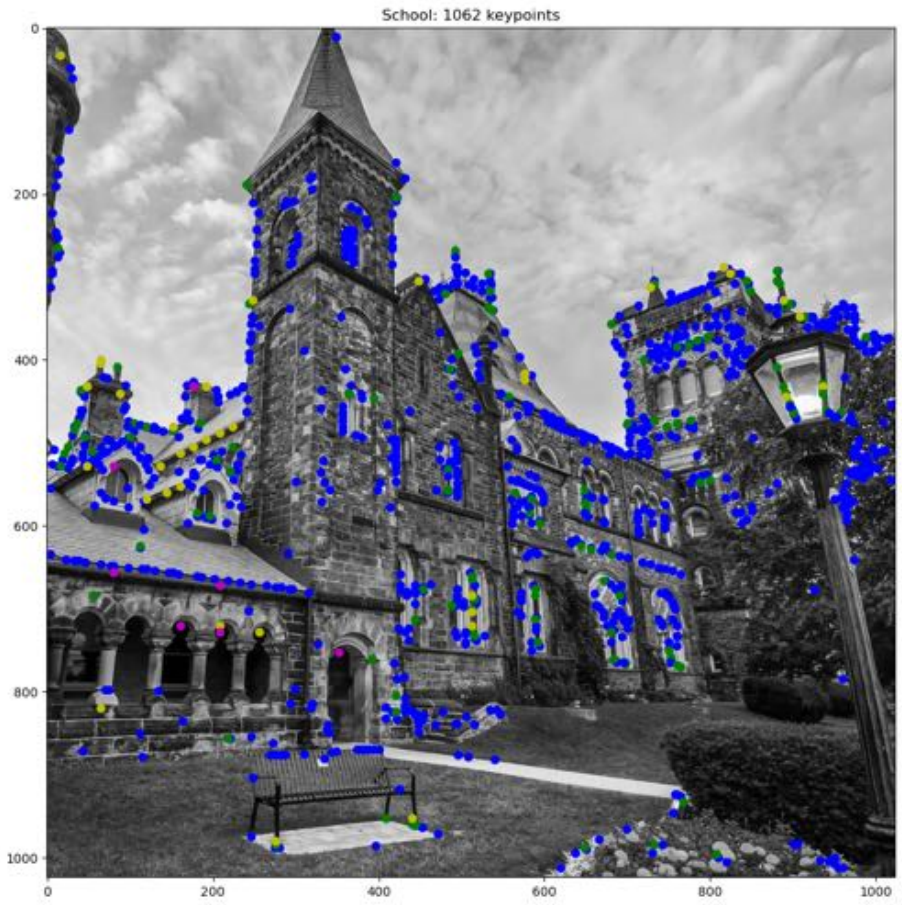
CSC420 A3

Jenney Ren

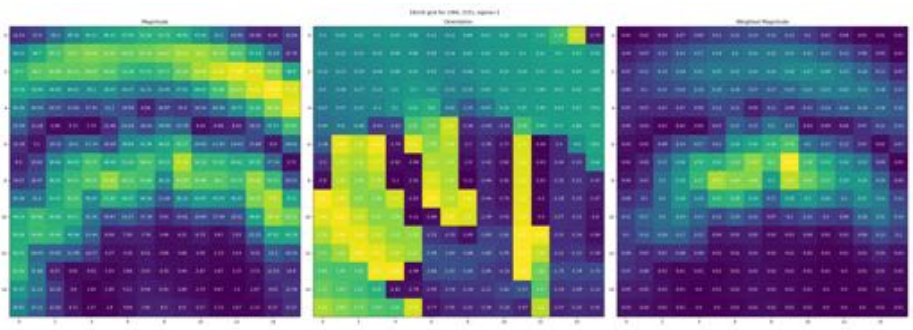
July 2020

Part I

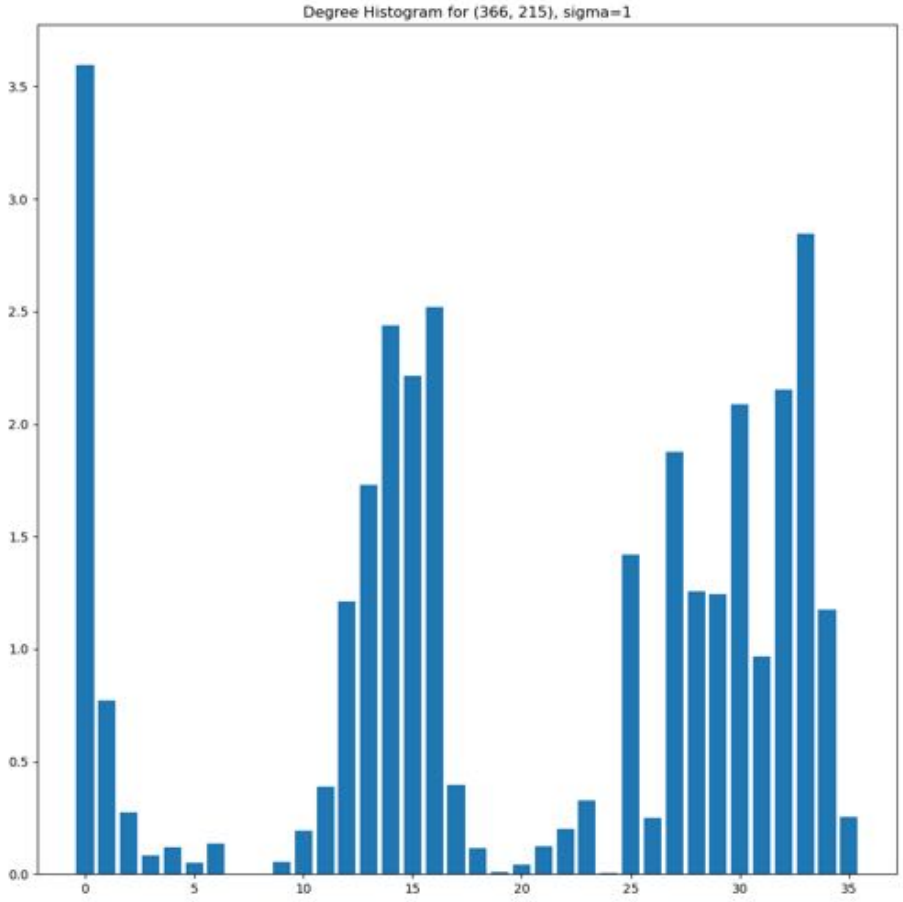
1. Keypoints detected on school image:



Gradient magnitude, gradient orientation, and 2D weighted gradient magnitude for a keypoint:

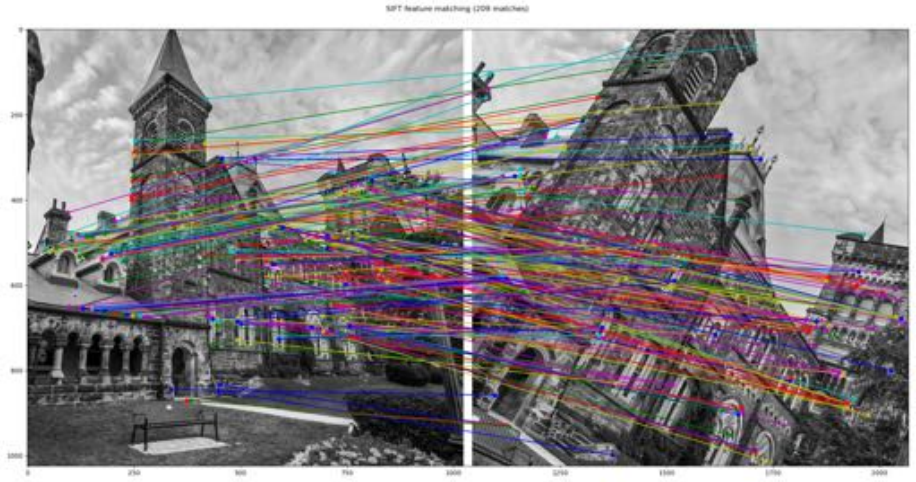


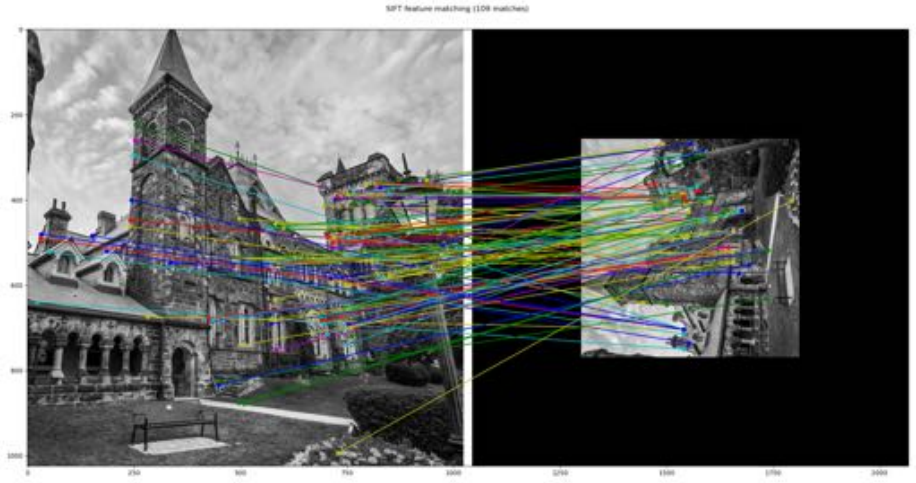
2. Orientation histogram for point shown in Q1 (after adjusting so that strongest peak is the first entry):



3. Rotated and scaled versions of original image (point at x0, y0 plotted in red):

4. Here are my results for the two images:



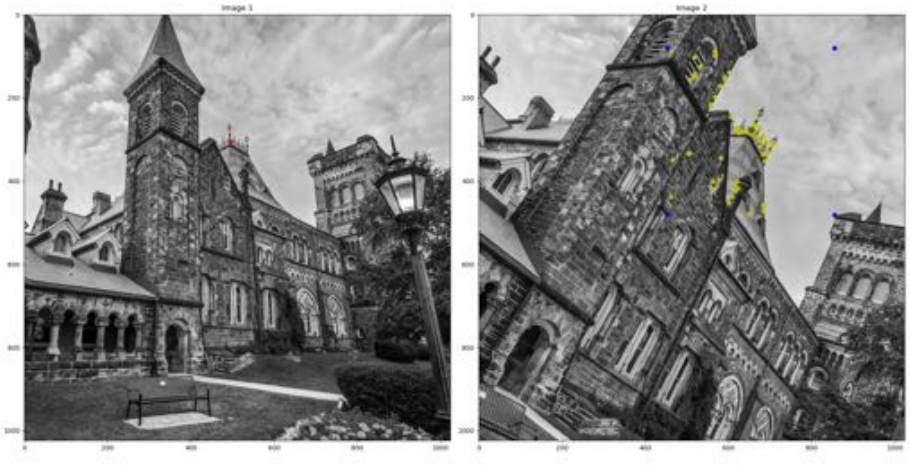


Looking closely, I found that some of the matches were better than others. The colors of the points in the below images represent:

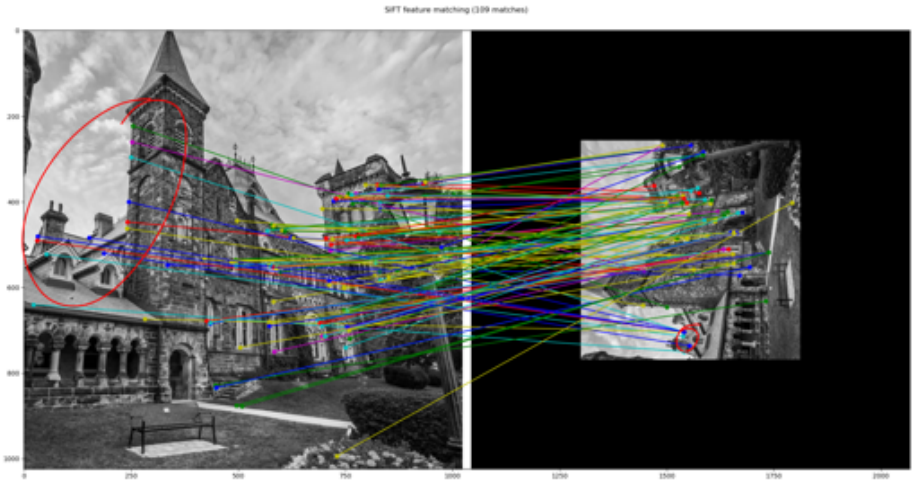
- Red: the original point on the first image and its corresponding location on the second image
- Blue: the four corners of the square representing the region the search is limited to
- Yellow: other keypoints found in the search region
- Green: the matched keypoint

In this case the matched keypoint was incorrect:

For this point, the match was very close to the actual position of the point:



It also seems that the image that is scaled down is more problematic than the one that is scaled up. For instance, the points in the red circle in the below figure show that many points tower of the original image are incorrectly matching with a few points on the roof windows of the second image.



My hypothesis for why the smaller image has worse matching (despite SIFT being scale invariant) is that this is caused by my original keypoint detector not being exactly scale invariant. I noticed that with smaller images, fewer keypoints are detected (presumably since there are fewer pixels). After tresholding, some areas with many keypoints in the original image may not have corresponding keypoints in the second image. Since the correct keypoint is not present, other incorrect keypoints seem like the best match.

5. The current matching strategy only looks at individual points, but we can also look at neighbouring keypoints since a correct match would also have similar neighbouring keypoints. One strategy would be creating a box (e.g. all the pixels in a 16x16 area) centered around a keypoint and another box centered around it's match, and seeing if there are other matching keypoints within the boxes. If there are other matching keypoints, we can be more certain the original keypoint is a correct match. The problem with this approach is that if we don't know the scaling for the second image, it is difficult to determine how large the box around the matching keypoint should be. For example, if second image was scaled by 10, the second box should be 10 times larger than the box for the first image. So, to take this approach we might have to try a series of box sizes for the second image (e.g. a 16x16 box, a 32x32 box, and a 64x64 box).

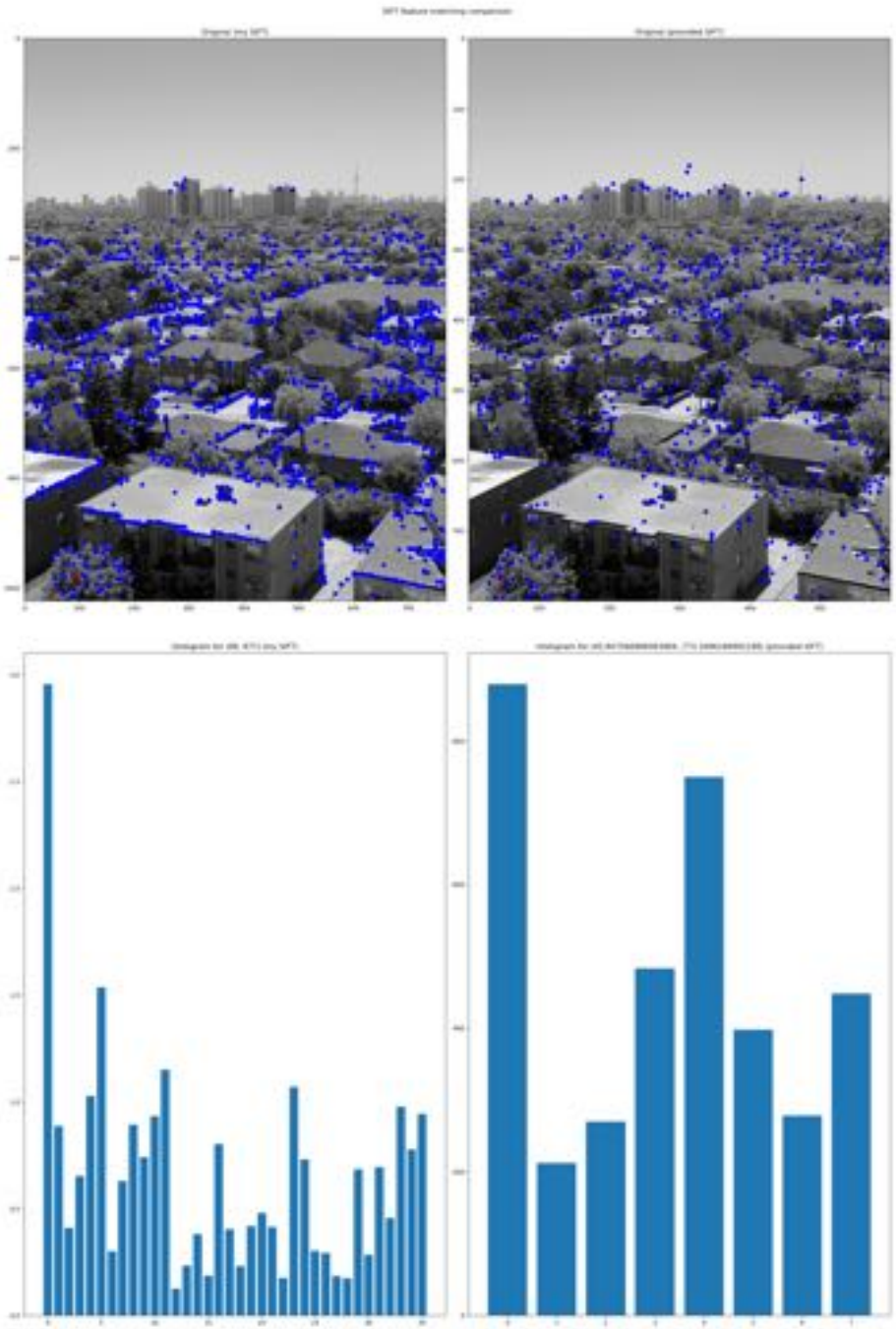
Part II

6. There seems to be no Q6.

7. I plotted a comparison of the keypoints generated by my implementation and the keypoints generated by the provided .mat file. The one red point in each image is the point being shown as a histogram on the bottom row.

   I noticed that my implementation of the keypoint generator tends to place keypoints on edges. For example, the building at the bottom of the image in has many keypoints on the edge of the roof in my implementation. Looking at the image closely, there seems to be a black border there, but I don't think this should be detected as a blob. This might just be a problem with my implementation, since for the previous sunflower image we were using, my blob detector also detected some false blobs on the horizon line.

   In comparison, the provided keypoints seem to be more spaced out compared to my keypoints, though there is still clustering. I looked briefly at the matlab implementation and I saw that the keypoint location was actually the fractional centre of a 16x16 grid, so I'm not sure if the matlab version's keypoints are more spread out because of that.



The SIFT feature vector is computed by dividing the 16x16 grid into 16 4x4 grids with each 4x4 grid having 8 bins, so I decided to add up all the

values in each of the 8 bins to make it easier to compare to our implementation. I choose to compare keypoints on the tree in the bottom left corner, since I believed this would result in a similar feature vector. Under my implementation, the dominant orientation's magnitude is significantly greater than the magnitude of other orientations, but under the matlab implementation there is another bar that comes close to the dominant orientation. I found this to be curious, since the orientation and magnitude calculations should be the same in both implementations, and the texture of the tree looks even throughout.

8. I used OpenCV2's ORB instead of SIFT to detect the keypoints, because SIFT is not available on all versions of OpenCV and I read that the work similarly. To match the features, I used OpenCV2's brute force matcher (which uses Euclidean distance) and the results were pretty good after using RANSAC, so I just kept it with the brute force matching. I also enabled cross-checking, which means the match only holds if the second image also finds that the matching point in the first image is the point with the lowest Euclidean distance.

I also plotted the top 10 matches (by lowest Euclidean distance). Compared to the image with all matches, the top 10 matches does not have any visibly poor matches.

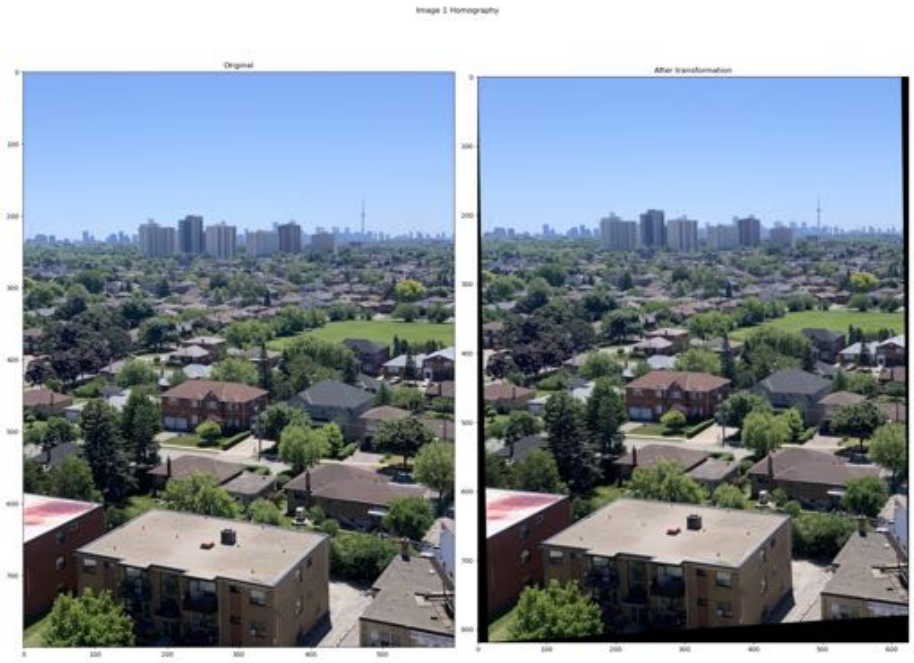
9. RANSAC algorithm:

   (a) Randomly select 4 matching points from the set of matching points and calculate the homography using these points. Select 4 because this is the minimum required number to get the transformation matrix.

   (b) Calculate the estimated new position of each point after the transformation, and compare to the actual position of the matched point by calculating the SSD between the actual and predicted points. Points are considered inliers if they fall within an error threshold.

   (c) Repeat steps 1 and 2 iteratively to improve the inlier estimate. Homographies with lower SSD are considered better models.

   When implementing the algorithm, I set the error threshold so that there are no visibly poor matches included in the inlier set. I found that around 2000 iterations of the algorithm avoided the homography changing significantly between different runs.



10. My strategy for creating the panorama was merging the left images and the right images, and then combining the left side and the right side to create a final image. Initially I only merged from the left, but I found that leftmost image was very warped, so I tried merging from both sides instead.

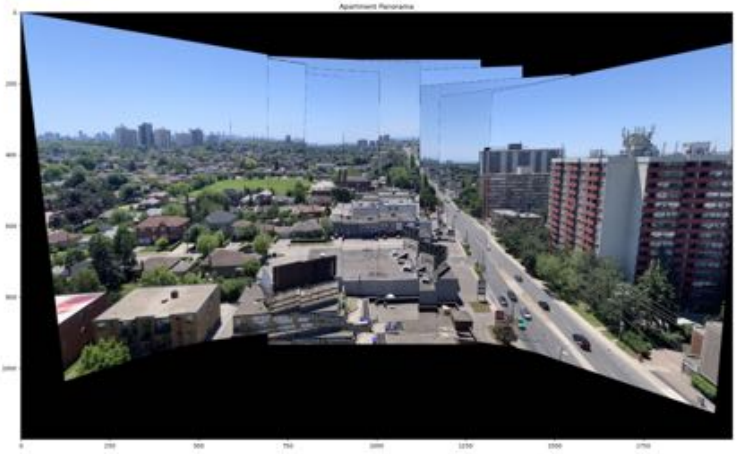
    I created a merge_left function takes two images, applies the homography found by RANSAC to the first image, and then aligns the two images using the inlier keypoint with the lowest SSD. This is the result of combining image 1 and image 2:

    Then, to merge multiple images I used my combined image of image 1 and image 2, and applied left_merge with the combined image and image 3:

This worked ok when a small number of images were used, but with more images the results did not look as good. The results of the merging images 1 to 5 are shown below.

To merge the right side, I used the left merge but flipped the images vertically, and then flipped the image back after I was done merging.

Finally, I combined the left and right side together to get the end result.



There are some problems with this approach. The bottom area near the building with the big billboard seems to be combined poorly, because you can see some of the later images in the sequences peeking out from the

bottom since the later images in the sequence are slightly longer. It would be necessary to either crop that part out or to change the combining of the image so that a slice of the next image in the sequence is added, instead of having the next image in the sequence fill in all the parts that are not black. There were also problems with combining the left and right side, and the middle part where the left and the right side join together has some noticeably bad edges. For example, horizon line is clearly lower where the left images join the right images. I believe this is a bug with my code, but I haven't been able to fix it. I think the reason this happens is partially because there is overlap between the right image and the left image (since the large black billboard is in both images), but my algorithm doesn't weight the two images; I set the code to pick the left side as long as the left side is non-zero. It might also be a problem with the translation I used to align the keypoints.

Bonus

11.

12. I read that two commonly used methods to remove seams are optimal seam algorithms, which find a curve in the overlapping region where the difference between the two images are minimal, and smoothing algorithms that smooth the transition between the two images. A common way of smoothing the transition between the images is Laplacian pyramid blending.

    Laplacian pyramid blending steps:

    (a) Generate the Laplacian pyramids for image 1 and image 2

    (b) Create a Gaussian matrix to use as weights for the blending. For example, if we are trying to blend a left image and a right image, the Gaussian matrix will create a smooth seam in the middle by almost equally weighting the two images close to the middle, weighting the left image more as we go closer to the left side, and weighting the right image more on the right side.

    (c) Apply the Gaussian matrix to the two images on each level of the Laplacian pyramid to get a new combined Laplacian pyramid.

    (d) Collapse the Laplacian pyramid into one image to get the final merged image.