Image Stitching

-Arnav Dhamija, Saumya Shah

Note: Instructions for running the code and a brief description of the folder structure are available in README.md

Code Documentation

1. **Corner Detector**

Harris Corner Detector was used to detect the feature points. cornerHarris function from cv2 library was used for the purpose. The function returns corner_detector.py contains the function corner_detector(img) which returns the corner response score of the feature points and suppresses the scores of all the points with the response score less than 1 percent of the maximum. Also, the corners detected within the 20 pixel margin near the borders of the images are suppressed. This was done because the descriptors generated for the points near the borders had many zeros whenever the window moved out of the image frame and thus, caused mismatches with the points near the border of the other image with many zeros in the descriptor vector. The output of the corner detector is as seen below:

Significant features selected by Harris Corner Detector

2. **Adaptive Non-Maximal Suppression**

anms.py file contains anms(cimg,max_pts) function which takes the output of the corner detector and the maximum number of desired points and returns the coordinates of the corners after suppression along with the suppression radius. The features after applying anms for max_pts = 500 is shown below:

Features selected after ANMS

3. **Feature Descriptors & Matching**

Feature Descriptors: feat_desc.py contains the feat_desc(img,x,y) function which takes the image and the coordinates of feature points after suppression as input. Descriptors are generated for each feature points. The descriptors were generated by following methods:

1. Feature Descriptor using image pixel intensity: It considers 64 patches of size 5*5 in the 40*40 window around each of the feature point and takes the maximum intensity of the pixel value in that patch. A vector of length 64 is obtained and it is then normalized to get the corresponding feature descriptor.
2. Feature Descriptor using gradient magnitudes: feat_desc_gradient.py in the variations folder contains the feat_desc(img,x,y) function which generates feature descriptors using gradient magnitudes. Instead of taking the maximum intensity from the patches, this method used the mean of the gradients in that patch. Maximum values of the gradient magnitude from the patches were also tried but this worked better.
3. Feature Descriptor using histogram of gradient orientations: Feat_desc_histogram.py in the variations folder contains th efeat_desc(img,x,y)

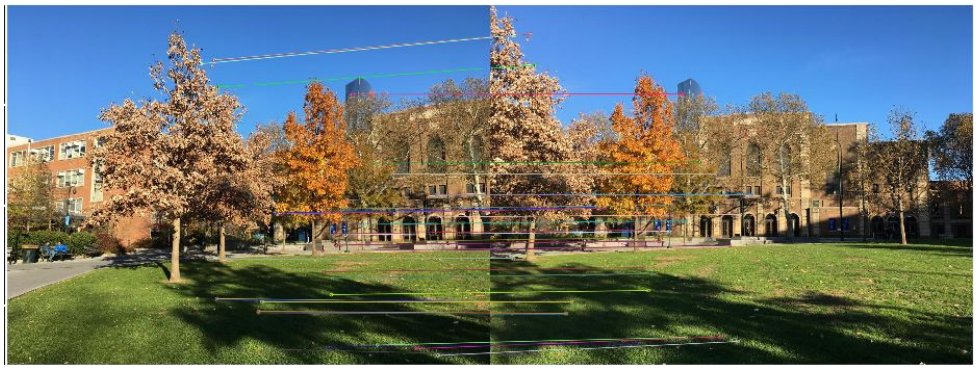
function which generates feature descriptors using the histogram of the gradient orientations. In this case, angles are divided into 8 bins and the number of pixels with orientations in each of the bins is counted for each window. Thus, 8*64 length vector obtained is then normalized to get the corresponding descriptor.

SIFT descriptors were also tested as a benchmark to compare our descriptors against. Out of these methods, the best results were obtained by using the feature descriptor generated using the image pixel intensity (a.).

Feature Matching:

1. Feature matching using FLANN: FLANN based feature matching was implemented. The ratio test was used as the decision rule for checking if the matches are relevant.
2. Feature matching using brute force search: The feat_match_bruteforce.py file in the variations folder contains the feat_match(descs1,descs2) function which uses this method. Feature matching was done by calculating L2 distance between the descriptors for all the feature points. The points which passed the ratio test were selected and the matches were returned.

The FLANN based match function resulted in matches which were not consistent when several iterations were performed over the same sets of descriptors. However, for a large number of descriptors it was faster than our own feature matching function, and the output was satisfactory. We used a ratio of 0.7 for the ratio test.

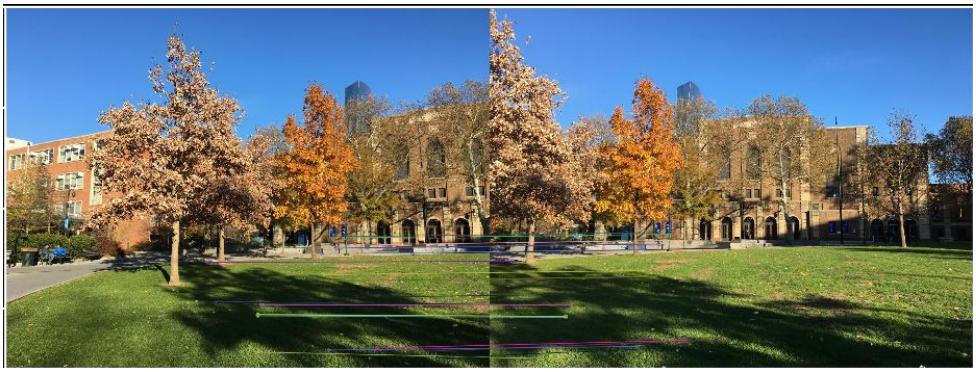


Features matched using the ratio test

4. RANSAC

The homography matrix for a set of 4 random points from the feature points was calculated and the inliers were counted. The decision rule for selecting the best H matrix not only considered the maximum number of inliers but also the sum of the distances of all the predicted points from the actual points. Thus, the H matrix with maximum inlier counts also minimized the sum of the distances which indirectly gave the least squared

distance solution for the given maximum number of inlier points. This improved the performance for the images with lesser number of feature matching.



Correspondences selected by RANSAC

For the below plots, blue points are the inliers, red points are the ones we found in the ANMS step after n iterations of selecting four points in RANSAC. As observed by the below plots, there is decreasing utility in increasing the number of iterations versus inliers discovered. However, with more iterations we are more likely to obtain a homography which minimises the total error. We chose to use 1000 iterations for RANSAC to capture as many inliers as possible.

2 iterations - 18 inliers

10 iterations - 72 inliers

50 iterations - 78 inliers

200 iterations - 82 inliers

1000 iterations - 88 inliers

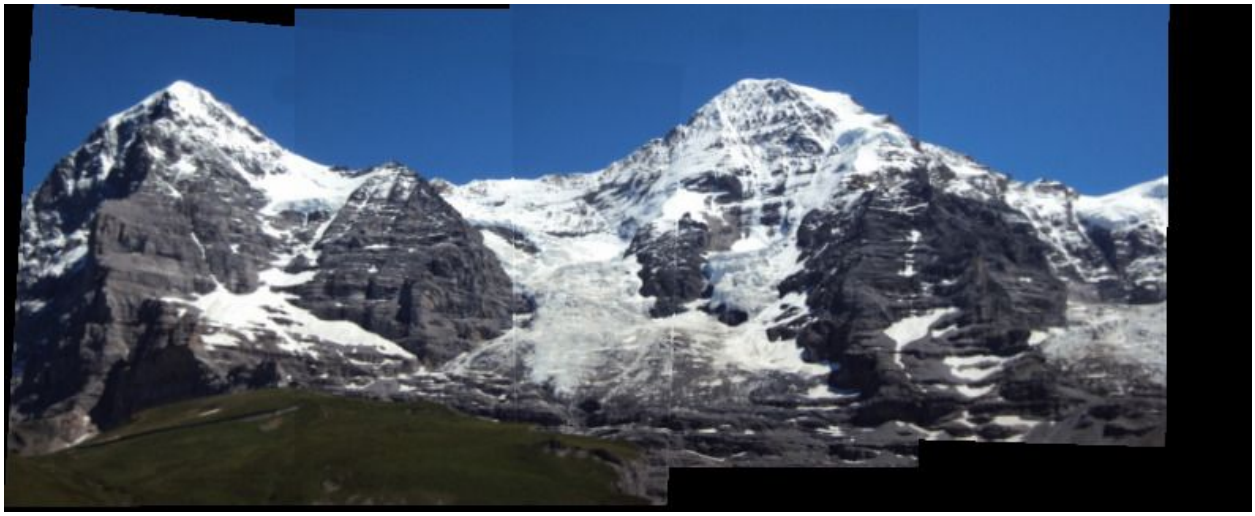
5. **Stitching**

   The RANSAC step gives us a homography matrix H which we can apply to the source (left or right) image to merge it with the target (middle) image. In our implementation we find a homography to warp the left image to the right image. Hence, we need to apply H_{left} to the left image and H_{right} to the right image and then appropriately position the middle image to obtain a panorama.

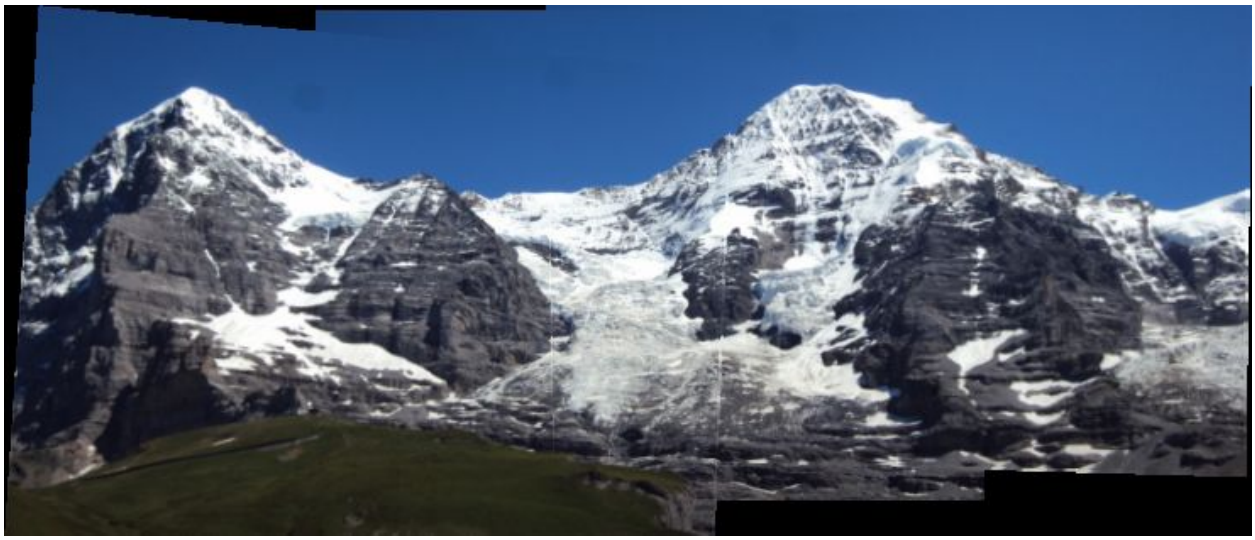
However, the H of the left image would normally translate the left image out of the image plane. To rectify this, we first compute the coordinates of all the corners of the left image by multiplying it with the H_{left} matrix. From these transformed coordinates we can find the minimum x and y coordinates of the corners. If these values are negative, we will need to translate the left image into the “canvas” by using a translation matrix of the form T=[[1 0 xmin],[0 1 ymin],[0 0 1]].

The final matrix we will need warp the left image by is matrix product of TH_{left}. Similarly we will need to adjust the middle image by warping it by T, and the right image by warping it by TH_{right}. The size of the canvas is determined by the position of the corners of the right image after applying TH_{right}. Now, we can stack the three images on top of the other to generate a panorama!

The images are well aligned. However, the seams between the images is distinctly apparent. Can we do better?

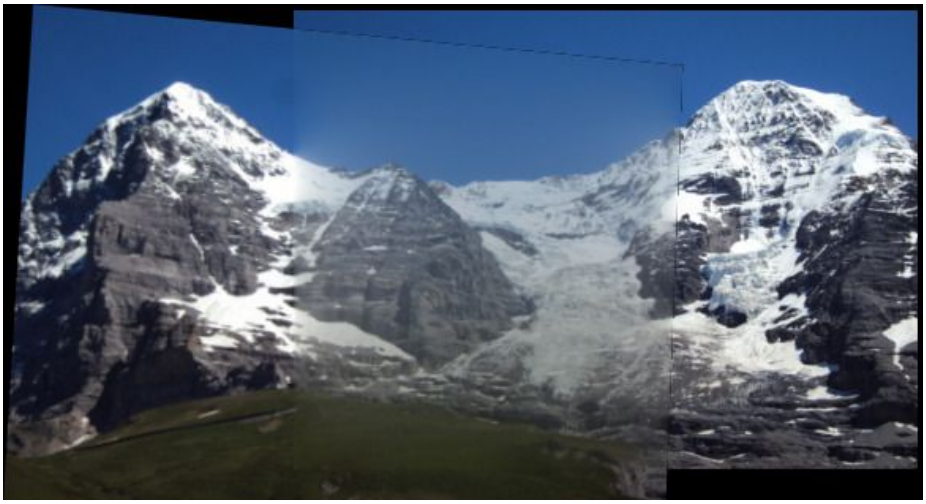


On first glance, **alpha blending** gives better results than just stacking the images. However, the seams still haven't gone away and some more have appeared in the panorama. Can we do better?



With **feathering** in the window of the overlap region between a pair of images, we can obtain a far more seamless transition between images. Artifacts of combining images to form the panorama are far less apparent. This is the method we have used in this project for the **final** output panorama.

We also tried other approaches such as Poisson blending, but as shown in the generated panorama below, the results were not quite as nice.



6. **Hacks And Tricks**

   This project would not have been successful without applying a few hacks to fix some of the issues we came across. Some of them are below:

   In our initial implementation, we were obtaining drastically different homography matrices from the RANSAC operation depending on which points were selected. We realised that since RANSAC randomly selects points, the homographies computed would be different on each call. Hence, we made a modification to RANSAC in which RANSAC would return a homography that 1) satisfied the most inliers 2) had the lowest error among all other points. This small tweak gave us far better homographies that worked reliably on every image we tested

   For the image feathering operations, we needed a mask of the transformed image to determine the overlapping regions. We did this by creating a mask of the image by creating a matrix of np.ones (numpy nD array with 1's) of the same dimensions of the image and warping it by the same transformations as the original image. Below is the warped mask for the left image.

The middle image would have a mask like this:

It is now trivial to apply a logical AND operation to find the overlapping region:

By using the width and position of this region, we can create another matrix for the feathering operation by using some meshgrid magic:

Mask 1 for the middle image

Mask 2 for the left image

Finally we multiply this by the middle and left images and add the results to get a smooth overlapping region

Applying Mask 1 on the middle image

Applying Mask 2 on the left image

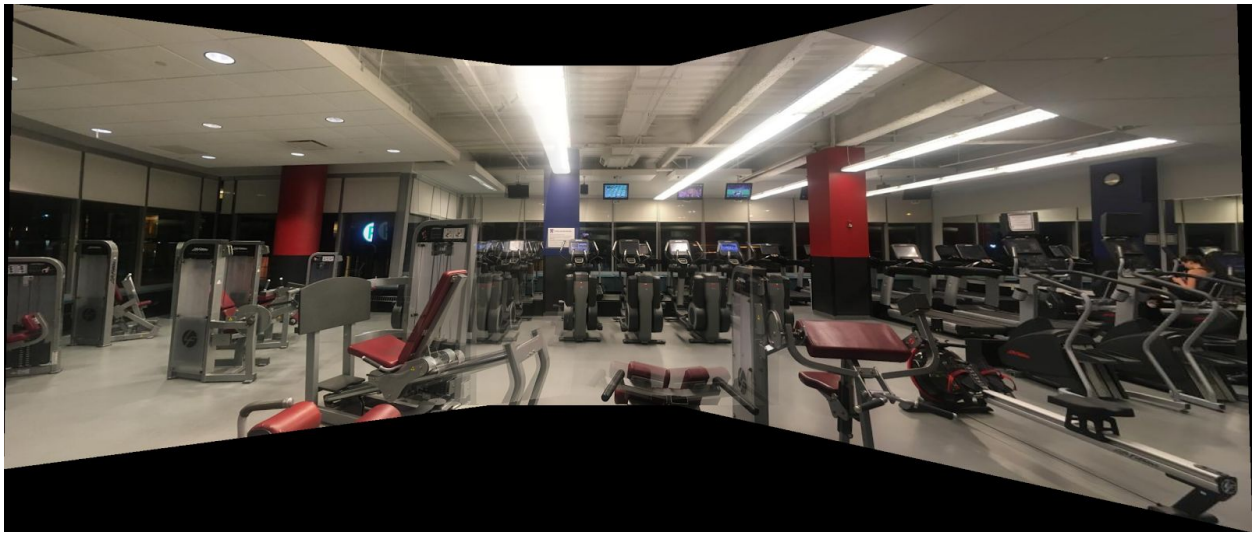
Adding the masked images

Results

All the below results can be found in their respective results/ folder which includes the outputs of all the intermediate. The left, middle, and right images used to create these panoramas can be found in the images/ folder.

To get started with simpler images, we chose this set of images from the Firewatch game, which had only translation involved.

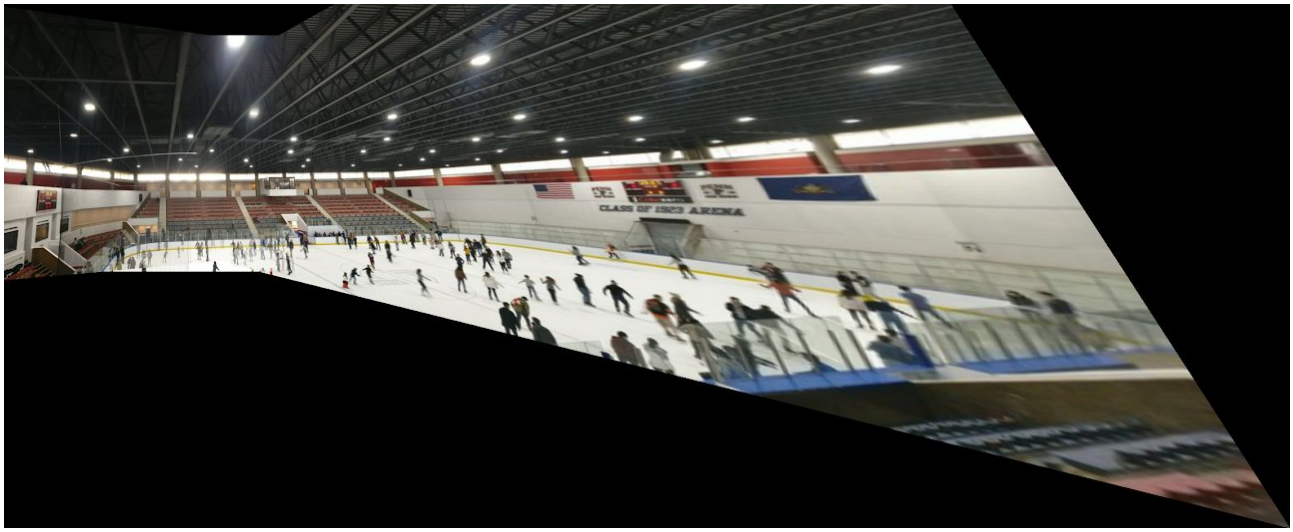
Next, we tried the standard mountain image



Panorama at the Pottruck gym

Low light panorama with darker features. Used for testing in the feathering operation.

Shoemaker Green/Franklin Field Panorama



This was the most challenging image which illustrates how well the blending and homography worked. The alignment of rails on the roof are indicative of the fact and no distinctive edges can be seen at the joints.