P3 Matching Pipeline

Benjamin Geyer
G00997436

P3.1 Scaling and Rotating Patches
P3.1.1 Scaling and Rotating Features: Concepts
The first thing we will need is the ability to compute an image patch corresponding to the feature. To do that, the image patch will need to translate the images, rotate them, and scale them. Fortunately, in the last assignment, you were asked to write code that transformed an image using a general homography matrix H. First, a conceptual question:

(QUESTION) If I have a feature located at (xf,yf) with orientation θ and radius ("scale") s, what is the transformation matrix H that simultaneously moves the feature to the origin, un-rotates it, and un-scales it (so that the feature becomes 1 pixel wide)?

For example, if I had a feature that was already at the origin, and not rotated, but was scaled such that it's radius was 10 pixels wide, the transformation matrix would need to make the feature smaller, so H would be defined as:

$$H = \begin{bmatrix} 1/10 & 0 & 0 \\ 0 & 1/10 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note: Because of the challenges involved with intuiting the direction of the transformation, I will accept either the transformation I have described, or its inverse (which you will likely need for the next part of this question).

H^{-1} = [
[s*cos(θ) s*sin(θ) xf],
[s*-sin(θ) s*cos(θ) yf],
[0, 0, 1]]
H = [
[1/s*cos(-θ) 1/s*sin(-θ) -xf],
[1/s*-sin(-θ) 1/s*cos(-θ) -yf],
[0, 0, 1]]

P3.1.2 Scaling and Rotating Features: Implementation¶
The inverse of the transformation matrix I have asked for above can be used to compute an image patch surrounding a feature that compensates for both the scale and the orientation of that feature. The computed patches can then be used as feature descriptors for feature matching to align images. In this part of the question, you will implement the warping function to compute these patches. I have provided you with starter code in the function get_scaled_rotated_patch below. Missing is the transformation matrix, which can be implemented using your solution to the previous part of this question. I have used scipy.interpolate to implement the interpolation in the warping loop itself; you should feel free to use this implementation.

I have included some sample code following the get_scaled_rotated_patch function that generates some image patches for various parameters on a reference image. If your transformation is implemented correctly your figure should look as follows:

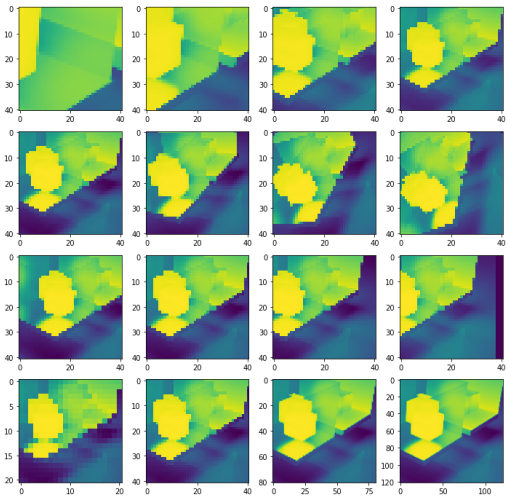
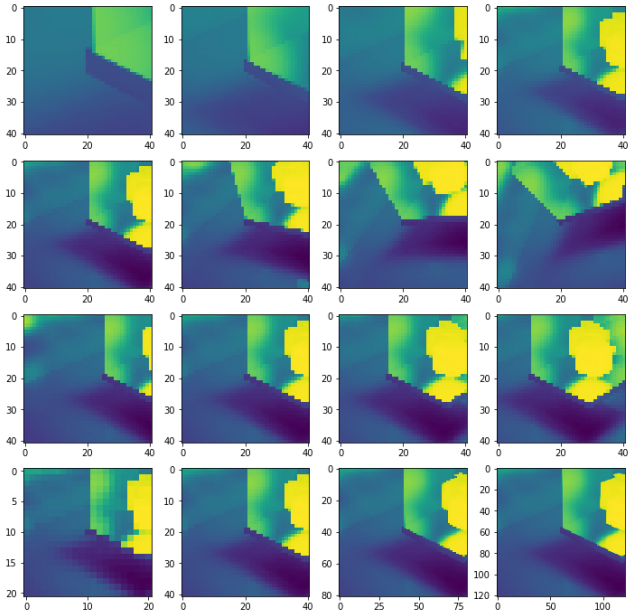


FIGURE: To demonstrate that your code is working, change the following parameters base_center_x = 500 and base_center_y = 640 and regenerate the figure. Include this figure in your writeup.

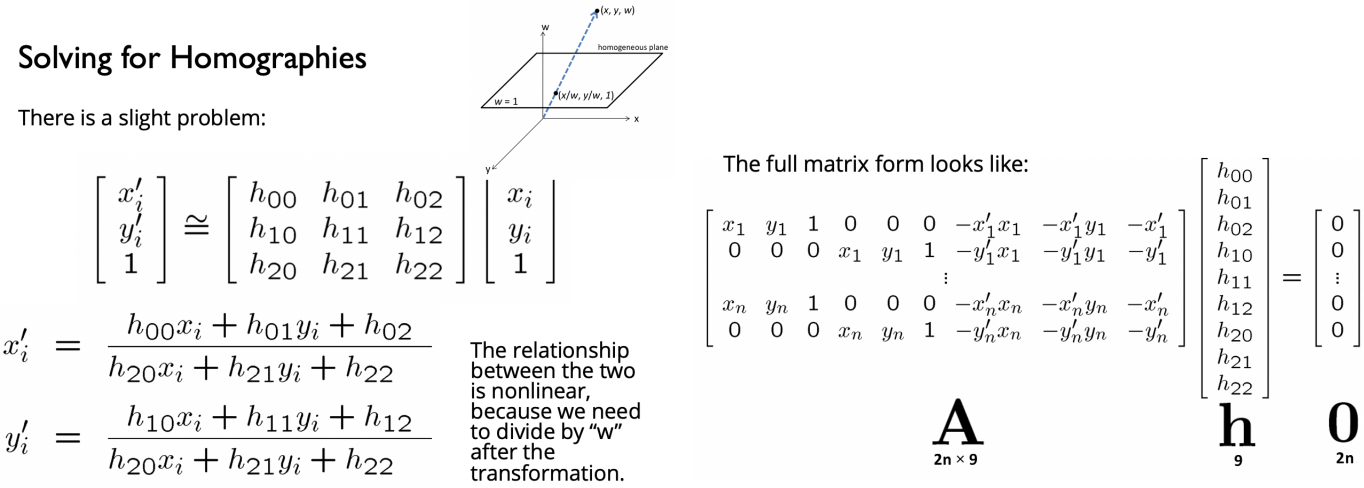


P3.2 Computing Homographies from Matches

P3.2.1 Computing Homographies from Perfect Matches

In this problem, you will be computing homographies from feature matches that you generate by hand. This involves two steps:

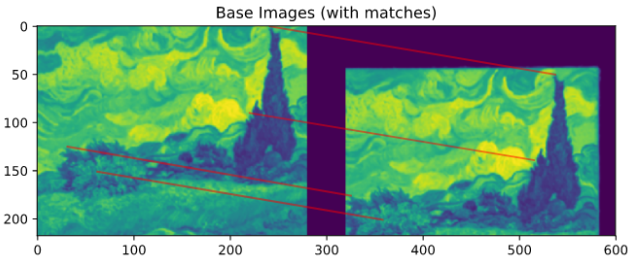
1. Obtain feature matches between the two images. For this question will be "computing" these matches by hand (you will do this automatically in another question). Most operating systems have an image inspection program that allows you to quickly get the coordinates of a pixel. Alternatively, you can use trial and error with the visualize_matches function I have provided for your convenience below.
2. Compute the homography matrix H using the procedure introduced in class. This means that you are trying to find a matrix A that satisfies the following relation:

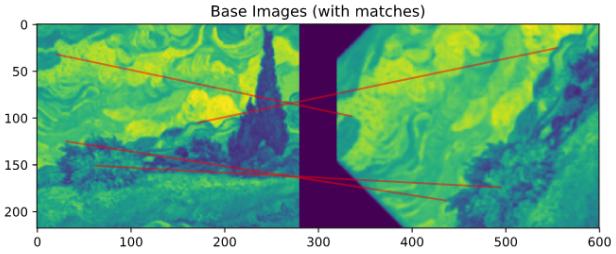


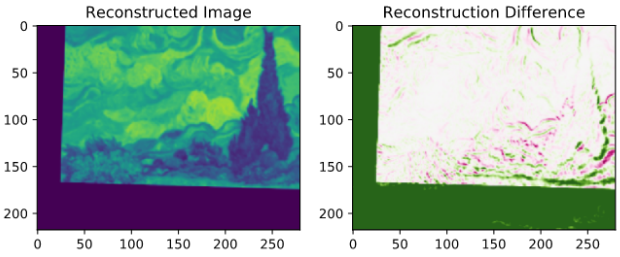
In the code block labeled An example of match visualization below, I have given you a full worked example of what this process will look like: I have generated an example transformed image using a known homography H_known, provided some matches, used those matches to compute the homography H_computed (using a function you will write), and then visualized the results using visualize_computed_transform. This is what a "correct solution" should look like.

FIGURES I have provided you with 6 images: img_base (the starter image in the code below), and 5 "transformed images", each with different homography matrices. For each "transformed image", manually identify at least four matches between it and the base image, use these matches to compute the homography H and use the visualize_compute_transform function to generate a plot. Include these plots in your writeup.

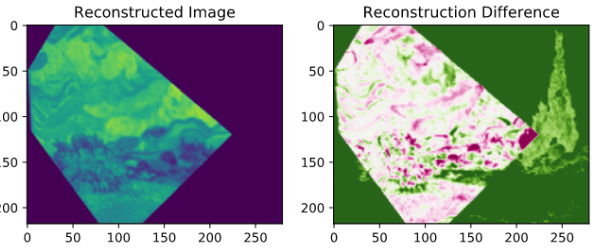
Your "reconstruction difference" plots are not expected to be perfect, but should be reasonably close to accurate; you will not be penalized for small differences. If the reconstructed image is completely different from the base image, you will be marked as incorrect.



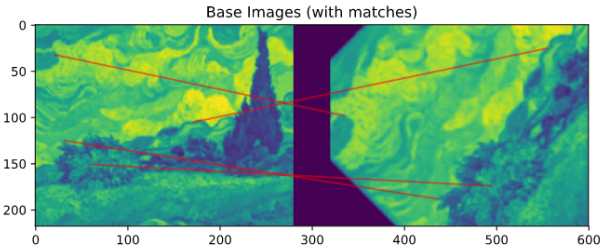


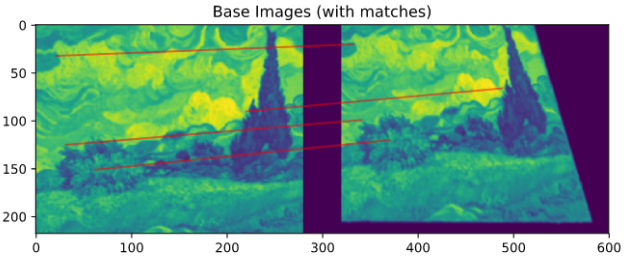


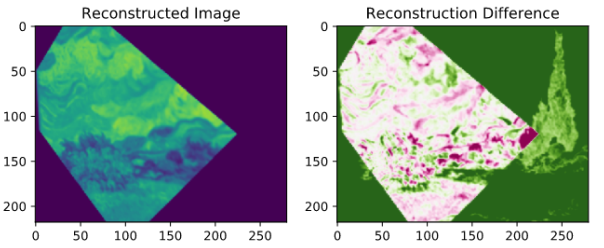
Translated



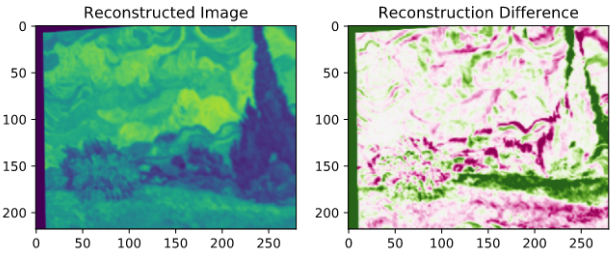
Rotated



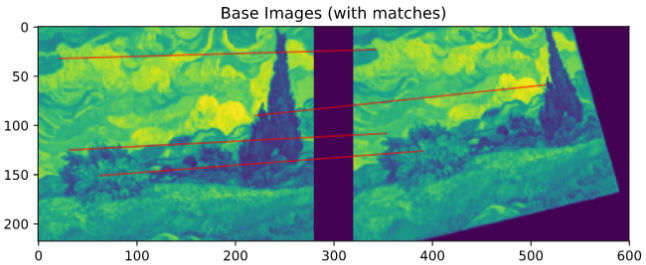


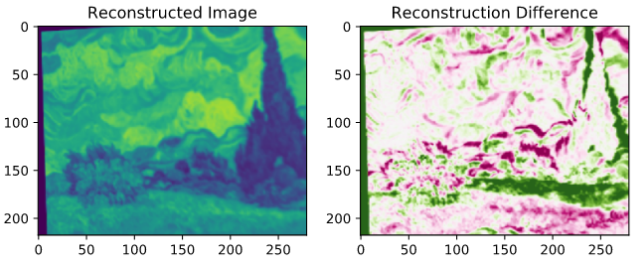


Aspect Scaling



Homography A



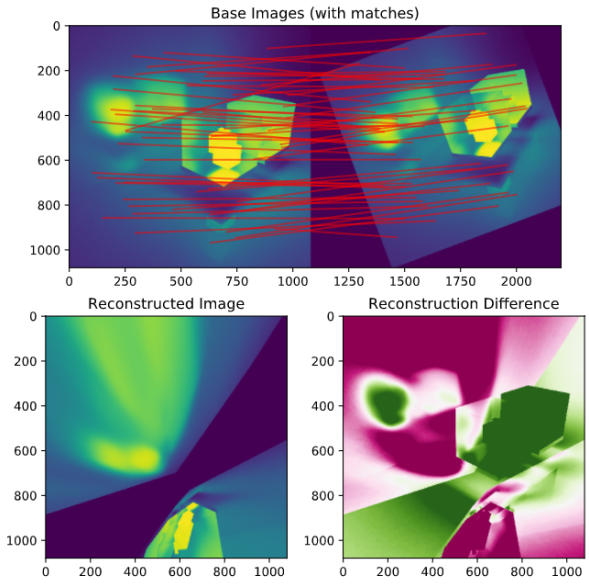


Homography B

P3.2.2 Computing Homographies from Noisy Matches
Now, I will ask you to compute the homography of a transform from a set of matches, where some of the matches are "outliers". The idea is that you will need to use RANSAC to compute which samples are inliers and which are outliers.

I have provided you with two sample images below (in Noisy Matches Base Code), and a set of matches. In the plot I have generated, you can see that though many of the matches are correct, there are a few outliers that will ruin the computation of the homography.

FIGURE Compute the homography with all of the matches_noisy I have provided and visualize using visualize_computed_transform. The resulting transform should be quite poor. Include this plot in your writeup.



Noisy Reconstruction

```
Computed Noisy Homography:
[[-2.36177072e-01 -4.33950744e-01  4.40522664e+02]
 [-4.37147094e-01 -2.79563922e-01  4.62647298e+02]
 [-6.59026927e-04 -7.99738555e-04  1.00000000e+00]]
```

Next, you will be implementing the RANSAC procedure we discussed in class and use it to compute a homography that is robust to the outlying detections. Implement RANSAC procedure from class and use this in combination with your solve_homography function to compute the homography despite outliers in matches_noisy. You will need a function that computes the inliers from the set of all matches and a proposed transformation matrix H. matches_noisy has a 10% outlier ratio. You should call your function solve_homography_ransac(matches). You will need it again later. Please include a code block containing your implementation of the RANSAC procedure in your report; it will help us give partial credit in the event that it does not appear to be working correctly.

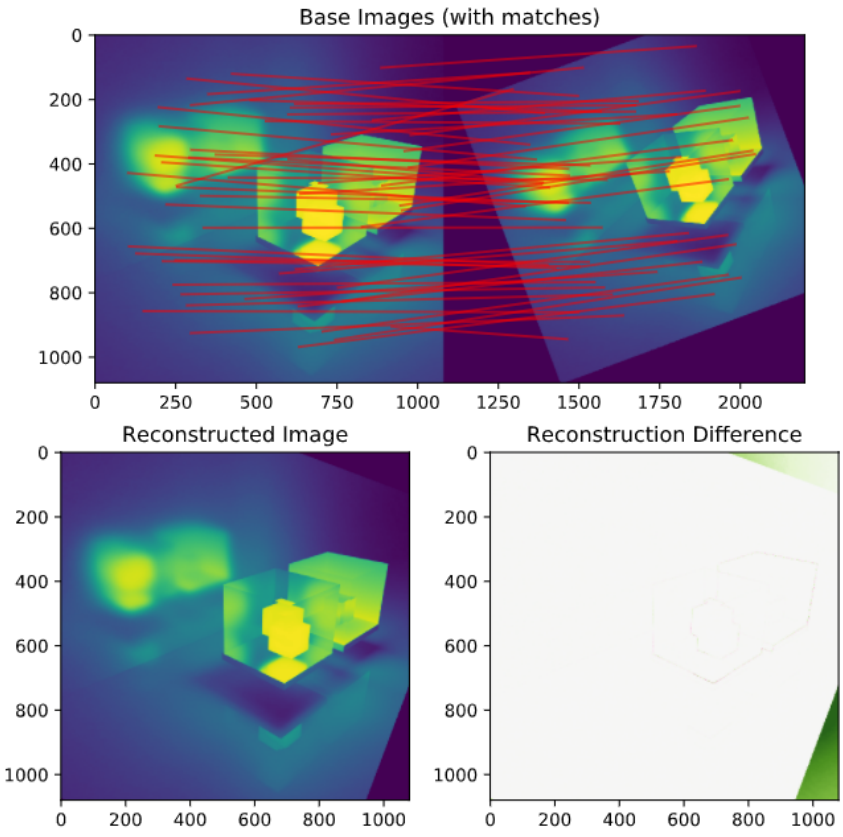
```
def get_inlier_matches(matches, homography, sigma):
    # Show that the chi-squared threshold
    # results in 95% inliers.
    # d, chsq_thresh = 1, 3.84
    d, chsq_thresh = 2, 5.99

    threshold = chsq_thresh * (sigma**2)
```

```
    ones = np.ones(matches.shape[0])
    orig_coords = np.stack((matches[:, 0], matches[:, 1], ones), axis=-1)
    match_coords = np.stack((matches[:, 2], matches[:, 3], ones), axis=-1)
    new_coords = (homography @ orig_coords.T).T
    w = new_coords[:, 2]
    new_coords[:, 0] = new_coords[:, 0] / w # normalize by w again
    new_coords[:, 1] = new_coords[:, 1] / w # normalize by w again

    x_diff = new_coords[:, 0] - match_coords[:, 0]
    y_diff = new_coords[:, 1] - match_coords[:, 1]
    debug_print(f'x_diff: {x_diff}, y_diff: {y_diff}')
    rsq = x_diff**2 + y_diff**2
    debug_print(f'rsq: {rsq}')
    debug_print(f threshold: {threshold}')

    return matches[rsq <= threshold]

def solve_homography_ransac(matches, rounds=100, sigma=5):
    best_inlier_matches = []

    for i in range(rounds):
        debug_print(f'round: {i}')
        points = np.random.choice(np.arange(matches.shape[0]), size=matches.shape[1])
        initial_homography = solve_homography(matches[points])
        inlier_matches = get_inlier_matches(matches, initial_homography, sigma)
        if len(best_inlier_matches) < len(inlier_matches):
            best_inlier_matches = inlier_matches

    return solve_homography(best_inlier_matches)
```

RESULT Include the computed homography matrix in your solution. Make sure the matrix is normalized such that the bottom right corner is equal to 1.

```
Computed Robust Homography:
[[ 8.00000003e-01  3.00000002e-01 -1.39703864e-06]
 [-3.00000000e-01  8.00000003e-01  2.19999999e+02]
 [ 1.79237931e-12  2.44681114e-12  1.00000000e+00]]
```

FIGURE Using visualize_computed_transform, visualize the transform you have computed using RANSAC and matches_noisy. Your solution should be quite accurate. If most of your reconstruction_difference plot is non-zero, something is probably wrong.



RANSAC Noisy Reconstruction

3.3 Feature Matching Pipeline

I have provided you with a function compute_features_with_descriptors which, well, computes features and their descriptors (via the compute_scaled_rotated_patch code you wrote earlier). However, the function is not quite complete, since it still needs a multi-scale feature detector. Fortunately, you wrote one of those in your last assignment:

TASK Define the function compute_multi_scale_features I have created in the Code you need to provide below. You can do this with the code you wrote for your last assignment. Notice that I have provided you with a Feature class in the code below. The compute_multi_scale_features function is expected to return a list of these Feature objects for the remainder of the code to work as expected.

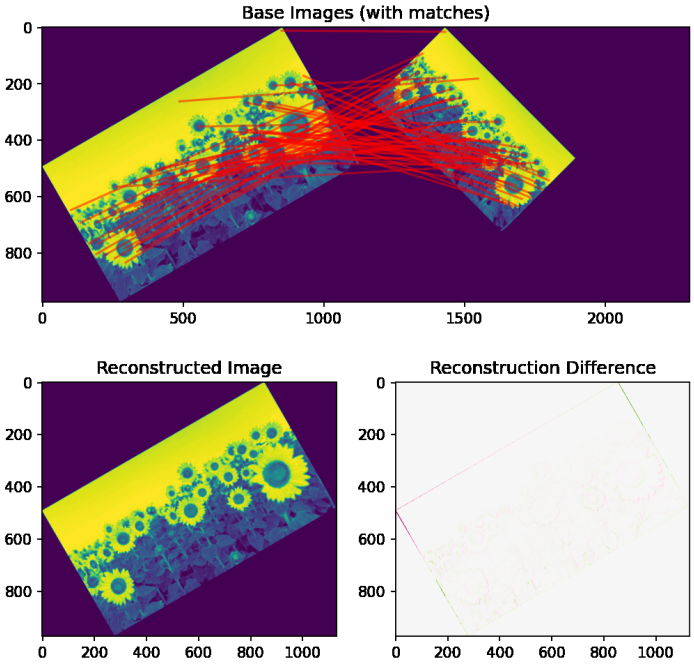
To confirm that you are computing feature patches and orienting and scaling them correctly, it might be worth visualizing them (though you do not need to include these in your writeup). An example code snippet might look like:

```
## Visualize Patches
sigmas = np.arange(5, 40.0, 1)
image = load_image('light_cubes_base.png')[::1, ::1, 0]
features = compute_features_with_descriptors(image, sigmas, 0.6)

# Plot a few of the feature patches for your own reference
# You should see that they are all aligned.
plt.figure()
for ind, f in enumerate(mfeatures[:9]):
    plt.subplot(3, 3, ind+1)

    plt.imshow(f.descriptor)
```

TASK Finally, implement the function compute_feature_matches(fsa, fsb), which returns a list of matched feature pairs [fa, fb] from two lists of Feature objects. Once again, you should be using your feature descriptor matching code from the last assignment.

FIGURE I have included code under Putting it all together that, (1) computes features, (2) matches between them, (3) the homography to align the images, and (4) the plot showing the performance of the alignment. If you have finished implementing the previous functions, the final plot should show all the pieces working in harmony on the two transformed sunflower images I have provided! Run this code and include the resulting plot in your writeup, showing that you computed reasonable features/matches and the homography that aligns the images.

```
Computed Homography:
[[ 1.70436804e-01 -6.43588872e-01  5.78307500e+02]
 [ 6.39540962e-01  1.69470411e-01 -8.23602477e+01]
 [-2.28567653e-06 -3.37341069e-06  1.00000000e+00]]
Total Compute Time: 74.279541015625
```

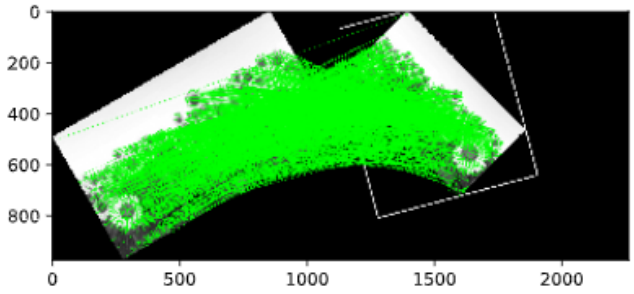


Homography Reconstruction

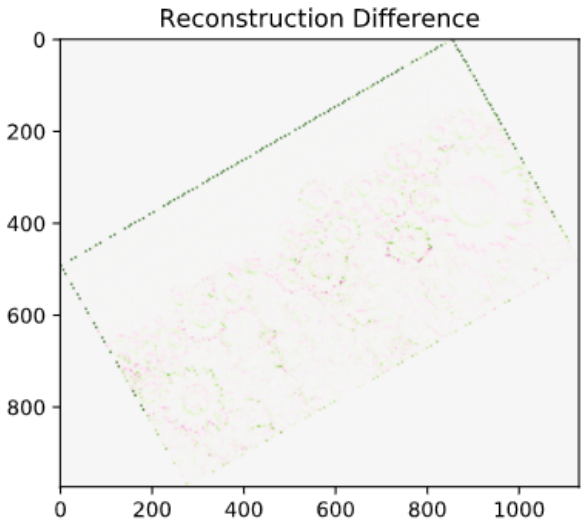
3.4 Feature Matching with OpenCV
Please do not attempt this question until the previous questions are completed; I would like you to try to get your system mostly working before trying a "professional" package.

Follow the OpenCV tutorial to implement feature matching and computing homographies (copy-pasting code is expected here).

FIGURE & DISCUSSION Generate an image like the one in the tutorial but for the two transformed sunflower images I have included. How does the performance (e.g., in terms of the number of features or accuracy of matches) of the OpenCV system compare to the system you implemented? How much faster (roughly, I do not need a precise number) is the OpenCV system compared to yours?



Feature Matches



Reconstruction Difference

The OpenCV algorithm performed very well in calculating the homography as shown by the reconstruction difference image. However, you can see that at the very edge of the reconstruction there is a single pixel wide line of errors where a gradient was probably handled wrong. The algorithm finished in 2 orders of magnitude less time than my algorithm as it finished in just about half a second whereas my algorithm took over a minute to complete. This means that the OpenCV algorithm is at least 100x faster.