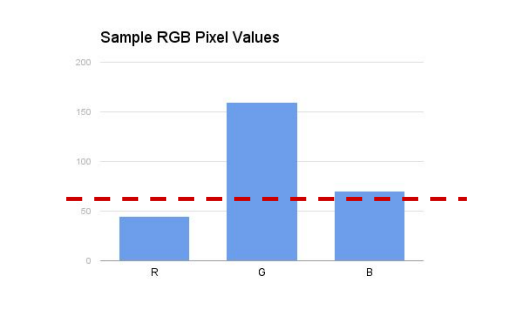
Project Development - Linear Brightness Rescaling

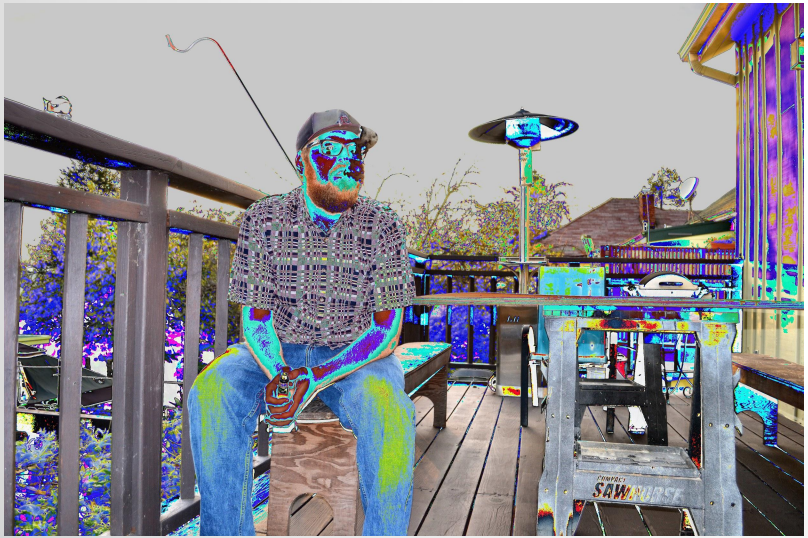
- With the bounds calculated, the image brightness can be rescaled. At a high level, this early implementation of linear rescaling attempts to adjust each color layer independently by rescaling it relative to the joint image
  - The actual implementation of linearRescale() constructs an output array of the same shape as the original image then calculates a scalar scale factor such that the interval found in the prior step can be mapped to the full [0, 255] range. Then every layer is adjusted appropriately, first by subtracting a third of the interval value (as there are 3 channels) then multiplying by the scale factor
  - Finally, the output values are truncated using array masking to fit within the uint8 range
  - Note that the code below is just an example of the structure that was written; the function went through many iterations based on testing feedback
- With this function, the middle range of the histogram, as determined in the prior step, was spread out to encompass the entire range, increasing contrast

```
def linearRescale(image, start, end):

    output = np.zeros(image.shape, dtype=np.float)

    scale = 256 / (end - start)

    for index in range(image.shape[2]):
        output[:,:,index] = (image[:,:,index] - start ) * scale

    lower_bound_mask = output < 0
    output[lower_bound_mask] = 0
    upper_bound_mask = output > 255
    output[upper_bound_mask] = 255

    return output.astype(np.uint8)
```



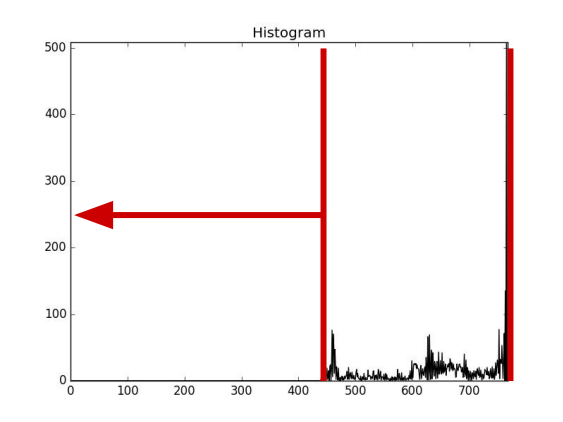
Project Development - Linear Rescaling Challenges





- While this methodology appeared sound, there were many challenges that limited its efficacy
  - On further review, this technique did not address the fundamental issue of saturation as it would only push extreme values further outside the range and in fact ended up shifting some pixels to the extremes from the middle
  - As each layer was handled independently, it was possible to end up with a drastically different color if corresponding pixels in different layers had significantly different values. As a constructed example, the chart shows possible intensity values for a given pixel as well as the start value of the interval (dotted red line). After scaling, some of the pixel colors, R, may fall to zero, giving the output pixel a different hue
    - This issue results in psychedelic artifacts, as seen in the sample output image. Note, however, that the wood grain of the bench is brighter and has more contrast than the original image

Project Development - Linear Rescaling Take Two



- Given the challenges discovered in the original linear rescaling option, the function was rewritten to try to address the issues with color fidelity. Namely, the updated function fixed the ratios of the different color channels based on the input image, thereby preserving color (to the nearest uint8 value)
- This update remedied color issues but still did not address one of the underlying motivations for the project, reducing saturation. As depicted in the joint histogram below, if there is a peak around 255, the interval function will include this extreme but may exclude some of the lower range. In rescaling, the image is effectively darkened, making the subject even harder to see
    - The sample histogram and image show the results of this unintended darkening

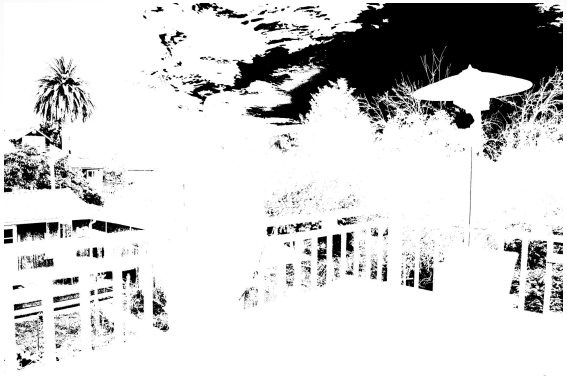
Project Development - Nonlinear Rescaling

```python
def nonlinearRescale(image, joinedimage, start, end, light_mask, dark_mask):

    output = np.zeros(image.shape, dtype=np.float)
    scale = 255.0 / (end - start)

    joined_rescaled = (joinedimage - start) * scale
    for i in range(image.shape[2]):
        output[:,:,i] = joined_rescaled * (image[:,:,i] / joinedimage) / (1.0/3.0)

    adark_mask = output < 15
    output[adark_mask] = output[adark_mask] * 2
    bright_mask = output > 245
    output[bright_mask] = output[bright_mask] * .95

    # Only scale unmasked regions by copying over original image data that was masked
    mask_3Dl = np.full(image.shape, False, dtype=np.uint8)
    mask_3Dd = np.full(image.shape, False, dtype=np.uint8)
    for i in range(image.shape[2]):
        mask_3Dl[:,:, i] = light_mask
        mask_3Dd[:,:, i] = dark_mask
    almask = mask_3Dl == 0
    admask = mask_3Dd == 0
    output[almask] = image[almask] * .98
    output[admask] = image[admask] * 2

    lower_bound_mask = output < 0
    output[lower_bound_mask] = 0
    upper_bound_mask = output > 255
    output[upper_bound_mask] = 255

    return output
```

- In an effort to solve the issue of peaks at the extreme of the pixel intensity ranges, I implemented a mask to cover the extremes and then paste back the masked values from the original image.
- From experimentation, I also found that these masked values, as well as pixels near the extremes from the second attempt at linear rescaling, could be brightened or darkened by multiplying by a fixed scalar as needed
- Similar to the value bounding in earlier versions of linear scaling, this function relies on masking to quickly adjust pixel values across the output image based on a condition

Project Development - Nonlinear Rescaling

Input



Mask

Output

- As seen here and in the demonstration photos, this method generated images that show greater contrast and improved brightness, while maintaining color fidelity, on the image subject. The person on the right looks brighter and more vibrant. The wooden table is also brighter and shows greater grain detail
- Please see the full resolution input and output for a better depiction of the improvement

Project Development - Nonlinear Rescaling Challenges

- This final method, though generating better image clarity of the subject, is not without its own challenges. The most apparent is the artifacting around where the masked areas are copied back into the output. Even with some scalar adjustment, the discontinuities are clearly visible
    - I attempted to smooth the transitions, both in the masked areas as well as the linearly scaled areas that were too bright or dark, by iterating over the pixel values at the extreme and setting a more gradual scale factor but this resulted in long processing times and limited improvement in picture quality
- As written, the model program does not do well with images in which the subject is overexposed or if the overall image already has good range and limited saturation. It may be possible to check for these conditions based on the color histogram and skip such images in batch processing

Ideas for Further Exploration and Improvement

- A tweak to the nonlinearRescale() function to reduce color issues is to lock color ratios for each pixel while executing the scalar adjustments using the masks. This could be accomplished by leveraging the joined image
- Though the subjects in my sample images ended up looking better, the backgrounds suffered. A possible solution is to manually create masks around the subjects and areas of interests and run the algorithm on that
    - The linear and nonlinear scaling may not even be necessary at that stage as it may be feasible to brighten or darken the entire masked area using a fixed scalar
    - Creating masks would entirely defeat the goal of automating this process unless the mask creation itself could also be automated
- There may be a fundamentally different technique to implementing automatic brightness correction by iterating over every pixel intensity and scaling it by some factor, increasing the brightness of the darkest areas and vice versa
    - This method could be time intensive as there are 256 values to iterate over. It may be possible to speed up the process by iterating over small pixel ranges instead
    - It is not clear how this may be automated so that it can handle a range of pictures

Resources

- http://stackoverflow.com/questions/1120707/using-python-to-execute-a-command-on-every-file-in-a-folder
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html
- http://docs.opencv.org/3.1.0/d1/db7/tutorial_py_histogram_begins.html
- http://matplotlib.org/users/image_tutorial.html
- http://www.bogotobogo.com/python/OpenCV_Python/python_opencv3_image_histogram_calcHist.php
- http://www.itsalwaysautumn.com/2014/05/21/easy-fix-dark-underexposed-photos.html
- http://stackoverflow.com/questions/7762948/how-to-convert-an-rgb-image-to-numpy-array
- http://www.ign.com/articles/2015/05/24/mad-max-fury-road-director-george-miller-demanded-black-and-white-version-for-blu-ray
- https://github.com/github/gitignore
- http://stackoverflow.com/questions/1623849/fastest-way-to-zero-out-low-values-in-array

Appendix: Source Code

The source code has been submitted with this report but it can also be found using this link

Credits or Thanks

- I would like to thank my roommates, Devin Whitford and Conor Giambona, for posing for some of the sample pictures