You will be graded primarily based on your report. A demonstration of understanding of the concepts involved in the project are required show the output produced by your code.

Include visualizations of the output of each stage in your pipeline (as shown in the system diagram on page 2), and a description of what you did for each step. Assume that we're familiar with the project, so you don't need to spend time repeating what's already in the course notes. Instead, focus on any interesting problems you encountered and/or solutions you implemented.

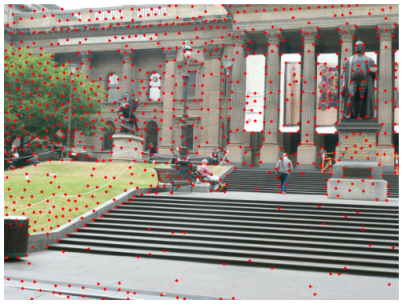
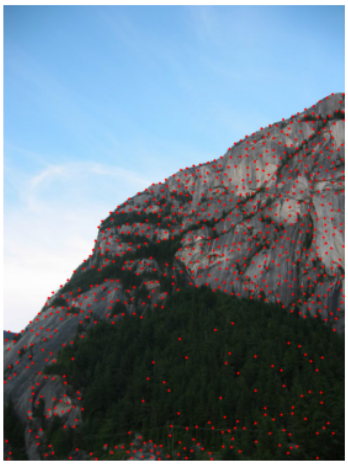
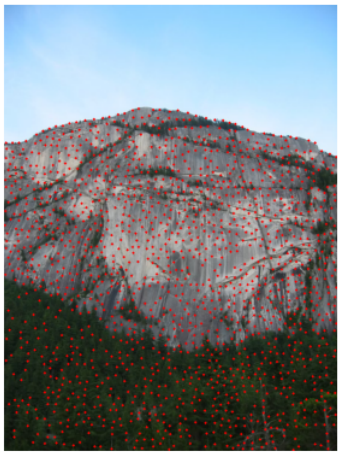
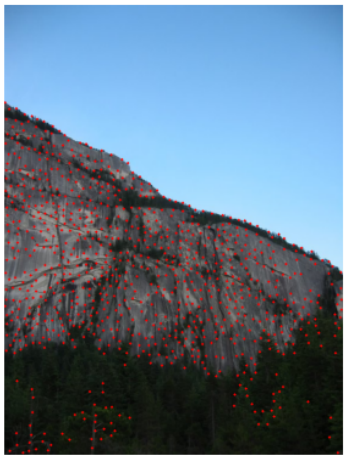
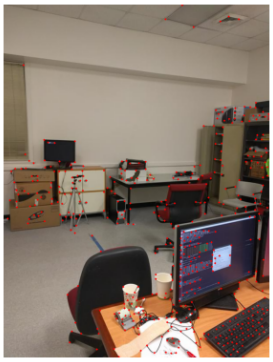
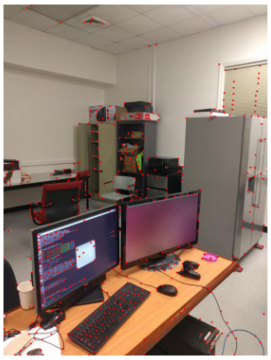
Be sure to include the output panoramas for **all three image sets (from the trainingsets)**. Because you have limited time in which to access the "test set" images, we won't expect in-depth analysis of your results for them.

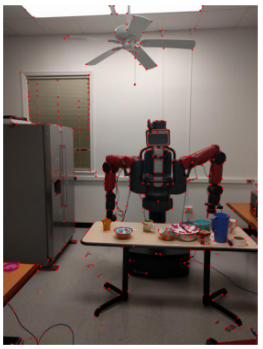
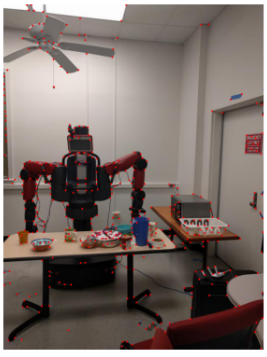
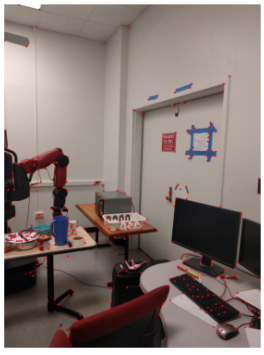
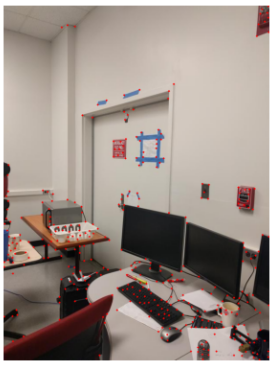
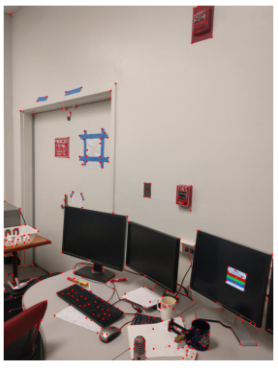
As usual, your report must be full English sentences, **not** commented code. There is a word limit of 1500 words and no minimum length requirement.

TEST SET will be available 4 days before deadline: TBD on Piazza

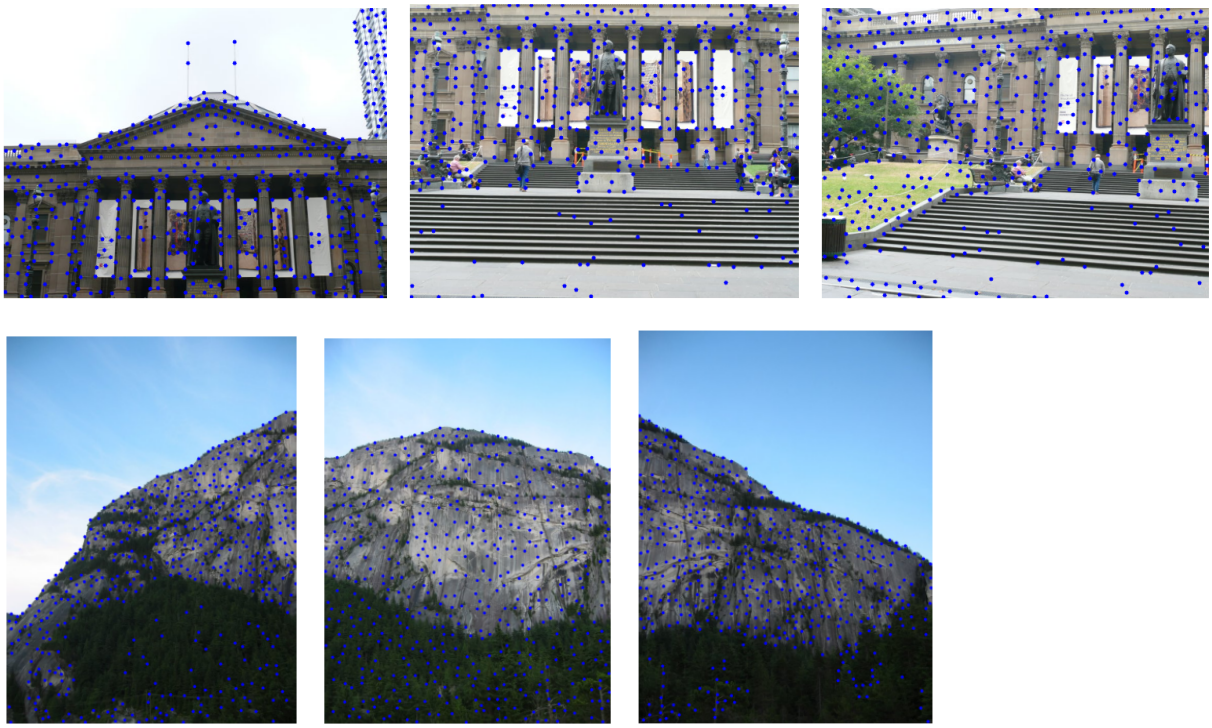
For the detect_corner method we first read up on the OpenCV wiki at OpenCV: Shi-Tomasi Corner Detector & Good Features to Track and OpenCV: Harris Corner Detection and noticed the differences between the two, especially how they handle the scoring function. We then compared the outputs between the two and decided to use goodFeaturesToTrack which uses Shi-Tomasi corner detection. To start off we convert to grayscale, apply goodFeaturesToTrack with 10000, an arbitrary number chosen after testing with 100, 500, 1000, 5000, and 10000. There were barely any differences between 5000 and 10000. Then we basically map each corner back to the cmap. The results are below:

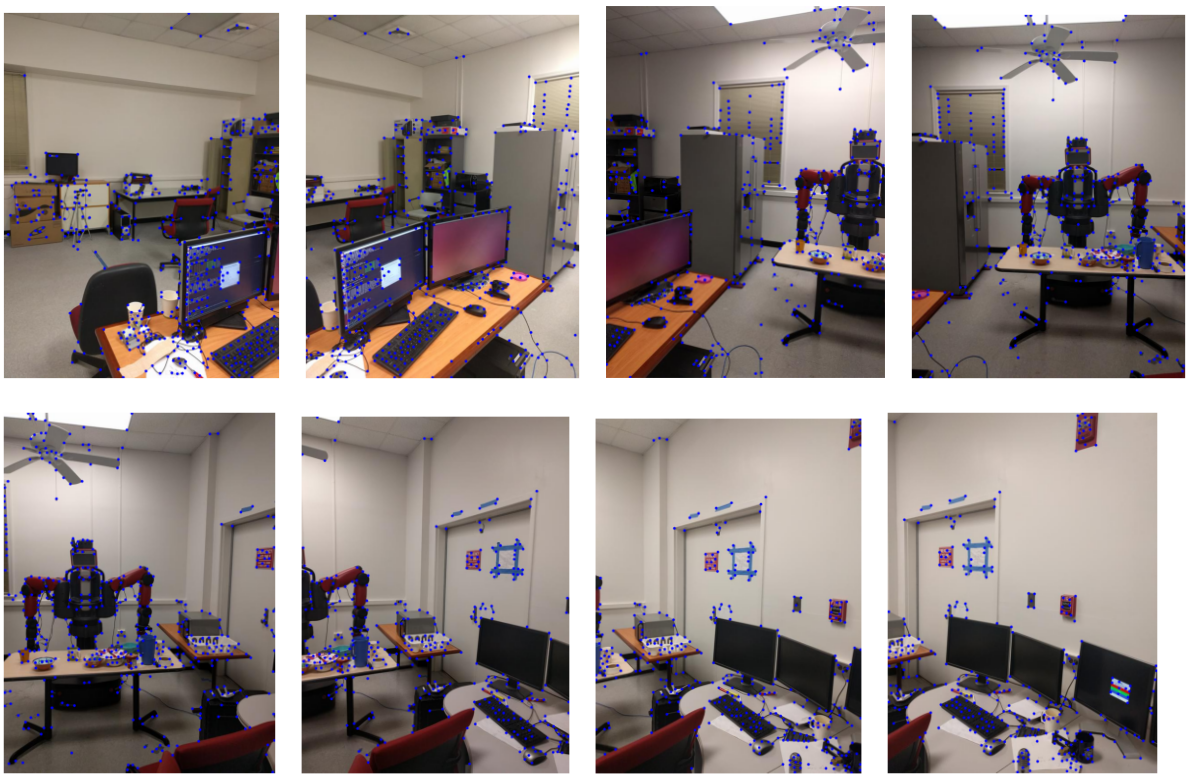


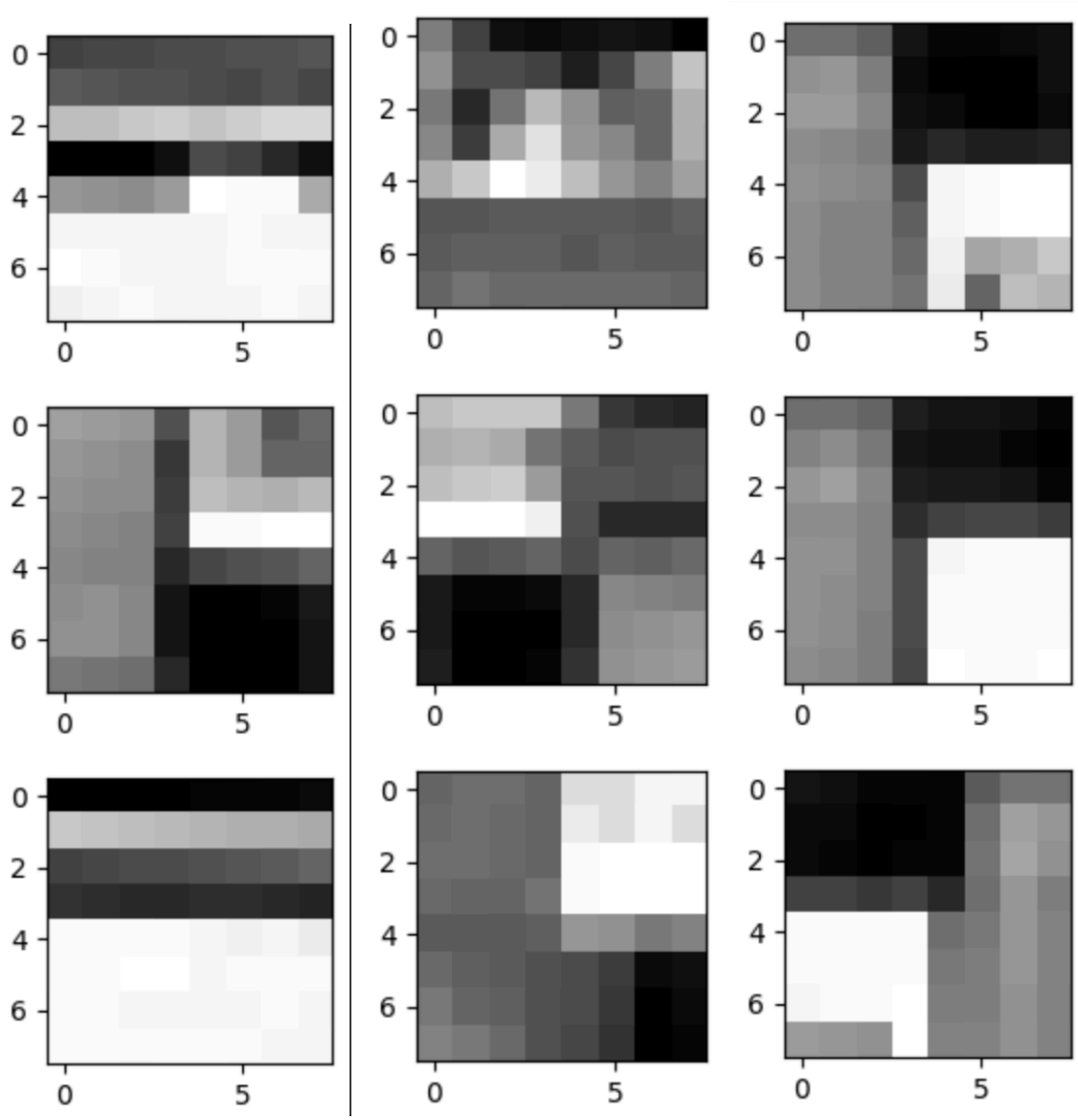
For ANMS, the algorithm seemed pretty straightforward, especially after reading the pseudocode given in the slides. We followed the pseudocode exactly. We started off by using a maximum filter from scipy which first dilates the image internally and then processes the local maxima. Next, we tried to match the maxima returned by maximum_filter but struggled until finally adding a second statement (cmap > 0) which seemed to fix weird dimensional issues. Next, we get the y and x coordinates of the maxima and initalize r. For each x and y coordinates we then find the local maxima then add the euclidean distance to r. Finally, we sort r and return the top num_best. The algorithm for ANMS was pretty simple however we struggled with its integration with the rest of the code due to the switching of x and y coordinates which resulted in weird outputs later in the code. We tried different num_best for ANMS such as 10, 50, 100, and 500 and settled on using 500. In the outputs, the corners seem much more concise now and you can visually match up many of them. Unlike just corner detection, which looks slightly noisy. The results are below:





The feature_descript method was pretty clear cut in the algorithm we needed to code to extract the features. The output of this method was an array of K x 64 x 1 vectors as well as the valid corners detected within the method, and the input was a grayed out image and corners collected by the ANMS algorithm. One part that was a bit tricky in terms of troubleshooting was building the vectors such that it would work within feature_match. However, it was a quick fix of the function flatten(). We also used this method to sort through valid corners for feature_matching; this took quite a lot of trial and error because it didn't initially occur to us while doing feature_matching that the matching was all over the place because of the use of all corners generated. To work around this issue, we filtered out valid corners while also creating the feature vectors in this method to then pass into feature_match. After doing this, the visualization improved dramatically, showing far more accurate matching between corners of the image.

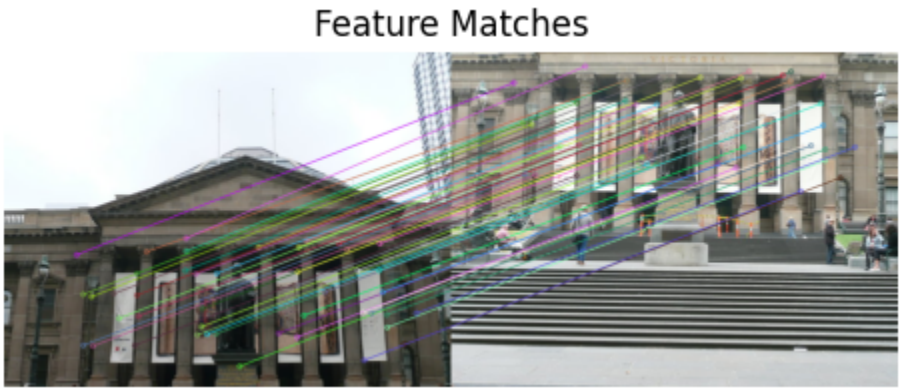
A couple of examples for the visualizations of the patches we used for feature descriptors are below:

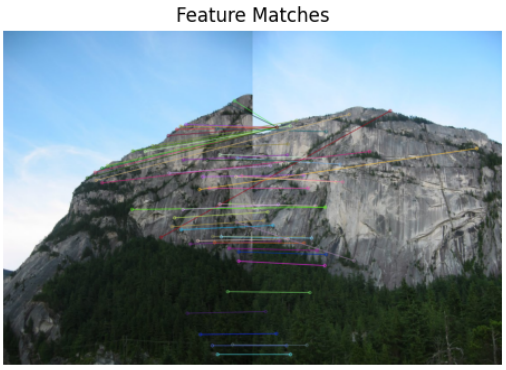


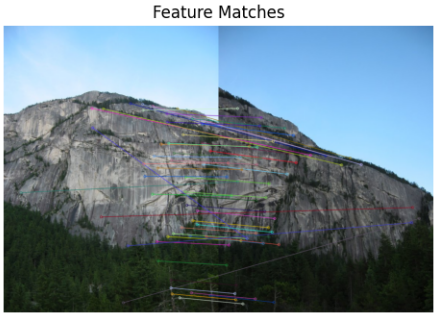
Feature Match:

For the feature_match function we had to find the top feature correspondences for the two images based on their feature descriptors. Our input for this method was a K x 64 x 1 flattened vector for K corners that was originally sized 8 x 8. In order to compute the best correspondences we employed the ratio test by looping through the descriptors for every corner in img2 and comparing it to a descriptor in img1. This was done through a nested loop in which we calculated the Euclidean distance between the descriptors, we originally utilized KNN to determine the closest neighbors; however we found this had lots of variation and couldn't find a K value with consistent results. We then took the smallest distance and divided it by second smallest to get our ratio. Another area where we had to do a lot of trial and error was with the thresholding we initially set a high threshold of .9 which allowed a lot of the ratios to fall under it. After

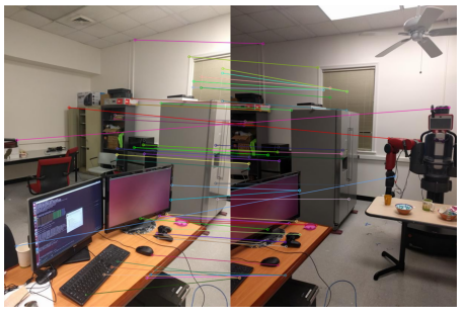
troubleshooting we arrived at .7 which worked well to eliminate some distances to ensure we only had distinct matches for each correspondence. In this section we ran into some trouble with the object creation of the DMatch objects as when accessing pixel values we used (y,x), but openCV still used the objects in the normal (x, y) form. This same issue was in the KeyPoint creation as well. The final portion of this section was drawing the matches for which we choose to display the top 50 matches to not clutter the visualization.



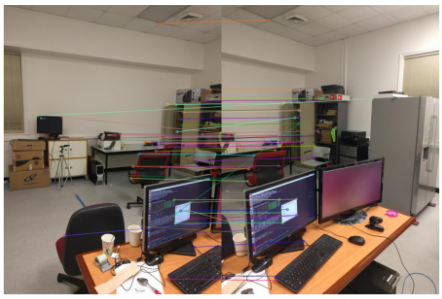




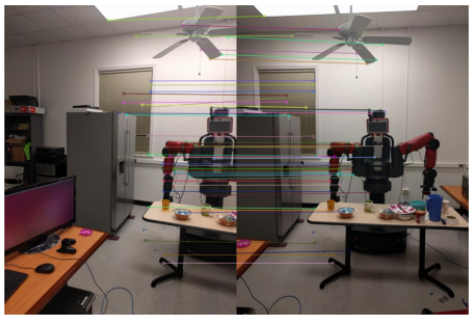
Feature Matches



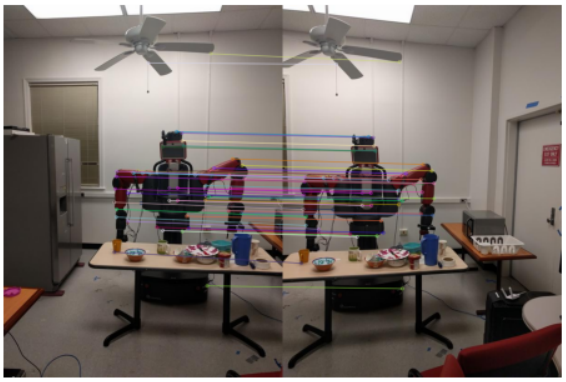
Feature Matches



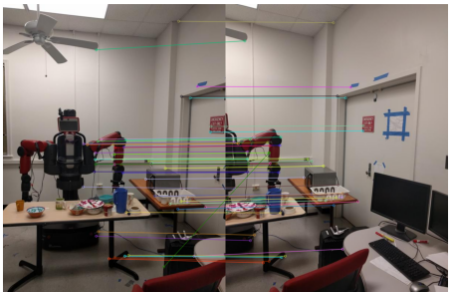
Feature Matches



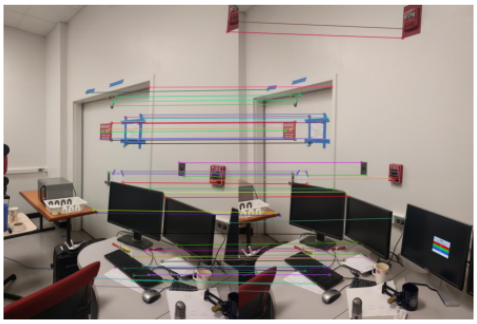
Feature Matches



Feature Matches



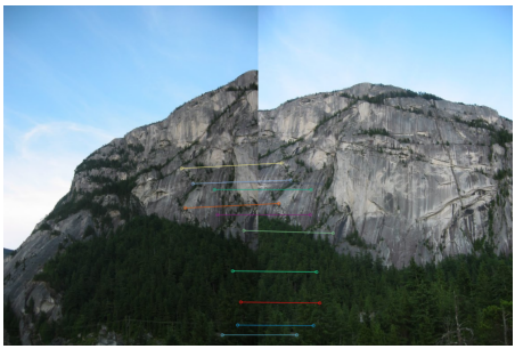
Feature Matches

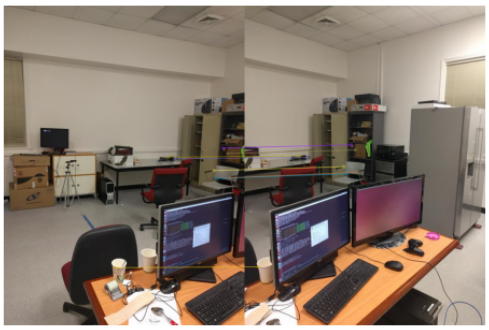


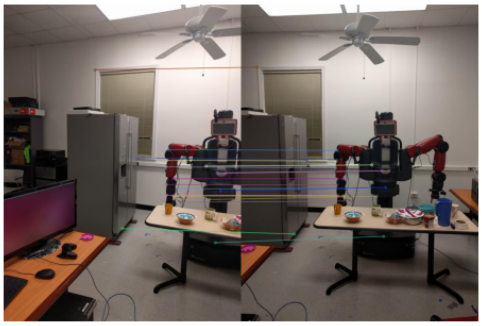
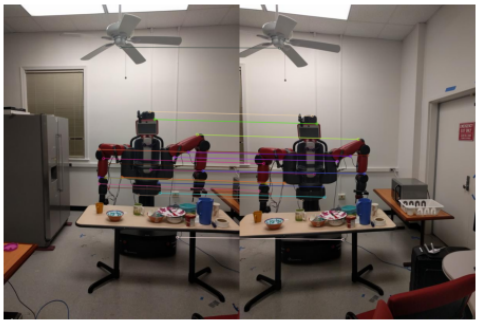
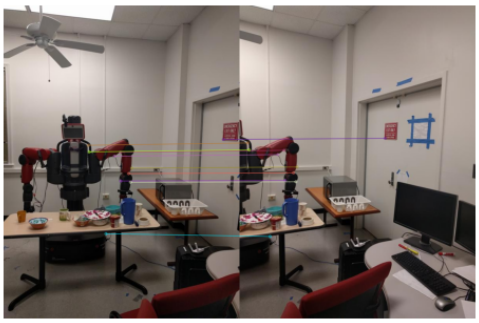
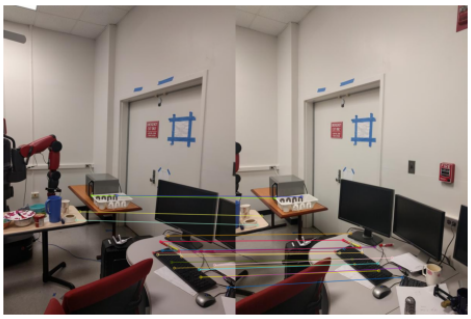
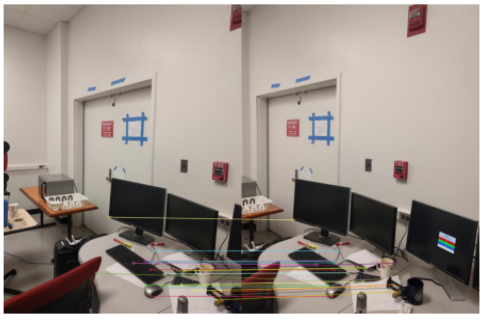
Feature Matches

RANSAC:

Since the RANSAC pseudocode and the header was already given, the implementation was very simple. We started off with two variables, homography and max_inliers to return from all the N loops. In each loop, we take 4 random pairs by using random.sample. Next, using those 4 pairs, we compute the homography by using cv2.gertPerspectiveTransform which returns a transformation matrix between the two sets of the points. Next, we calculate the inliers and loop through the matched keypoints and determine the outliers and inliers by using the homography transformation matrix. Then, we multiply the transformation matrix to get transformed versions of the two points. They are then normalized for further calculations. Then, we perform ssd on them to calculate the difference between the transformed points and the corresponding one. Finally, if the ssd is less than the threshold then that means it is an inlier. Then after the loop is done we update the inliers and the corresponding homography accordingly and then return the result. One small error that happened was mishandling the homography and max_inliers variables and returning the wrong ones. The return statement accidentally used the wrong variables and was indented too much. We were also partially stuck on how to calculate the ssd initially and had several errors due to it. There were some dimensional errors initially but they were quickly solved.



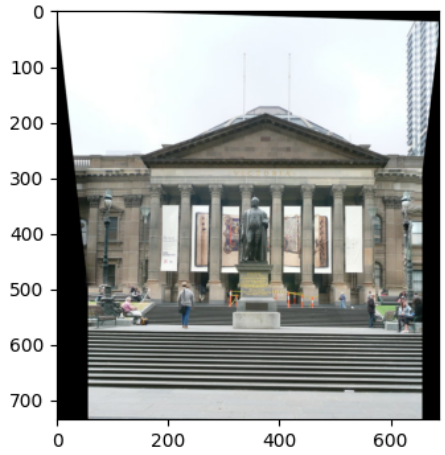


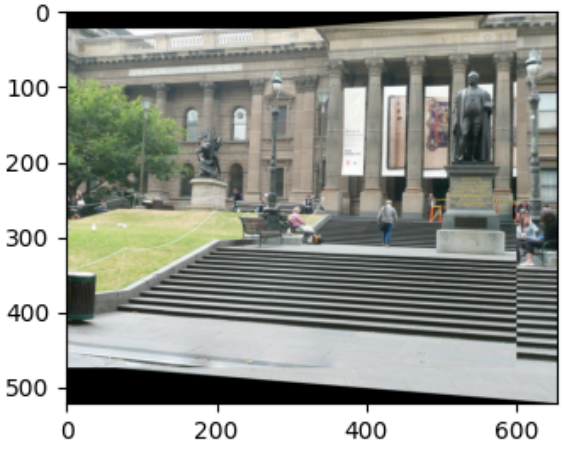






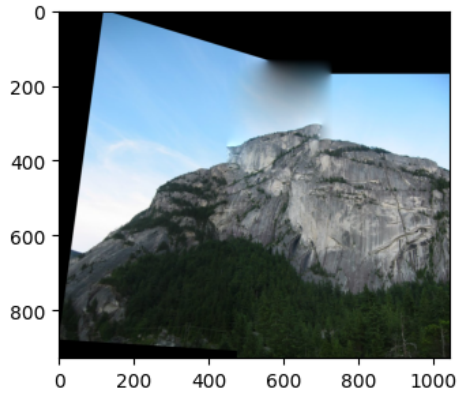
Warp and blend took quite a while due to the initially confusing linear algebra but we figured it out in the end. First we obtain the h1,h2,w1,w2 of the image and we compute the corners using perspectiveTransform and passing in the matrix:
[[0, 0]],

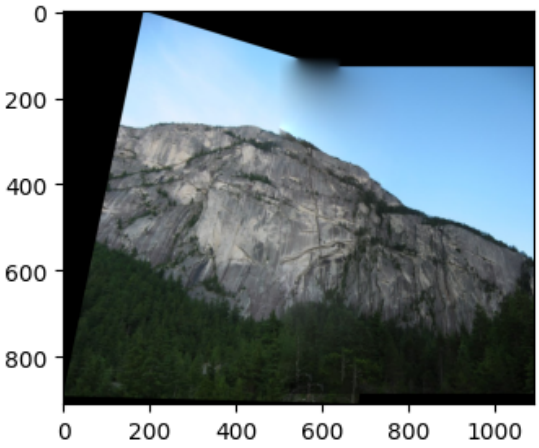
[[0, h1]],
[[w1, h1]],
[[w1, 0]]
(the 4 corners) along with the 3x3 homography to get the transformed corners.
Next we compute the min and max x and y values from the computed transformed corners. We initially found this confusing due to the unknown inputs and outputs of the perspectiveTransform method but the opencv page OpenCV: Geometric Transformations of Images examples as well as some testing off the ipynb helped us obtain the minimum and maximum x and y values which we can then feed into warpedPerspective.
Next, to get the x1x2y1y2 we used the same transformed corners and performed some basic calculations below:
x1 = max(0, -int(x_min))
y1 = max(0, -int(y_min))
x2 = x1 + w2
y2 = y1 + h2
With the min and max x and y values we then made a transformation matrix with the help of page 23 of the project 2 slides and OpenCV: Geometric Image Transformations
[[1, 0, -x_min],
[0, 1, -y_min],
[0, 0, 1]]
For parts without blending and basic images, we could stop here by just doing result[y1:y2, x1:x2] = img2 and returning result but some images we had to expand the result because of img2 issues. So then we check if result.shape[0] < y2 or result.shape[1] < x2 and if it is then we find the expanded height and width by taking the max of the shape or x and y. Then we make a new matrix expanded, copy over the result, then set the result to the expanded. Finally we can return the result with just result[y1:y2, x1:x2] = img2 but to do the seamlessClone, we need to add several more things.
In order to avoid dark images because of the black borders introduced in the transformations, we need to create masks. We start off by creating two masks, one to fill up the black spaces with white for where img1 is and a second mask for img2. The seamlessClone is still a confusing process but this masking technique worked the best after hours of different masks. After we obtain the two masks we perform & on them to get the overlap. We then get the coordinates of the overlaps. Next, we get the centers of the overlaps since PoissonBlending requires that. Now, we need to handle expanded sizes of masks just like the result matrix so we then again make a new matrix expanded, copy over the result, then set the mask to the expanded, for mask1. Finally, we make the image to blend by making an empty matrix of zeroes, copy over img2 and then feed it in seamlessClone as img 2. Finally, we return blended. We tried different types of cloning but full clone gives funky double image type results and monochrome just washes the colors away so we remained at normal clone. The most confusing part of this was adding the blending because we need to create and manage masks and expand them if necessary but that

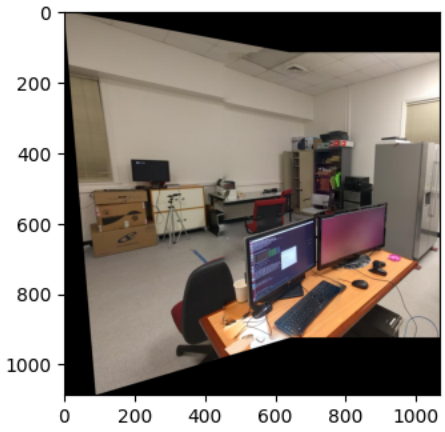
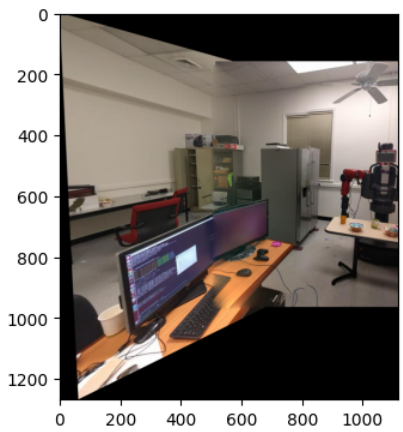
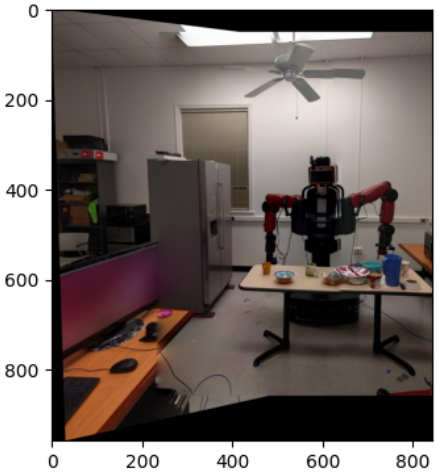
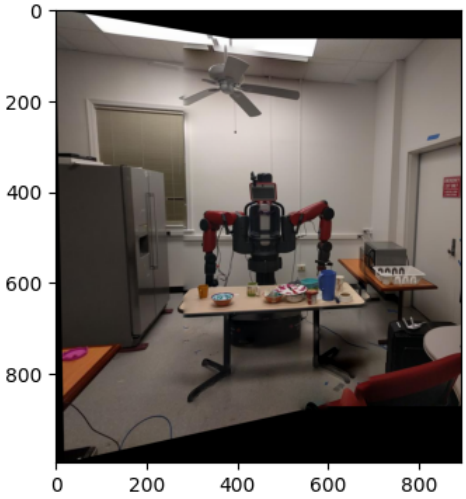
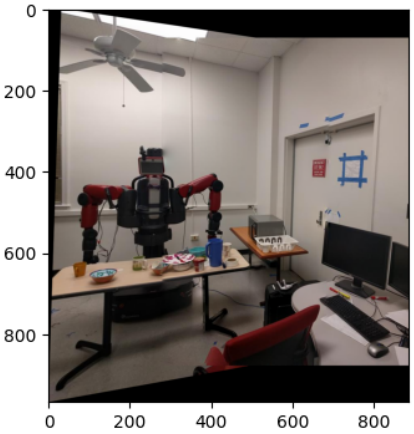
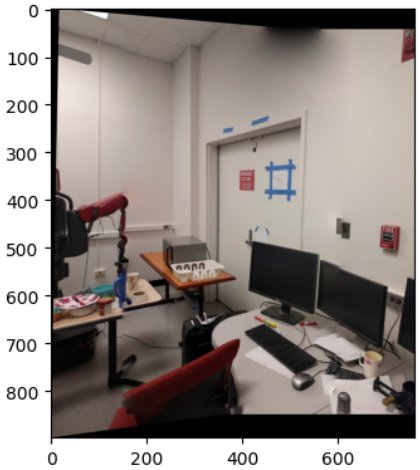
was solved with enough time. There is still some darkness seeping in but we are not sure anymore what other mask and how we need it. The results are:

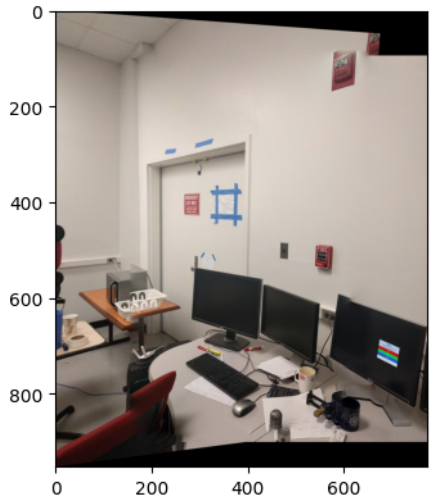










Final Panorama Images

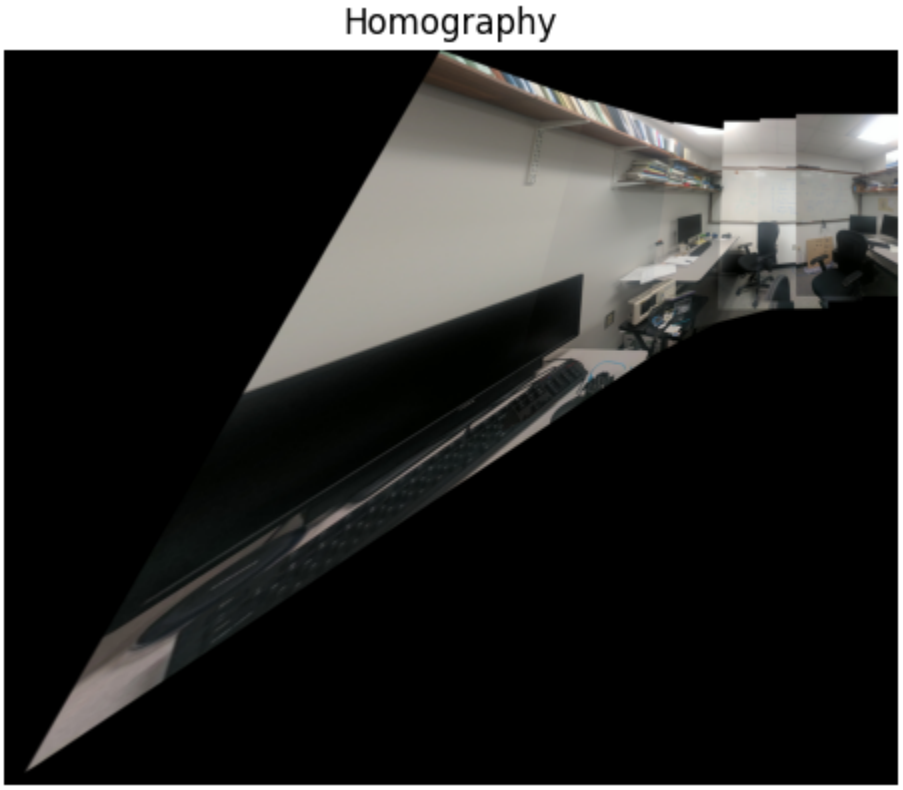
For pano_imgs we just had to combine all the different portions of our code in order to stitch images together. For this we funneled the result of detect corners with 1000 corners into ANMS which was then used by feature_match. Internally feature_match called feature_descript to get our new descriptors and keypoints to match on. From there we used RANSAC with 500 to compute the homography between the two images which was used in warp and blend to stitch them together. After these calls we stored the output of warp and blend in a result image which we successively fed back into the loop for every image in the set to arrive at our complete panorama.

When running our complete function on the test images we noticed that there was some darkness seeping in during warp and blend when run iteratively on subsequent images. In order to fix this we attempted to adjust our masks but couldn't fully erase it. We also attempted to recolor those portions of the images by applying a mask from the original image to restore those pixel values but it interfered with our pano_img on the test set so we opted to leave a few bits of darkness.

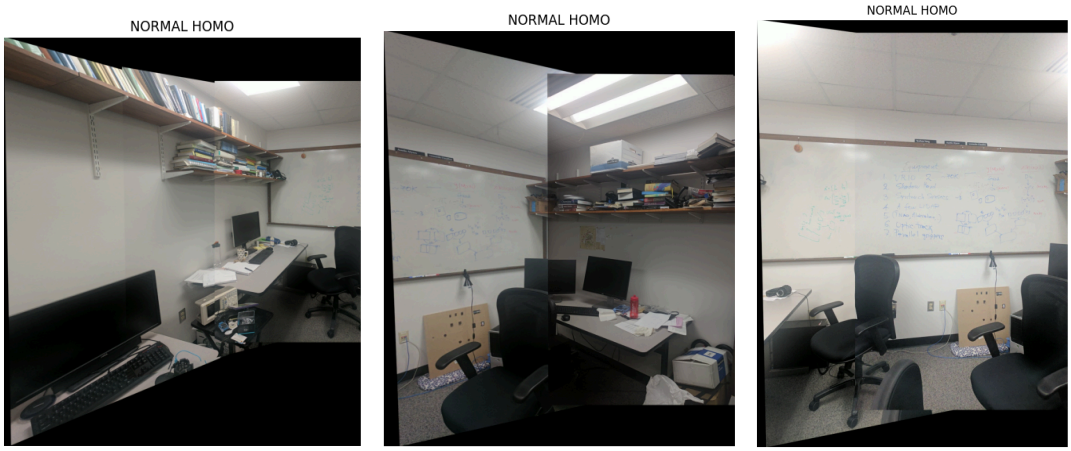
When dealing with stitching successively for the sets with more images namely TestSet2 and Set3 we noticed extreme distortion. We attempted to use cylindrical projection of the image and feature vectors for stitching but this threw off the results despite reducing the distortion. These two sets required the most effort out of the two problems, as initially the homographies we calculated for the recursively built stitched image and the new image were causing out of bounds error during blending. In order to remedy this for our results we removed seamless cloning and were able to produce results with distortion for Set3. However this did not resolve the issue with TestSet2 we believe was not stitching the images as the images within the folder were not successively taken and the two middle images were placed at the end making it extremely difficult to produce one cohesive result. I have pasted the intermediary results for the stitching

below where we were able to stitch images together that were successive in order. In order to adjust for this we used a cumulative homography but that resulted in a larger distortion due to image 3 and 4 being on opposite sides of the room with image 7, 8 being the ones to connect them. When reordering the test set images and going back to intermediate homographies we were able to achieve distorted stitching. Another method we attempted for this was to normalize the cumulative homography to isolate any scaling differences between the stitched images. We also scaled down each homography by the last element in the matrix in order to ensure it is always one.

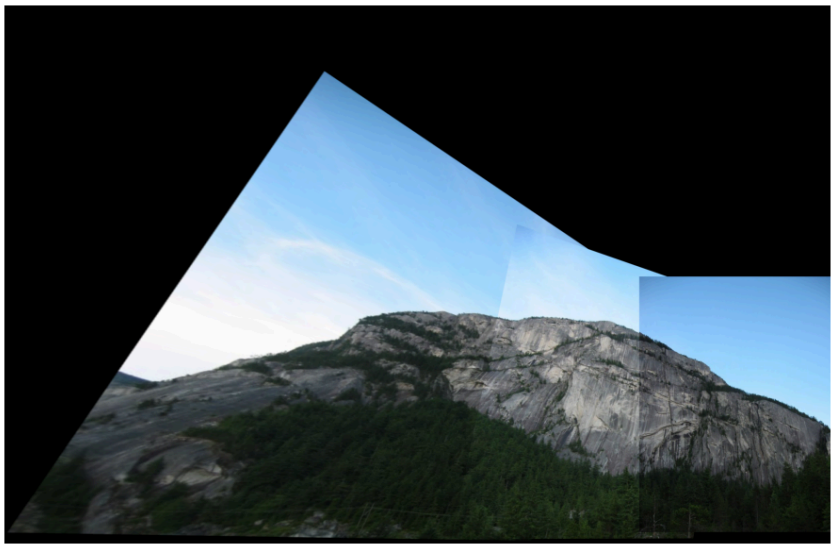
Reordered TestSet2 Results:

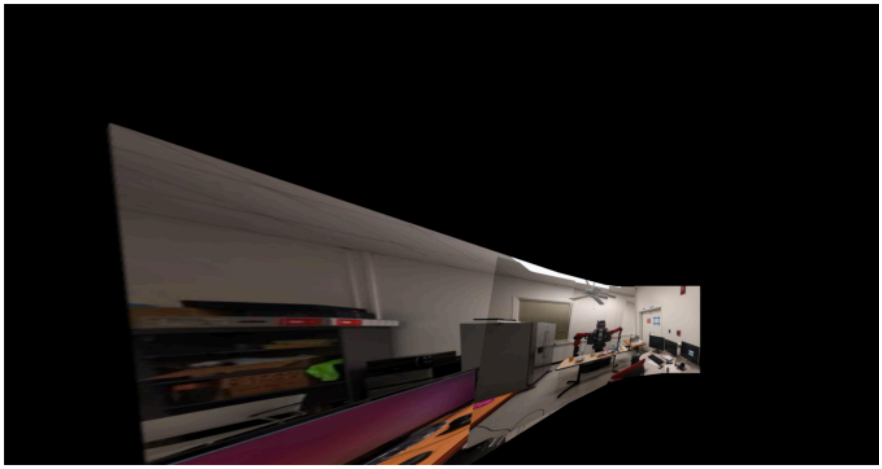


Intermediary Results:



Final Results for pano_img:





Panorama for Set3 before final stitching: