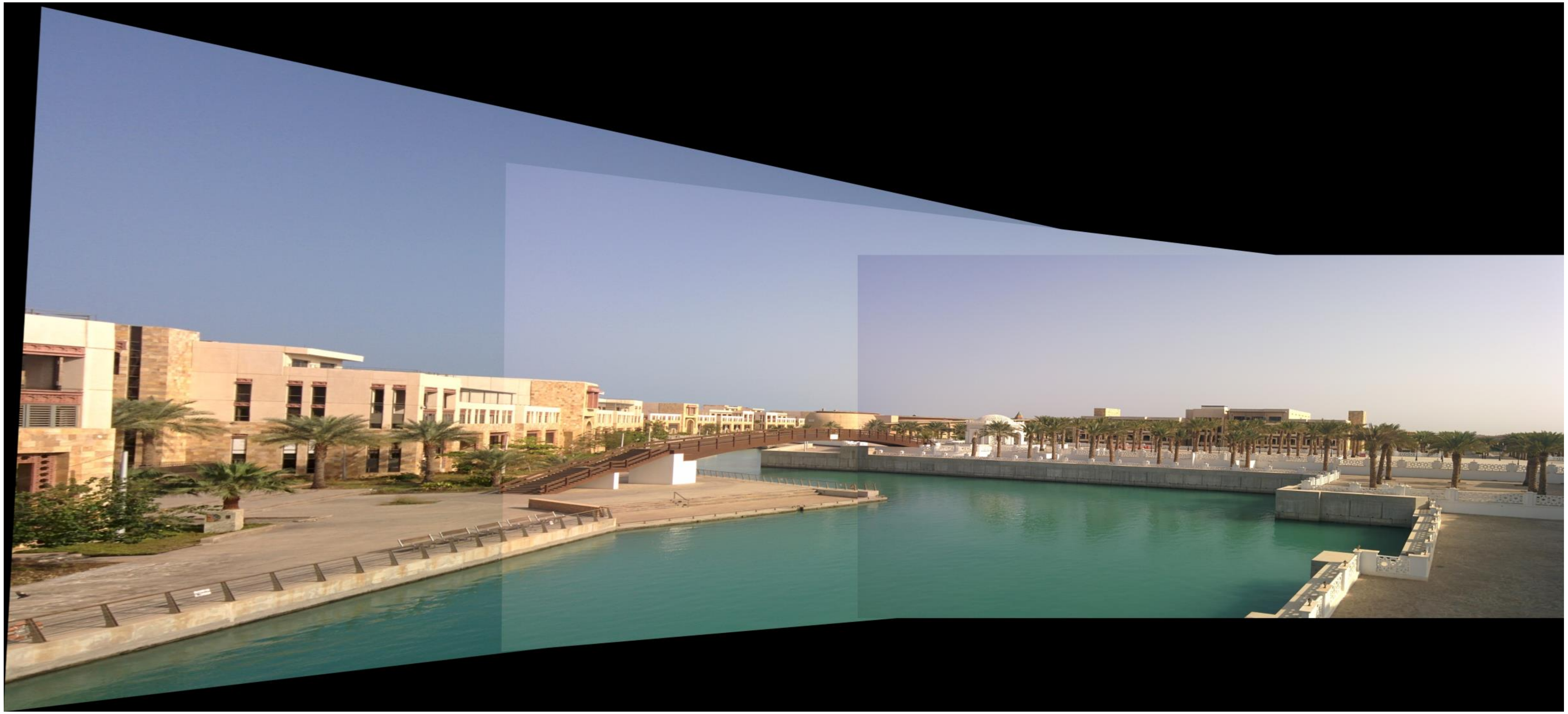
Computational Photography

Assignment #4: Panoramas

Josh Adams

CS6475 - Fall 2019

Results - Test Images



Show the resulting panorama from the images we provided. **Do NOT crop your result!** - This is important.
You may add additional slides if you want to show cropped results

Setup and Panorama Type

- Describe the setup for your camera. In addition, make sure to include camera parameters that you think are relevant (e.g. ISO, etc.)
    - *Sony α5100*
    - *ISO -100*
    - *Exposure 1/60*
    - *f/3.5*

- Describe the camera movement that you used when you took your pictures.
    - *For taking the pictures I panned from left to right slowly and took a picture when I felt there was enough coverage to produce a good panorama.*

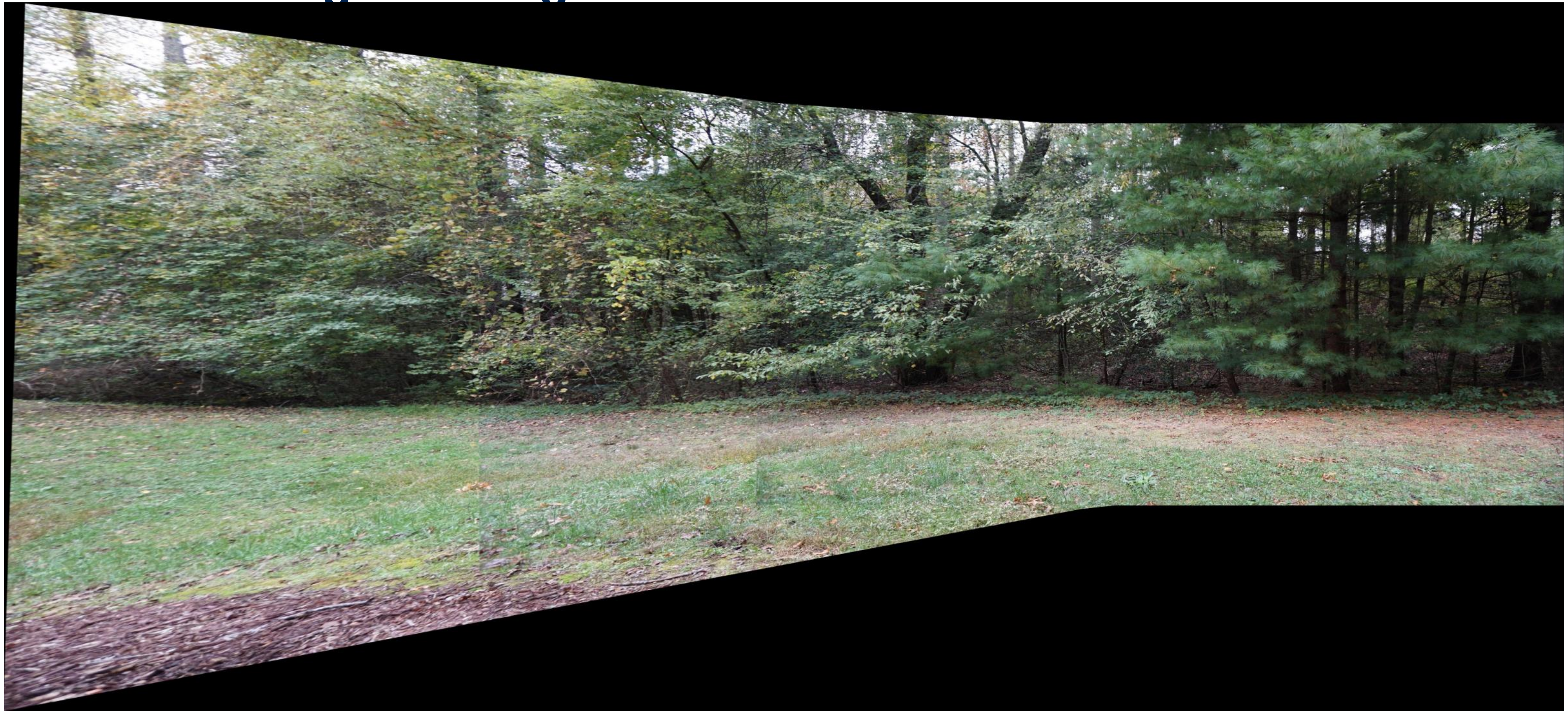
- What kind of projection does your panorama use for the combined image canvas?
    - *Refer to lesson 5-03 and the Szeliski text in Section 9.1 for more information.*
    - *According to the lesion 5-03, my panorama uses 2D Re-Projection. The reason is that we are relating two images using a shared pixel to align the two images. But could be argued that this is just a simple planar projection.*

Original Input Images

Show the input images used for **<u>your own</u>** panorama. These should be images that YOU took. Remember, a minimum of 3 images is required.

- These are the three images I took that I will combine into a panorama.
- They were obtained by starting from the left-hand side and panning to the right and taking a new image
- These images are just of the tree lined area behind my apartment

Results - Original Images



Show the resulting panorama from the images that you took. **Do NOT crop your result!** - This is important.
You may add additional slides if you want to show cropped results

warpCanvas()

- Why multiply x_min and y_min by -1 in the warping function?

$$\begin{bmatrix} X' \\ Y' \\ W' \end{bmatrix} = \mathbf{M} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

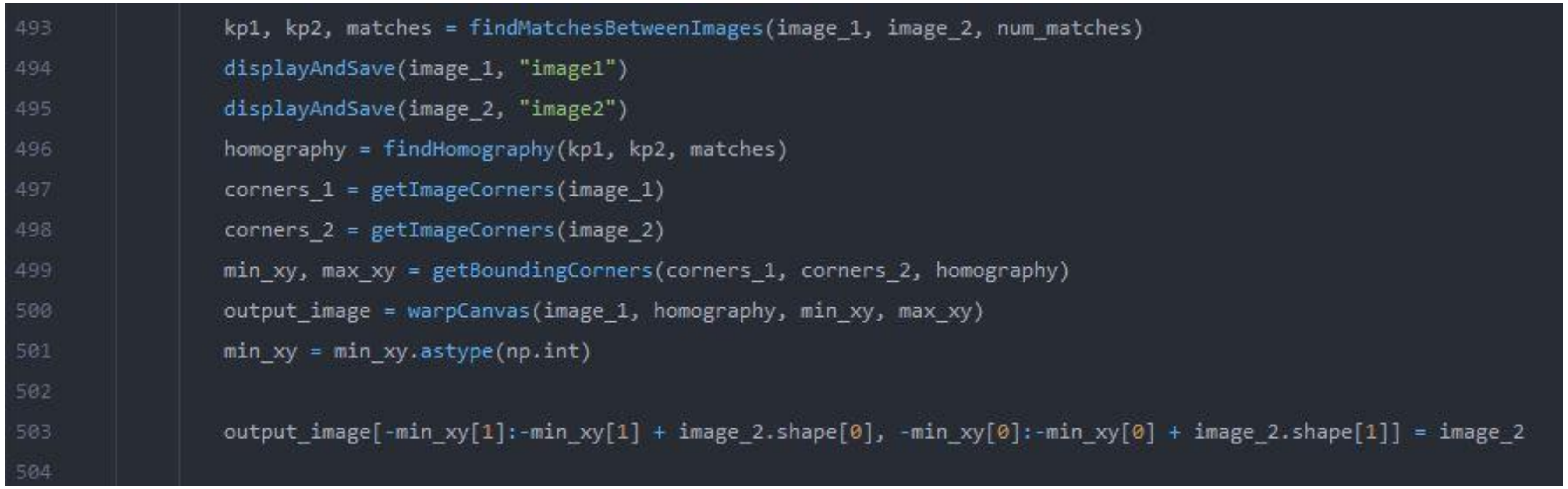
$$\mathbf{M} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} X + t_x \\ Y + t_y \\ 1 \end{bmatrix} = \begin{bmatrix} X' \\ Y' \\ W' \end{bmatrix}$$

In the warping function we are using x_min and y_min. Those coordinates were obtained in the ‘getBoundingCorners’ function. As a result they are going to be negative because the two images are overlapping. If you looked at the min x value for one image you are not able to go past the edges of that image. But we are overlaying a smaller image on top of a larger one. This allows us to go past that boundary and results in getting negative numbers. If we did not multiple Tx and Ty by negative one, our images would try to translate in the wrong direction. This will cause many artifacts, typically not expected. (example of this is one of my failures at the end)

blendImagePair()

- Discuss your implementation for **blendImagePair()**.

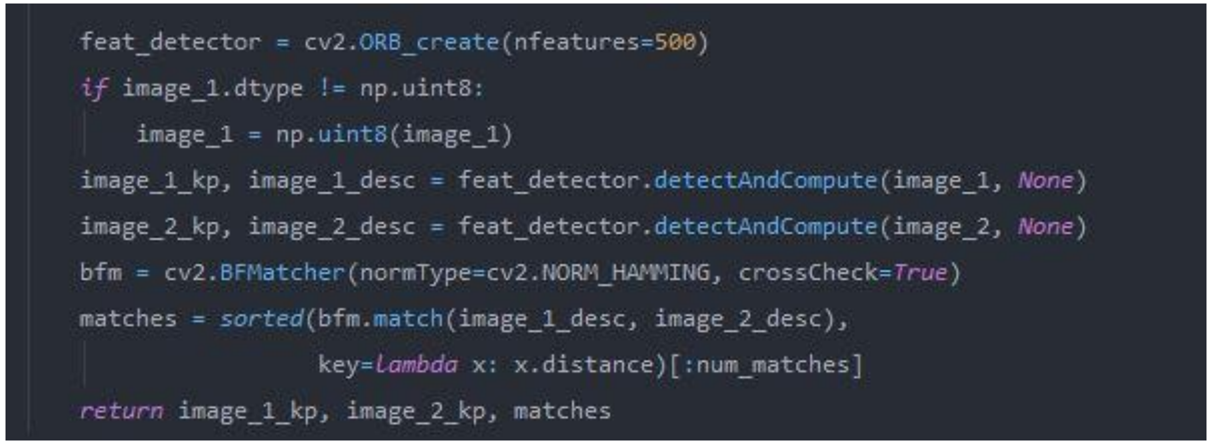


```
493         kp1, kp2, matches = findMatchesBetweenImages(image_1, image_2, num_matches)
494         displayAndSave(image_1, "image1")
495         displayAndSave(image_2, "image2")
496         homography = findHomography(kp1, kp2, matches)
497         corners_1 = getImageCorners(image_1)
498         corners_2 = getImageCorners(image_2)
499         min_xy, max_xy = getBoundingCorners(corners_1, corners_2, homography)
500         output_image = warpCanvas(image_1, homography, min_xy, max_xy)
501         min_xy = min_xy.astype(np.int)
502
503         output_image[-min_xy[1]:-min_xy[1] + image_2.shape[0], -min_xy[0]:-min_xy[0] + image_2.shape[1]] = image_2
504
```

- The first thing I do is get the key points for both images and the matches between the two – line 493
- Finding the homography matrix using those key points and matches – line 496
- Getting the corners for the two images – lines 497,498
- Find bounding corners of the two images – line 499
- Warp the image using the min_xy and homorgraphy matrix – line 500
- Insert image two into the slice defined for warped image – line 503

findMatchesBetweenImages(image_1, image_2, num_matches)

- Discuss your implementation for **blendImagePair()**.

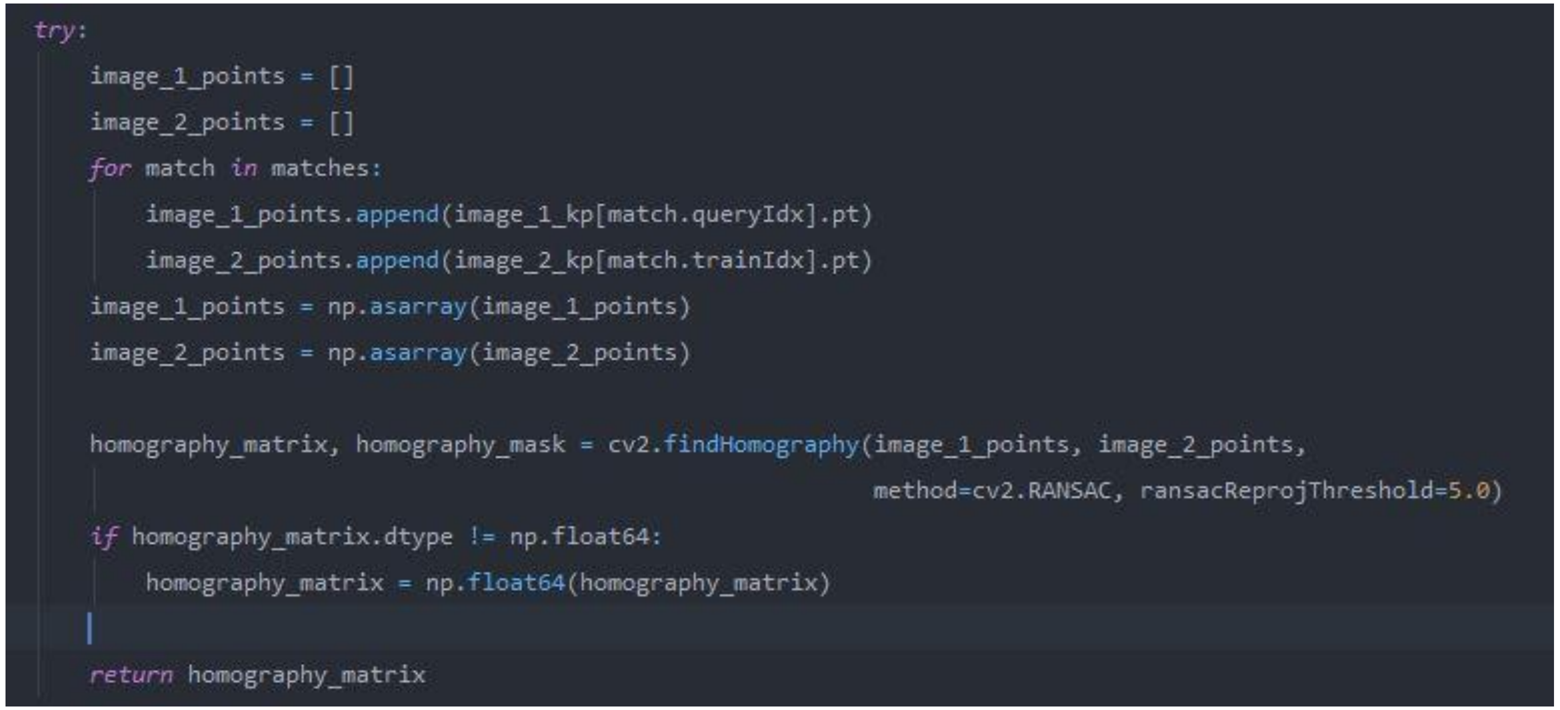


```
feat_detector = cv2.ORB_create(nfeatures=500)
if image_1.dtype != np.uint8:
    image_1 = np.uint8(image_1)
image_1_kp, image_1_desc = feat_detector.detectAndCompute(image_1, None)
image_2_kp, image_2_desc = feat_detector.detectAndCompute(image_2, None)
bfm = cv2.BFMatcher(normType=cv2.NORM_HAMMING, crossCheck=True)
matches = sorted(bfm.match(image_1_desc, image_2_desc),
                 key=lambda x: x.distance)[:num_matches]
return image_1_kp, image_2_kp, matches
```

- Find matches between images is called with in blend image pair.
- This uses OpenCV’s feature detectors to find the important features of the two images.
- We then use a brute force matcher on the returned key points and sort them by their distances

findHomography(image_1_kp, image_2_kp, matches)

- Discuss your implementation for **blendImagePair()**.

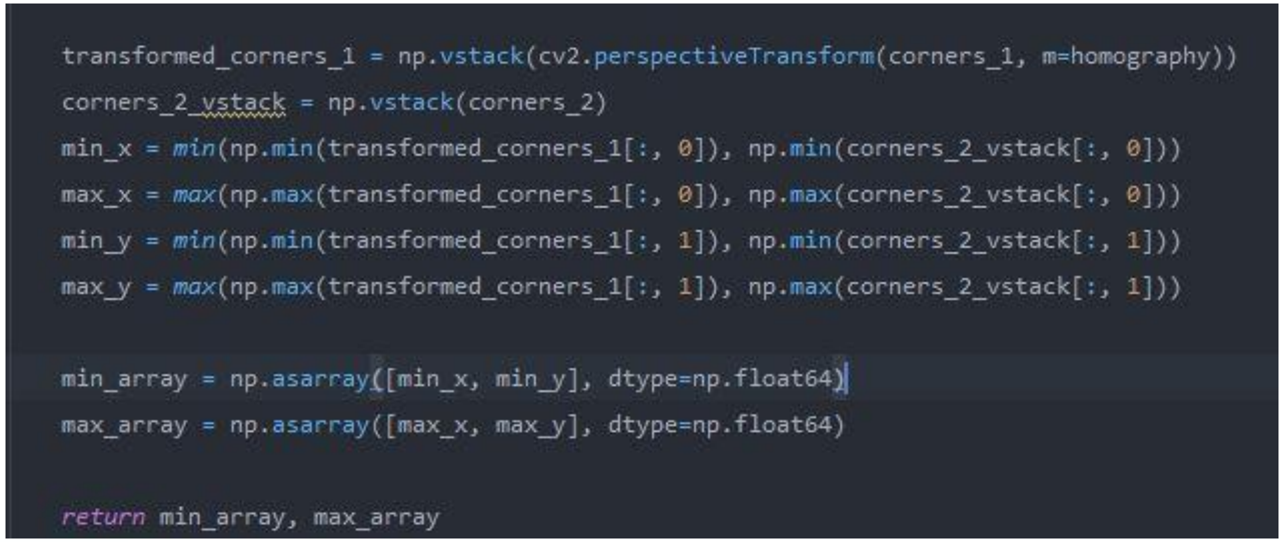


```
try:
    image_1_points = []
    image_2_points = []
    for match in matches:
        image_1_points.append(image_1_kp[match.queryIdx].pt)
        image_2_points.append(image_2_kp[match.trainIdx].pt)
    image_1_points = np.asarray(image_1_points)
    image_2_points = np.asarray(image_2_points)

    homography_matrix, homography_mask = cv2.findHomography(image_1_points, image_2_points,
                                                            method=cv2.RANSAC, ransacReprojThreshold=5.0)
    if homography_matrix.dtype != np.float64:
        homography_matrix = np.float64(homography_matrix)

    return homography_matrix
```

- Find Homography is called with in blend image pair.
- We iterate over the matches passed in and append the points to a list for each image
- Then using OpenCV findHomography function we get the homography matrix.

getBoundingCorners(corners_1, corners_2, homography)

- Discuss your implementation for **blendImagePair()**.

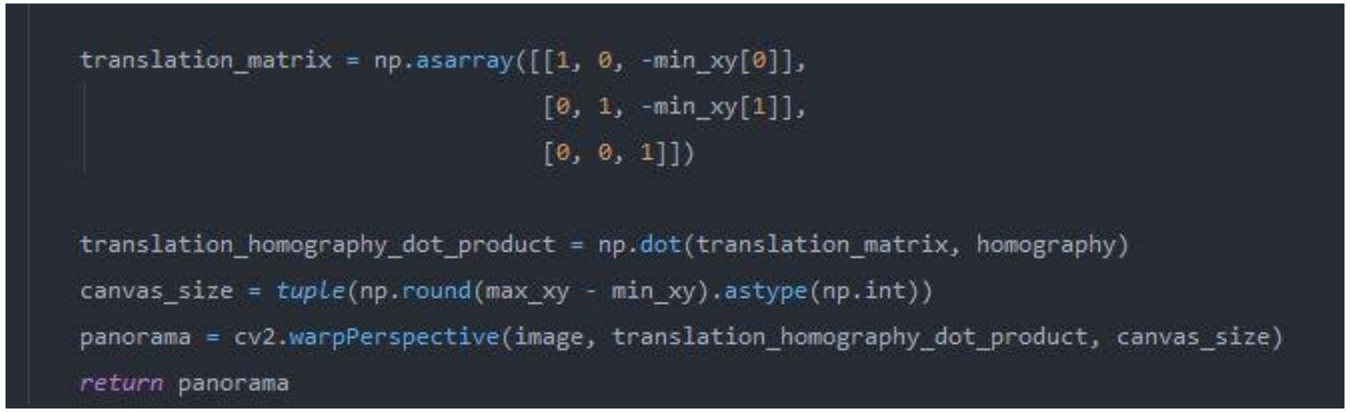


```
transformed_corners_1 = np.vstack(cv2.perspectiveTransform(corners_1, m=homography))
corners_2_vstack = np.vstack(corners_2)
min_x = min(np.min(transformed_corners_1[:, 0]), np.min(corners_2_vstack[:, 0]))
max_x = max(np.max(transformed_corners_1[:, 0]), np.max(corners_2_vstack[:, 0]))
min_y = min(np.min(transformed_corners_1[:, 1]), np.min(corners_2_vstack[:, 1]))
max_y = max(np.max(transformed_corners_1[:, 1]), np.max(corners_2_vstack[:, 1]))

min_array = np.asarray([min_x, min_y], dtype=np.float64)
max_array = np.asarray([max_x, max_y], dtype=np.float64)

return min_array, max_array
```

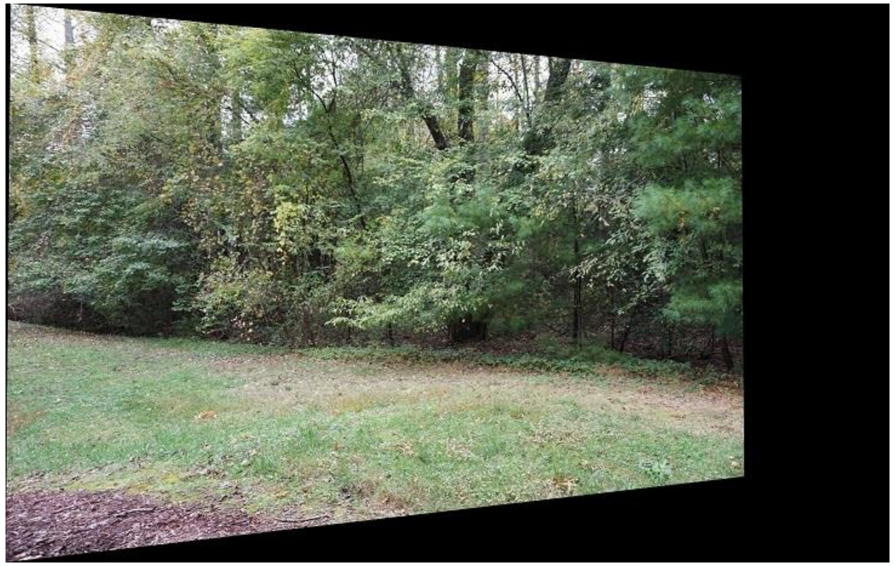
- GetBoundingCorners is called with in blendImagePair.
- I use the corners from image 1 and the homography matrix to generate a perspective transformation
- I then convert it to a vertical stack because it is more intuitive for me
- Obtaining the min_xy and max_xy using various applications of either np.min or np.max

warpPerspective(image, homography, min_xy, max_xy)

- Discuss your implementation for **blendImagePair()**.



```
translation_matrix = np.asarray([[1, 0, -min_xy[0]],
                                 [0, 1, -min_xy[1]],
                                 [0, 0, 1]])

translation_homography_dot_product = np.dot(translation_matrix, homography)
canvas_size = tuple(np.round(max_xy - min_xy).astype(np.int))
panorama = cv2.warpPerspective(image, translation_homography_dot_product, canvas_size)
return panorama
```



- WarpPerspective is called with in blendImagePair.
- I establish the translation matrix, explicitly using –Tx and –Ty
- Using Numpy I get the dot product between the homography matrix and the translation matrix
- I warp the perspective using warp perspective
- This is what the image looked like after warping the perspective

Discussion & Analysis

REMEMBER: We want to see **THOROUGH ANSWERS** for each of these questions! Include any images/drawings (as needed) to help in your discussion & analysis. You should use additional slides for these questions. Again, we expect **graduate-level answers** in the reports.

- Consider your blend function and answer the following questions - What worked well? What did not work well? Were there any problems you could not solve? If you had more time, what would you do to improve upon your existing blend implementation?
    - *In my blending function I attempted many different methods, and some worked better than others. When I worked with an alpha blend, I was not able to completely remove a small black outline where the two images intersect.*
    - *What worked well was being able to utilize many of numpy and OpenCVs functionality which drastically sped up computation.*
    - *If I had more time, I would definitely investigate more color mapping such as neighborhood mapping. Where you find like areas in an image map their corresponding values for their different channels. You then apply that mapping to the newly inserted image. So color match from image to image and can significantly reduce the visual appeared of cropping or inserting.*

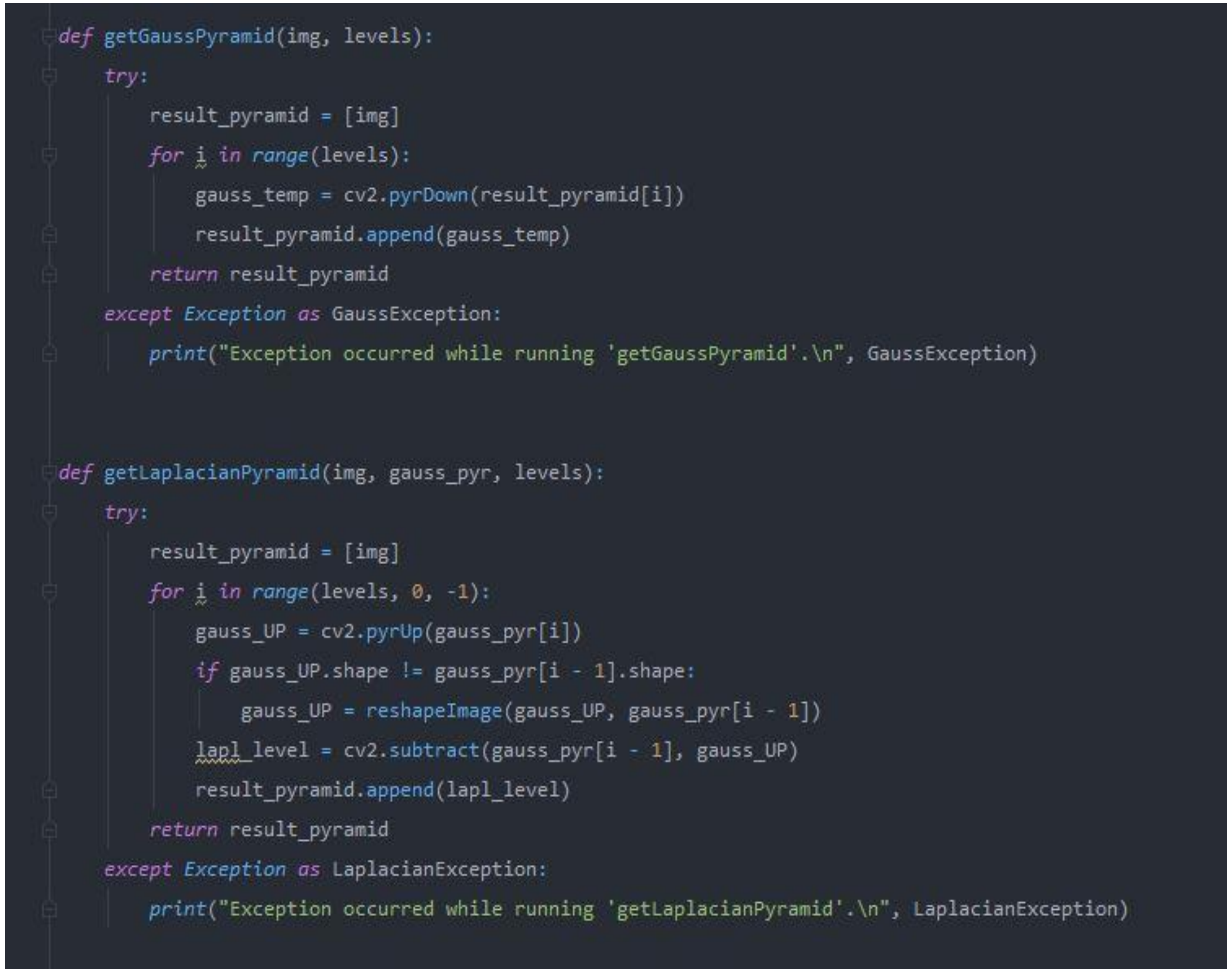
Discussion & Analysis

<u>REMEMBER</u>: We want to see **THOROUGH ANSWERS** for each of these questions! Include any images/drawings (as needed) to help in your discussion & analysis. You should use additional slides for these questions. Again, we expect **<u>graduate-level answers</u>** in the reports.

- If you started the project over again, what could you do differently and how would you go about doing it? Do NOT say there is nothing you could do differently. There's always a way to improve.
    - *If I could start this project over, I would have tried the panorama code on images I personally took much sooner. I pretty much used the test images for almost all my testing. I found out very close to the deadline how, even if the code worked well on the testing images it did not produce good results in self-taken images. I had to troubleshoot and find the issues relatively quickly which resulted in lesser quality results in the test images but much better in the self-taken panorama.*
    - *I would also have spent more time thinking/planning out the process instead of just coding and making changes on the fly. I found myself stumbling over code I had written or having difficulty because of something I changed somewhere. If I had really thought out the process and planned things, I believe I would have alleviated the wasted time debugging and possibly produced a better panorama as a result.*

Above & Beyond (optional: see the 90% rule on Piazza)

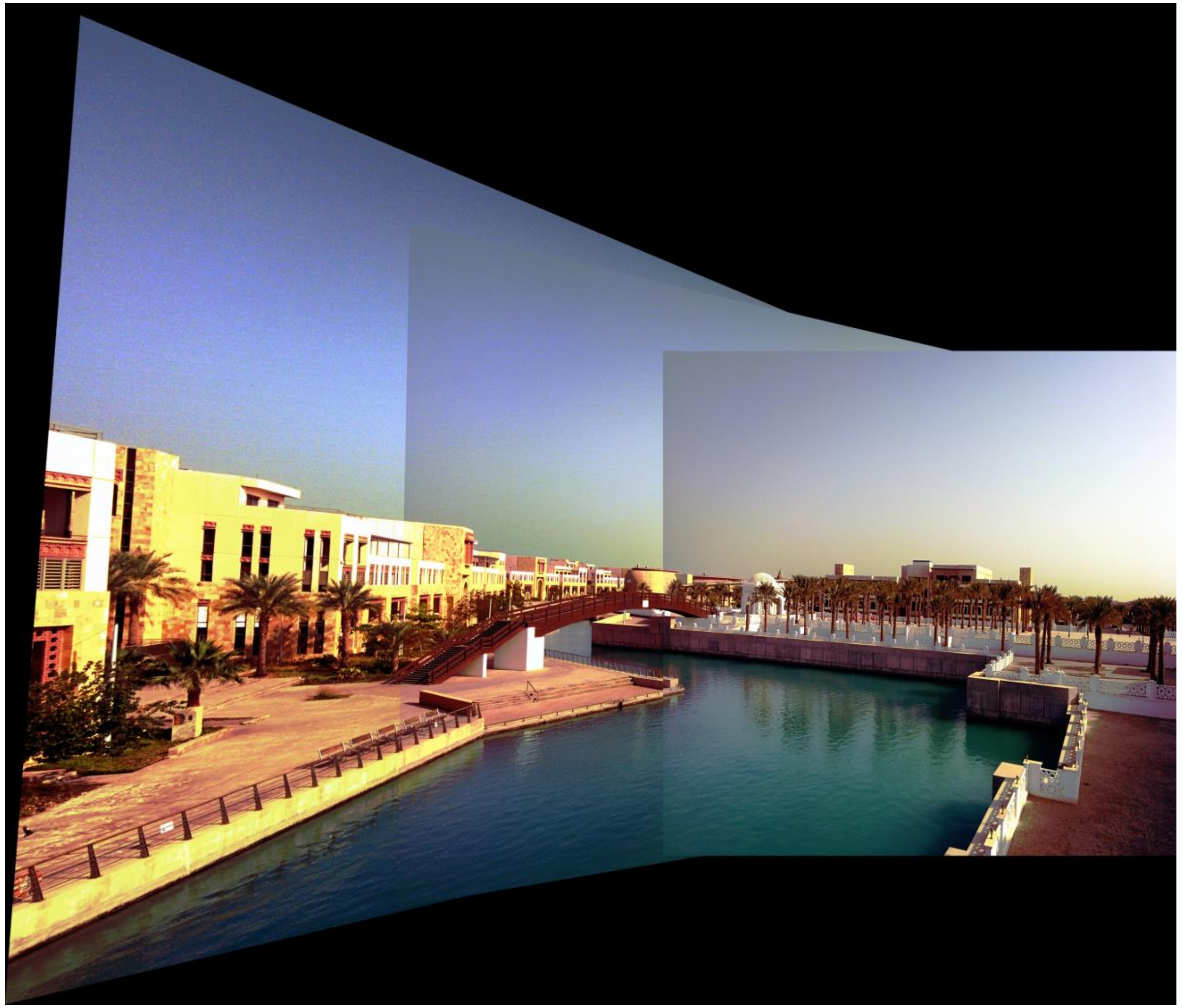
- **PyramidBlending.**
- For Pyramid blending I was not able to get even reasonable results.
- For each layer I was going to have in my pyramid I found the matching points in the two images
- I then generated a mask for those two images.
- Generated the Laplacian and the Gaussian pyramids for both images and the mask.
- The mask was use for when reconstructing to have the correct pixel from the different layers.
- The code where I applied the generated Laplacian and gaussian was lengthy, so I was not able to just add an image.



```python
def getGaussPyramid(img, levels):
    try:
        result_pyramid = [img]
        for i in range(levels):
            gauss_temp = cv2.pyrDown(result_pyramid[i])
            result_pyramid.append(gauss_temp)
        return result_pyramid
    except Exception as GaussException:
        print("Exception occurred while running 'getGaussPyramid'.\n", GaussException)


def getLaplacianPyramid(img, gauss_pyr, levels):
    try:
        result_pyramid = [img]
        for i in range(levels, 0, -1):
            gauss_UP = cv2.pyrUp(gauss_pyr[i])
            if gauss_UP.shape != gauss_pyr[i - 1].shape:
                gauss_UP = reshapeImage(gauss_UP, gauss_pyr[i - 1])
            lapl_level = cv2.subtract(gauss_pyr[i - 1], gauss_UP)
            result_pyramid.append(lapl_level)
        return result_pyramid
    except Exception as LaplacianException:
        print("Exception occurred while running 'getLaplacianPyramid'.\n", LaplacianException)
```

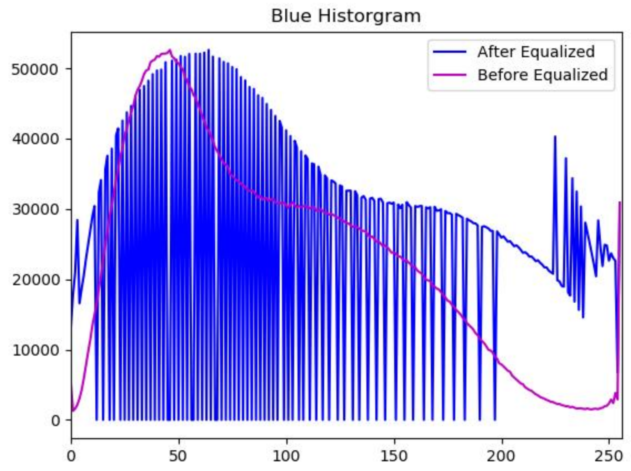
Above & Beyond (optional: see the 90% rule on Piazza)

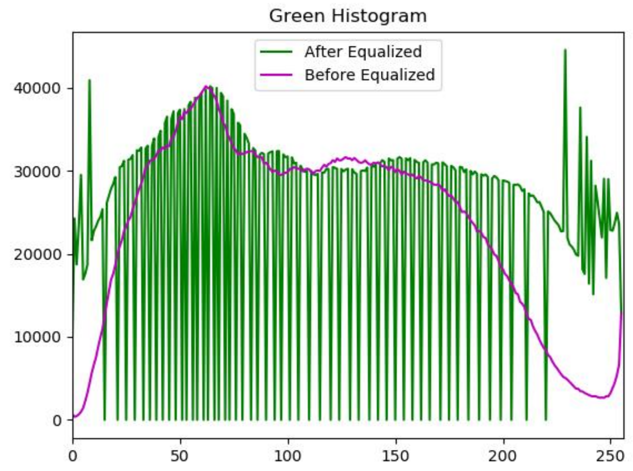
- **Histogram Equalizing**
- For Histogram equalizing the results for my own images worked much better than with the testing images.
- How I did this was splitting the image into its 3 channels and then running cv2.equalizeHist() on those respective channels and then merging back the image.
- The results of *this DID* make the images have more consistent colors, overall. But the downside was the resulting images did not retain their original color
- This method was also one of the slowest but if I had more time, I know I could have sped up many of the operations
- You can see how the colors do not match the original but seem to now all have similar color palettes.

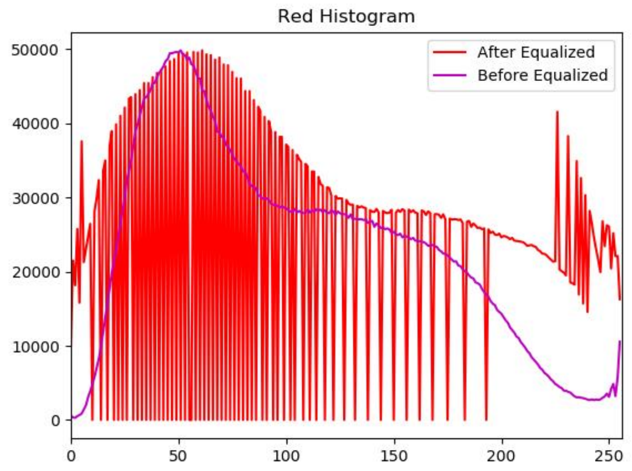


Above & Beyond (optional: see the 90% rule on Piazza)

- **Histogram Equalizing continued.**
- For Histogram equalizing the results for my own images worked much better than with the testing images.
- Here are some of the impacts that histogram equalizing has on the different channels.
- The magenta line is what the histograms looked like prior to the equalizing. They may look chaotic, but the drastic changes overall are what allows the histogram to be smoothed

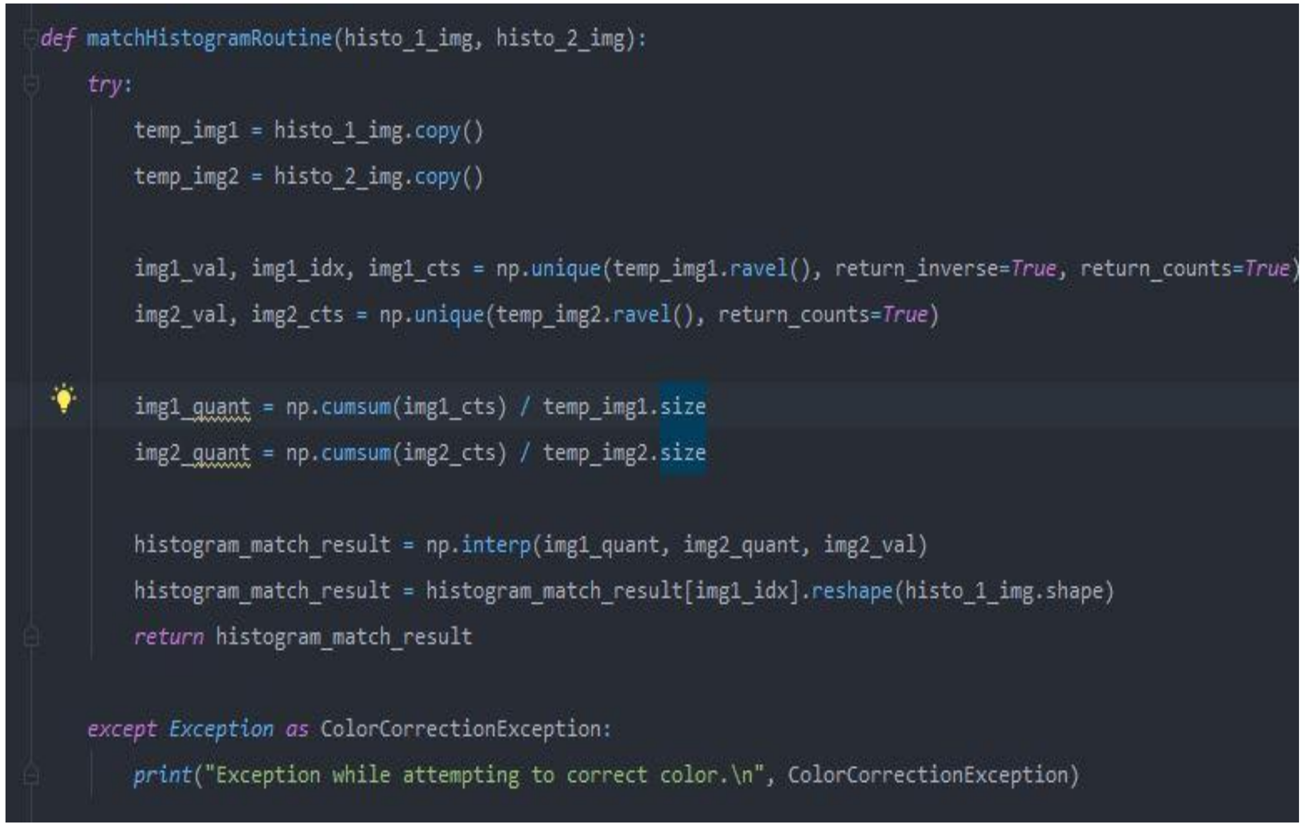






Above & Beyond (optional: see the 90% rule on Piazza)

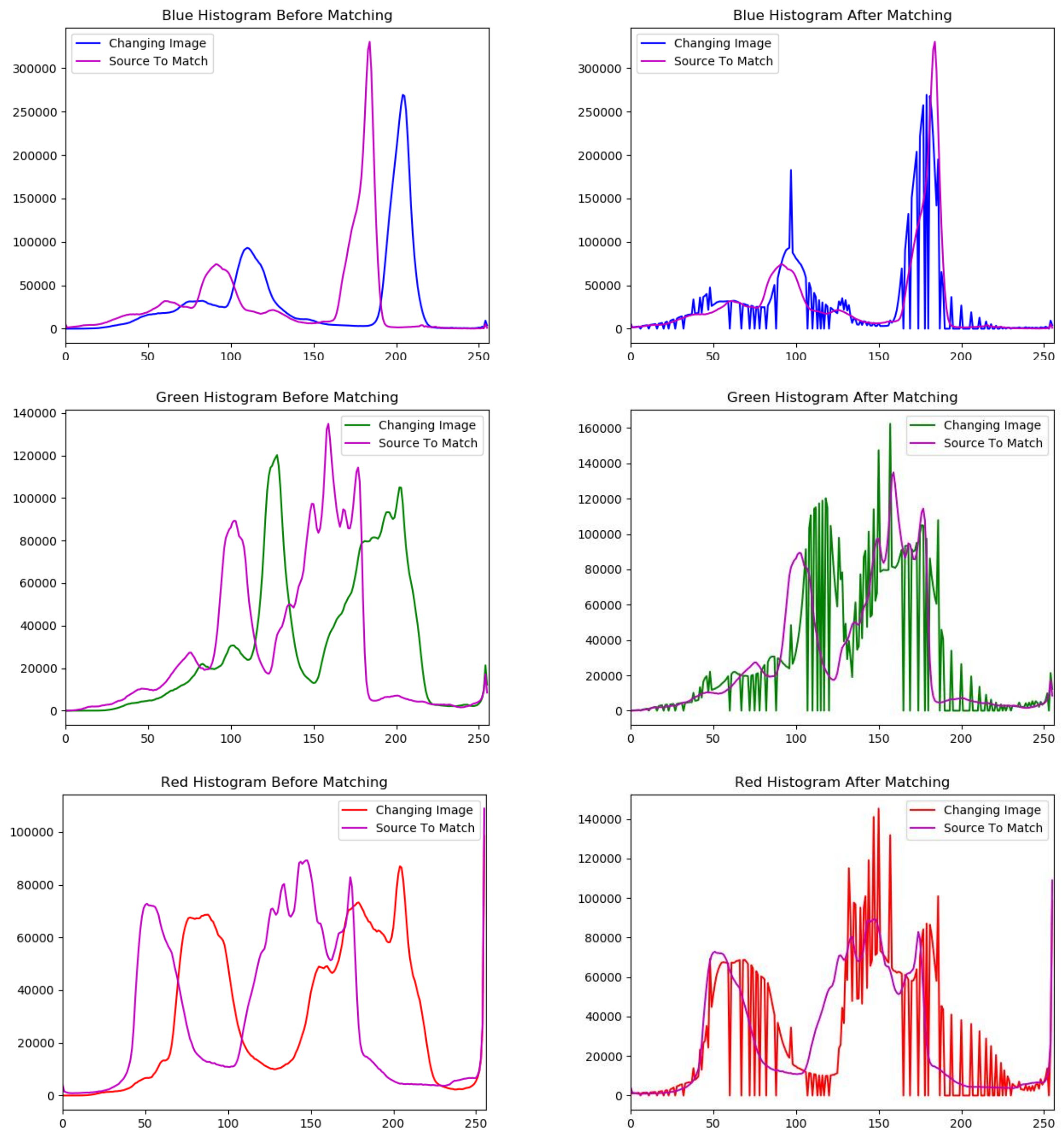
- **Histogram Matching**
- Histogram matching is like histogram equalizing. But instead of just using the OpenCV equalize histogram we have much more finite control.
- How this works is like a lookup table where we are placing the specific colors found in our target image ‘histo_2_image’. We then apply the color transformation to the image we want to change ‘histo_1_img’.
- We can specify how we are wanting to bin the different values found in the images.
- I found this to be the best method when blending my images together as it made subtle changes that were easily controlled.



```python
def matchHistogramRoutine(histo_1_img, histo_2_img):
    try:
        temp_img1 = histo_1_img.copy()
        temp_img2 = histo_2_img.copy()

        img1_val, img1_idx, img1_cts = np.unique(temp_img1.ravel(), return_inverse=True, return_counts=True)
        img2_val, img2_cts = np.unique(temp_img2.ravel(), return_counts=True)

        img1_quant = np.cumsum(img1_cts) / temp_img1.size
        img2_quant = np.cumsum(img2_cts) / temp_img2.size

        histogram_match_result = np.interp(img1_quant, img2_quant, img2_val)
        histogram_match_result = histogram_match_result[img1_idx].reshape(histo_1_img.shape)
        return histogram_match_result

    except Exception as ColorCorrectionException:
        print("Exception while attempting to correct color.\n", ColorCorrectionException)
```

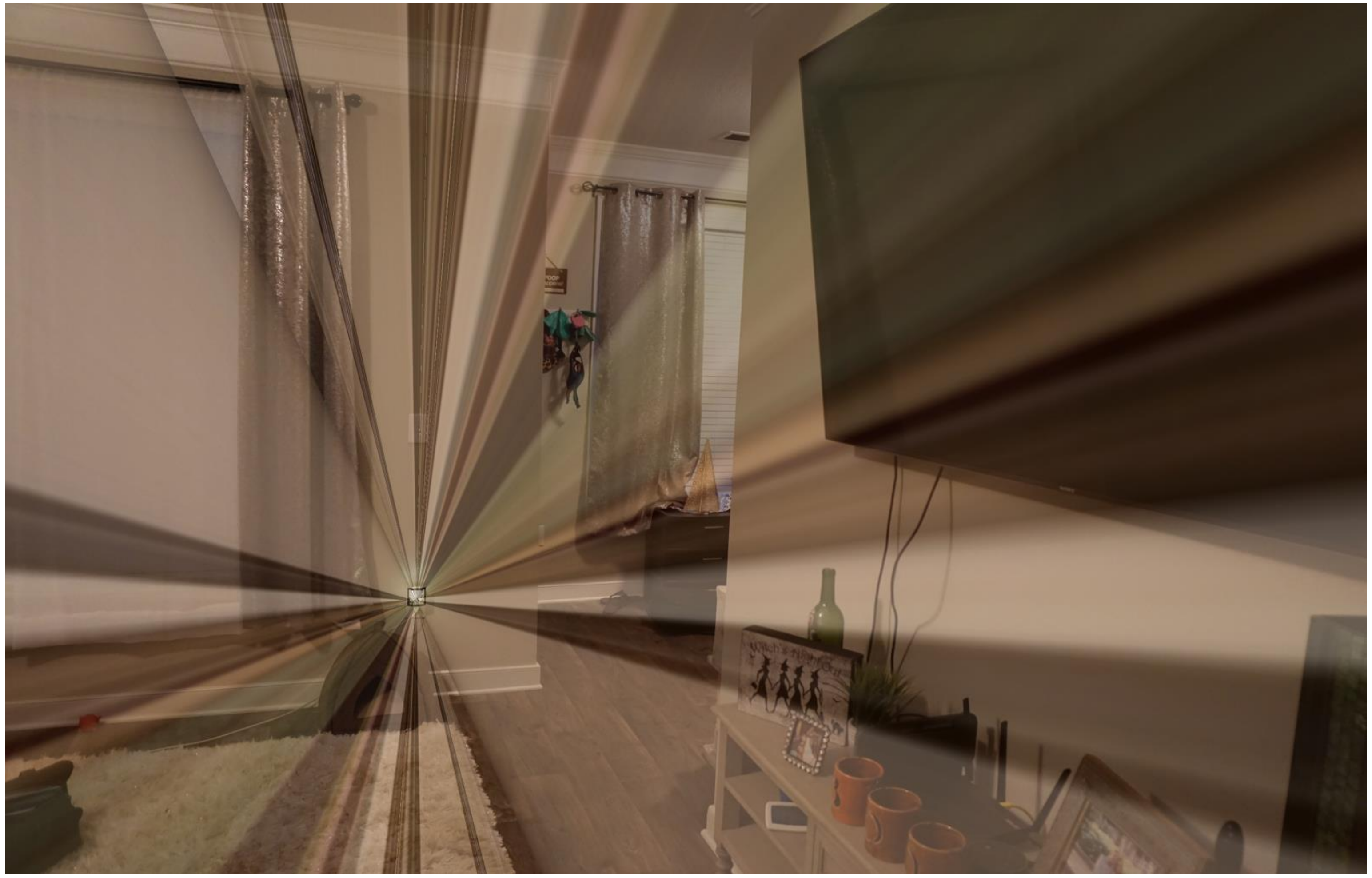
Above & Beyond (optional: see the 90% rule on Piazza)

- **Histogram Matching continued**
- Histogram matching is like histogram equalizing. But instead of just using the OpenCV equalize histogram we have much more finite control.
- On the left are the histograms for the image we want to change and the image we want to match, in their respective color channel.
- For Green (the second row of histograms) you can see that the two images have different histograms at first. After we apply the histogram matching the resulting histogram for the green channels are more similar.
- The histograms will never be identical unless you have very similar images or somehow managed to have images with the same number of each pixel value.



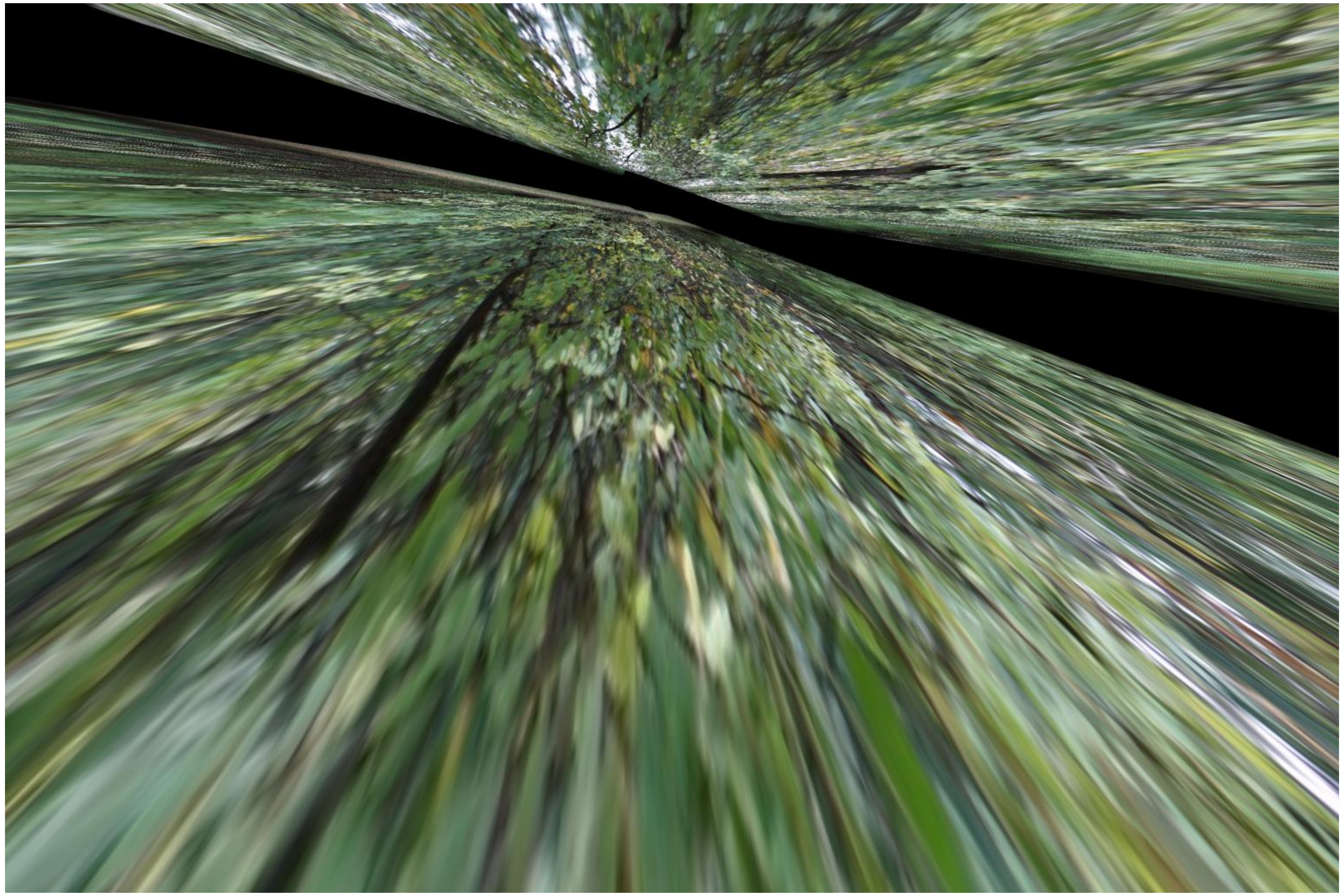
Above & Beyond (optional: see the 90% rule on Piazza)

- Failure Example
- I am not sure what caused this to happen, but I did think it was a cool looking failure.
- The point that all those rays seem to be converging is a night light.



Above & Beyond (optional: see the 90% rule on Piazza)

- Failure Example
- I am not sure what caused this to happen, but I did think it was a cool looking failure.
- I thought this failure was also very cool because it almost seemed to give a 3-dimensional effect to the image. As though the green from the images were walls.



Resources

Record your sources here. We accept all reasonable formats that would allow us to verify your sources.

- OpenCV-Python Tutorials Documentation – Alexander Mordvintsev & Abid K
- Piazza Posts – Alpha Blending post, Corners Post, PyrUp and PyrDown Posts
- Computer Vision: Algorithms and Applications – Richard Szeliski
- https://note.nkmk.me/en/python-opencv-numpy-alpha-blend-mask/ - Alpha blending and masking of images with Python, OpenCV, NumPy – by Nkmk (not completely sure if that is who wrote the information
- https://scipy-lectures.org/advanced/index.html – Image manipulation and processing using Numpy and Scipy
- Pyimagesearch.com Tutorial on Histogram matching – Histogram Matching
- https://vzaguskin.github.io/histmatching1/ Histogram matching in python by Victor Zaguskin
- Udacity video for the course CS6475 Computational Photography

Remember: It is plagiarism if you don't reference your sources!