CS 5670: Computer Vision

Project 2 Report

29 February 2024

Prepared By:

1. Andrew Park, abp73
2. Amit Shanbhoug, ans254

Table of Contents

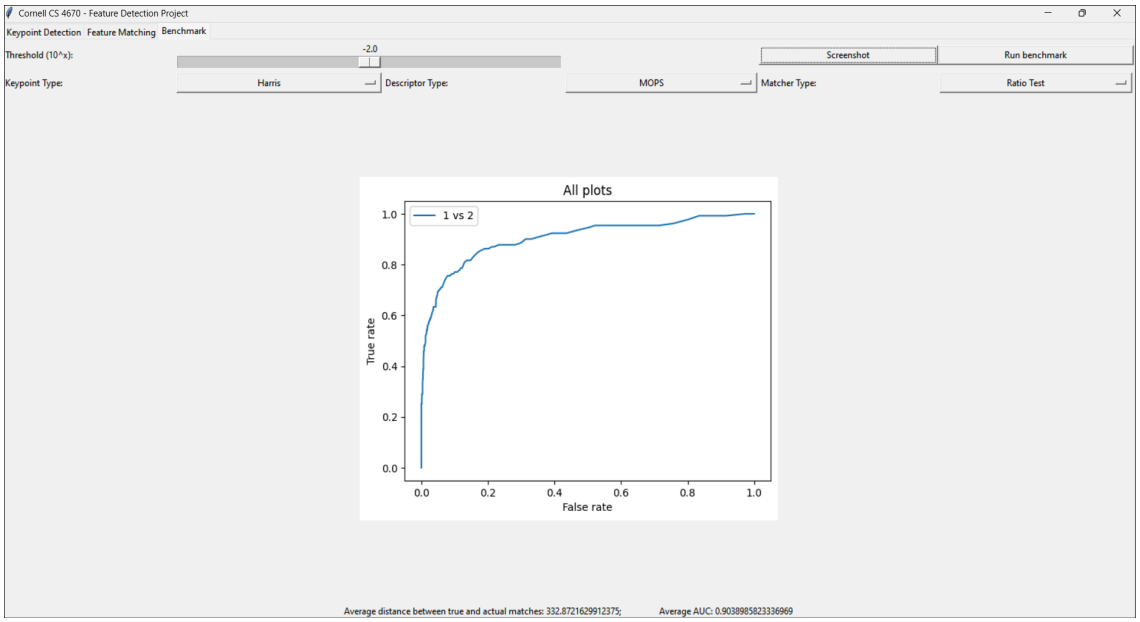
Note to the grader:

We computed the inbuilt screenshot as required and saved those screenshots under resources/Screenshots Taken and pushed it to the repository. To ensure that we are showing numbers computed (not manipulated), we used screenshots below that show the number + graph using a snipping tool.

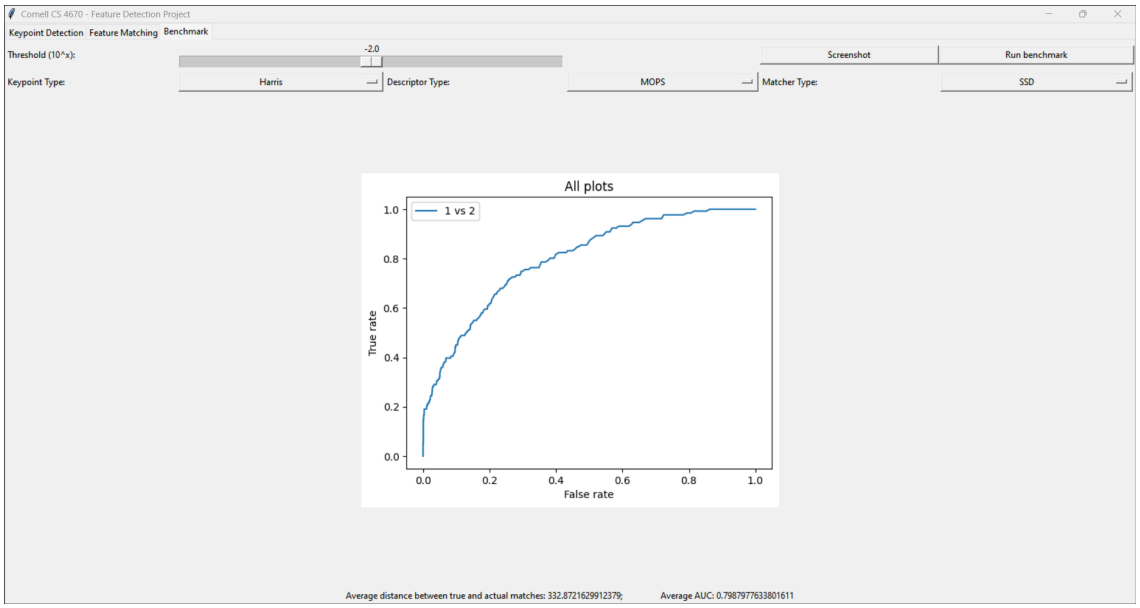
Benchmark in featuresUI.py

We can see a huge improvement in the AUC when we do MOPS with the Ratio Test v. MOPS with SSD. Our team believes that The ratio test performs a lot better than the SSD because the features on the rock has similar patterns and the ratio distance test enables greater matching.

MOPS with Ratio Test (Threshold:10^-2)
Result: Avg Distance: 332, Avg AUC: 0.9038

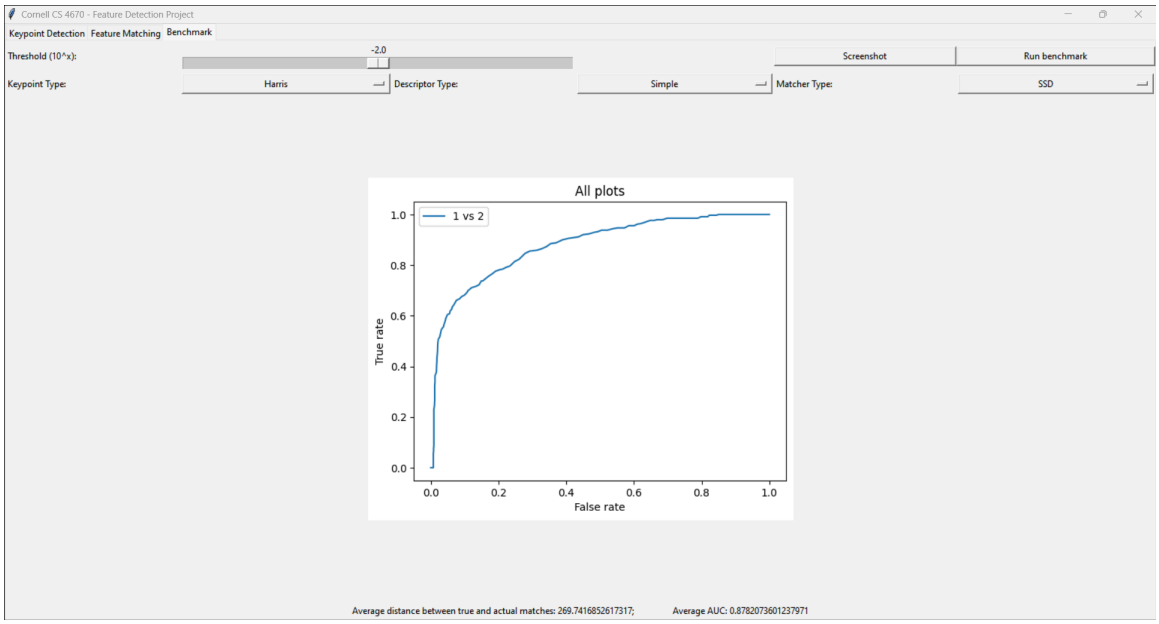


MOPS with SSD (Threshold:10^-2)
Result: Avg Distance: 332, Avg AUC: 0.7987

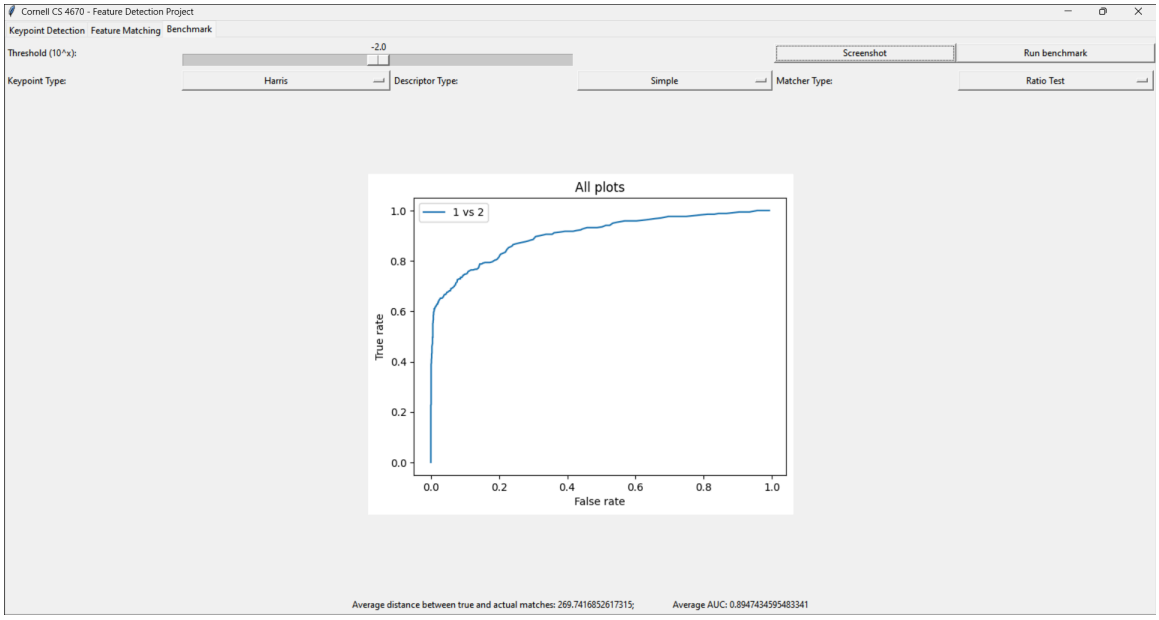


There is a minor improvement between Simple with SSD and Simple with the Ratio Test (~0.2), which is not as significant as the changes with the MOPS above.

Simple with SSD (Threshold:10^-2)
Result: Avg Distance: 269, Avg AUC: 0.8782

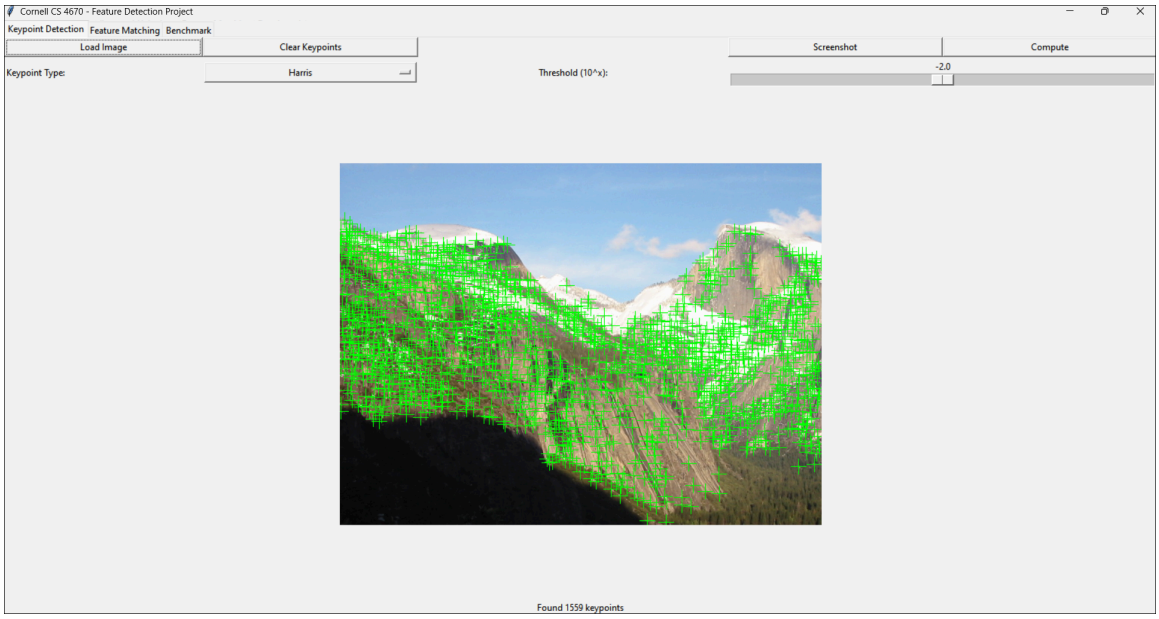


Simple with Ratio Test (Threshold:10^-2)
Result: Avg Distance: 269, Avg AUC: 0.8947

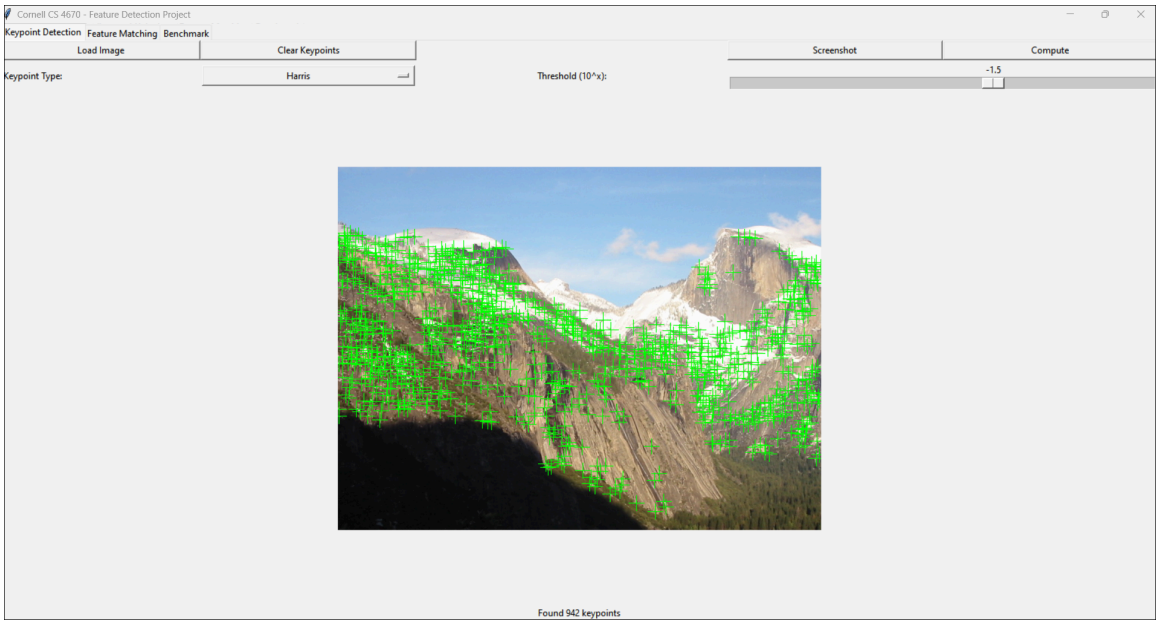


Harris Image on Yosemite1.jpg

Initially, without adjusting the threshold, there were 1559 keypoints found as shown below:



By adjusting the threshold to -1.5, the keypoints is lowered to 942 as shown below:



Interestingly some of the edge points between the mountain and the sky are not highlighted - that would have been incredibly useful. Here, we can observe that the highlighted regions are edges or ridges of the mountains, making it more distinct than points on the “face of the mountain”.

Feature Matching of own images

Provided 2 similar matches but different (notice the amount of sky and housing in the images). The feature matching computed 85 matches.

As the threshold is lowered to -3.2 from -2.0, the number of matches improves as shown below but there are more matches that can be shown that are not matching properly:

SIFT for scale invariant descriptor

For the scale invariant descriptor, our team tried to implement the method mentioned in the project guide, which was using various scales (Gaussian pyramid) and selecting the best descriptor by ANMS (adaptive non maximal suppression). However, the performance did not improve more than 15%, due to the lack of successful implementation.

Based on the lecture, our team came up with implementing a simple version of the SIFT, which is currently the state-of-art descriptor. Implementing SIFT was much more straightforward and robust, considering the superb performance that we will talk about in the following paragraphs.

The code can be found in the features_scale_invariant.py file, on the CustomFeatureDescriptor class.

How we made the descriptor scale invariant by SIFT and why it works

To achieve scale invariance, our team implemented a simple SIFT descriptor as mentioned above. This not only achieved scale invariance but also much higher performance in feature descriptions.

For the basic steps, we first did the preprocessing. We converted to a floating-point data type and normalized, as the basic feature descriptor did. Implemented this step to maintain the consistency in feature computation across images with different brightness. Next, we converted the image to grayscale. By doing this, we disregarded the color channel to focus on the intensity gradients. Lastly, for the basic steps, we calculated the gradient from each keypoints. By employing the Sobel operator, we determined the gradients along the x and y axes, essential for revealing the orientation and edge directions within the image.

```
# Compute Custom descriptors (extra credit)
class CustomFeatureDescriptor(FeatureDescriptor):

    def describeFeatures(self, image, keypoints):
        image = image.astype(np.float32)
        image /= 255.
        grayImage = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        orientationImage = np.zeros(grayImage.shape[:2],dtype=float)
        desc = np.zeros((len(keypoints),16*8))
        Ix = ndimage.sobel(grayImage,axis=1,mode='reflect')
        Iy = ndimage.sobel(grayImage,axis=0,mode='reflect')
        orientationImage  = np.rad2deg(np.arctan2(Iy,Ix))
```

Now are the steps we did for the SIFT descriptor. First, we feeded the keypoint value detected in the image. Then we constructed a 16x16 window around the keypoint based on the lecture notes. Next, we divided the window into 16 smaller, 4x4 sub regions to use it for the histogram. Inside each sub-region, we created an 8 bin orientation histogram from the magnitudes and orientations of the gradients. We also weighted the magnitude by a Gaussian window, giving more importance to gradients closer to the center keypoint. Finally, by concatenating these histograms from all 16 regions formed a 128-element feature

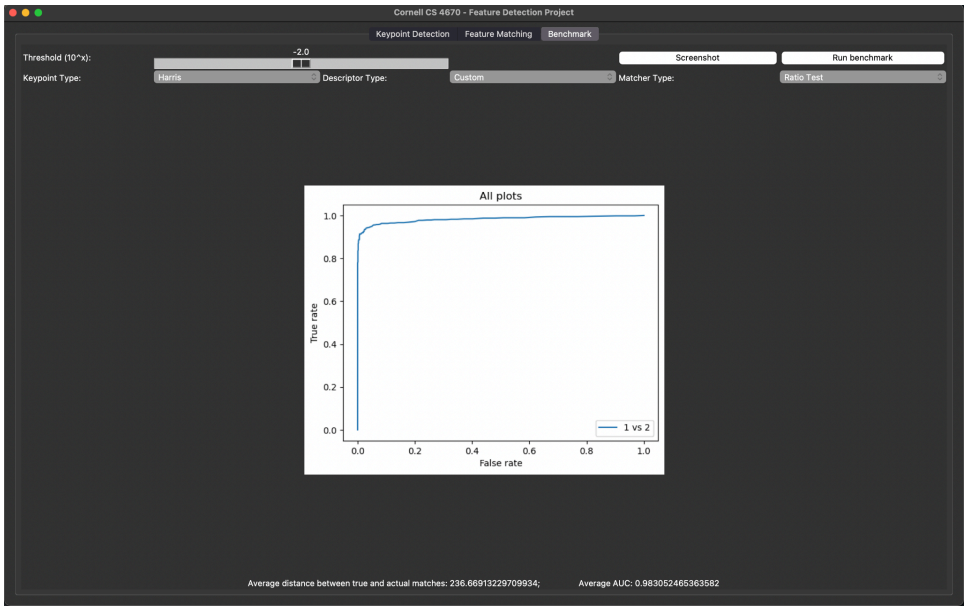
vector for each keypoint, which achieved scale invariant descriptor with superb performance. For the last step, we normalized the feature vector to unit length. In our insight, this process was may have been useful for dealing with varying light conditions.

```
#SIFT
for i,f in enumerate(keypoints):
    # Extract x, y coordinates of the feature point
    x,y = f.pt
    x = int(x)
    y = int(y)

    # Initialize a container for the histogram of gradient orientations
    contain = np.zeros((16,8))
    for outrow in range(4):
        for outcol in range(4):
            for inrow in range(4):
                for incol in range(4):
                    distcol = outcol*4 + incol # Calculate the column index for
                    distrow = outrow*4 + inrow # Calculate the row index for th

                    # Check if the indices are within the bounds of the image
                    if(y-7+distrow)<0 or (y-7+distrow)>grayImage.shape[0]-1 or
                        break # Skip this iteration if out of bounds

                    # Get orientation at this point
                    degree = orientationImage[y-7+distrow,x-7+distcol]

                    # Ensure the degree is within [0, 360) range
                    if(degree<0):
                        degree+=360

                    # Determine which bin the orientation falls into
                    degpart = int(degree//45)

                    # Increment the appropriate bin
                    contain[outrow*4+outcol,degpart] += 1

    contain = contain.reshape((1,128));
    stddev =np.std(contain)

    if stddev < 10**-5:
        contain = np.zeros((1,128))
    else:
        contain = (contain - np.mean(contain)) / stddev

    desc[i] = contain

return desc
```

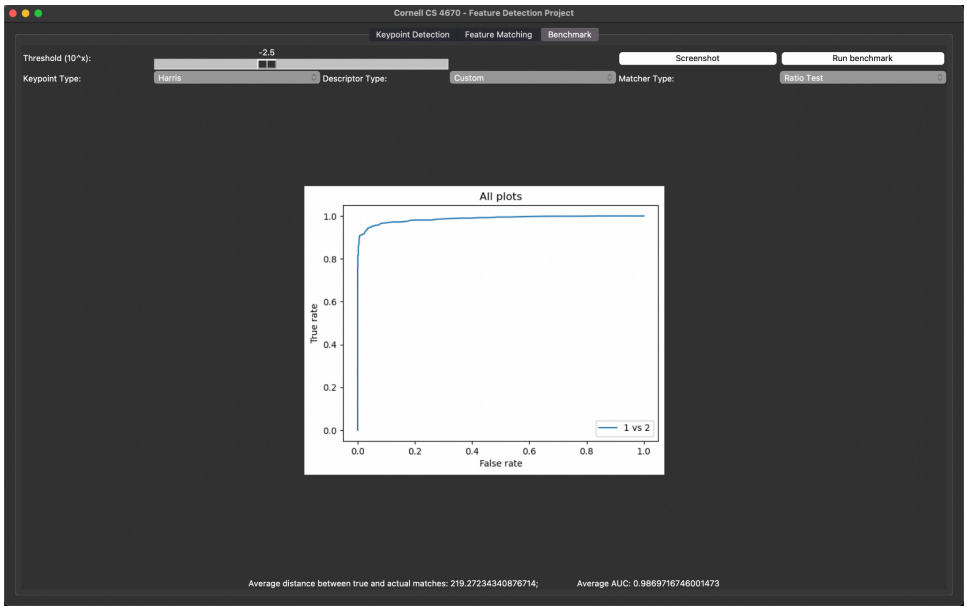
Our SIFT descriptor achieved scale invariance by combining features that do not get affected by changes in scale. Specifically, by building a histogram of gradients around each keypoint, and by examining a 16x16 window centered on the keypoint, subdivided into 16 blocks of 4x4 pixels, we successively captured local gradient information. This value is robust to changes in orientation and partially robust to changes in scale, making the code scale invariant.

ROC curve and AUC by our custom algorithm, SIFT (Yosemite dataset)

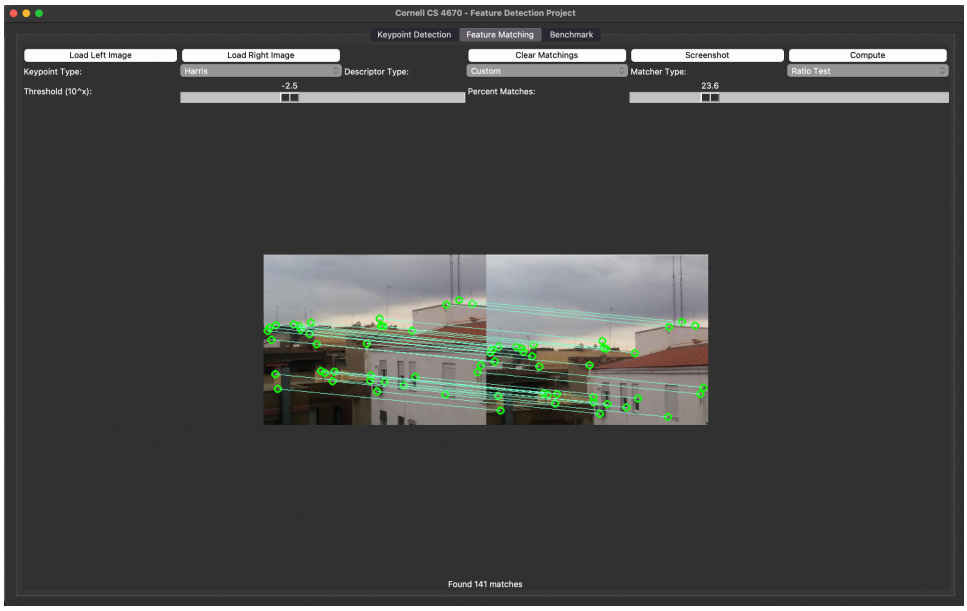
Custom Descriptor with Ratio Test (Threshold:10^-2)
Result: Avg Distance: 236, Avg AUC: 0.9830



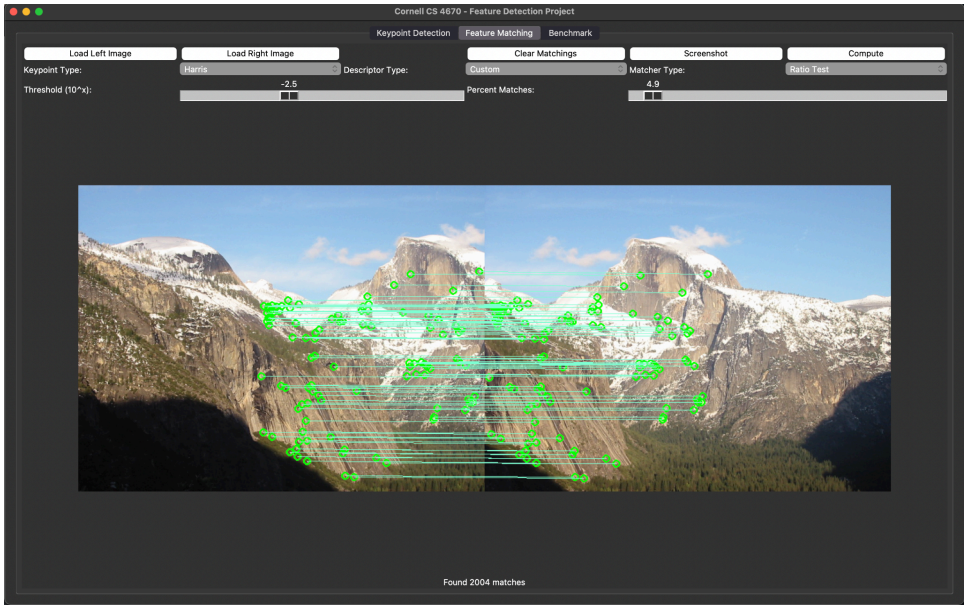
Custom Descriptor with Ratio Test (Threshold:10^-2.5) (Highest Performance)
Result: Avg Distance: 219, Avg AUC: 0.9869



Custom Image Feature Matching with Custom Descriptor (Threshold:10^-2.5) (23.6%)
Result: 141 matches



Custom Image Feature Matching with Custom Descriptor (Threshold:10^-2.5) (4.9%)
Result: 2004 matches



As the AUC value tells us, our SIFT descriptor matches robustly among the features. By examining each matches, our team was not able to find any missing points.

Changes to the algorithm and why they improve performance

Since our team was not able to use and customize MOPS descriptor, we had to change every code for MOPS and implement SIFT from scratch based on the lecture notes (code provided by link and image above).

By achieving not only scale but also rotation invariance (to some degree), the custom descriptor (Avg AUC: 0.9869) performed far better than the MOPS (Avg AUC: 0.9038, custom descriptor achieved 86.38% reduction in the area above the curve) or simple descriptor (Avg AUC: 0.8947, custom descriptor achieved 87.56% reduction in the area above the curve). This is due to the nature of MOPS being non-scale invariance unless it uses multi-scale detection, performing lower than SIFT.