ENSE 885AY

Application of Deep Learning in Computer Vision

Assignment A02

Local Feature Matching

Instructed by

Dr. Kin-Choong Yow

Student:

Marzieh Zamani

List of Sections and Sub-sections

1. Introduction

1.1. Overview (Key points from the assignment description) [1]

2. Student Code

2.1. Implantation of Harris Detector and ANMS (get_interest_points function)

2.2. Implantation of SIFT-like algorithm (get_features function)

2.3. Implantation of Feature Matching algorithm (match_features function)

3. Results and Discussion

3.1. Local Feature Matching Results for Provided Images

3.2. Effect of Scale Values on Correspondence Evaluation Results

Extra Works

References

1. Introduction

1.1. Overview (Key points from the assignment description) [1]

Assignment Subject:

Local Feature Matching

Assignment objectives:

Creating a local feature matching algorithm using following steps:

Steps to local feature matching between two images (image1 & image 2):

1. Detecting interest point in both images using Harris detector followed by adaptive non-maximal suppression (ANMS);
2. Describing local features of interest points in both images using a SIFT-like algorithm;
3. Comparing and matching interest points in both images using "ration test";

2. Student Code

2.1. Implantation of Harris Detector and ANMS (get_interest_points function)

First step is detecting interest point in both images using Harris detector followed by adaptive non-maximal suppression (ANMS). The Harris detector is implemented using algorithm 4.1 from textbook [2]. Then, the detected points are filtered using the ANMS algorithm from [2]. Both these steps are accomplished within the "get_interest_points" function:

```
x1, y1 = get_interest_points(image1, feature_width)

x2, y2 = get_interest_points(image2_bw, feature_width)
```

Implantation of Harris Corner Detector using algorithm 4.1 [2]:

1. Compute the horizontal and vertical derivatives of the image (Ix, Iy) by convolving the original image with derivatives of Gaussians (Horizontal and vertical Sobel filters with ksize = 3);

```
    # Step 1: Compute the horizontal and vertical derivatives of the image (Ix and Iy) using
Sobel filters
    Ix = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
    Iy = cv2.Sobel(image,cv2.CV_64F,0,1,ksize=3)
```

2. Compute the three images corresponding to the outer products of these gradients (Ixx, Iyy, Ixy);

```
    #Step 2: Compute  the outer products of gradients (Ixx, Iyy, Ixy)
    Ixx = Ix * Ix
    Iyy = Iy * Iy
    Ixy = Ix * Iy
```

3. Convolve each of these images with a larger Gaussian (ksize = 5) to obtain updated gradients (Ixx, Iyy, Ixy);

```
    #Step 3: Convolve each of these images with a larger Gaussian (ksize = 5)
    Ixx = cv2.filter2D(Ixx, cv2.CV_64F, cv2.getGaussianKernel(ksize=5, sigma=1))
    Ixy = cv2.filter2D(Ixy, cv2.CV_64F, cv2.getGaussianKernel(ksize=5, sigma=1))
    Iyy = cv2.filter2D(Iyy, cv2.CV_64F, cv2.getGaussianKernel(ksize=5, sigma=1))
```

4. Compute a scalar interest measure using the following formula:

$$A = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \rightarrow Response\ Strength\ of\ Interest: R = \det(A) - \alpha trace(A)^2$$

```
    #Step 4: Compute a scalar interest measure (R)
    Determinant = Ixx*Iyy - Ixy*Ixy
    Trace = Ixx + Iyy
    R = Determinant - alpha*Trace**2
```

5. Find the local maxima above a threshold and return the results as matrix XYR with 3 columns of x, y, and R values of detected interest points;

$$threshold = 0.01 * \max(R)$$

```
    #Step 5: Find the local maxima above a threshold and return the results as matrix XYR
    # Define threshold
    threshold = 0.01*np.amax(R)
```

```
    # Result matrix
    XYR = []

    for i in range(feature_width, R.shape[0] - feature_width):
        for j in range(feature_width, R.shape[1] - feature_width):

            # Find if the point is a local maxima above a threshold
            if R[i,j] > threshold:
                x = int(j)
                y = int(i)
                res_strength = R[i, j]
                # Save results
                XYR.append([x, y, res_strength])
```

Implantation of Adaptive Non-Maximal Suppression (ANMS) [1,2]:

1. Sort all points (XYR matrix) by the response strength (R column), from largest to smallest responses;

```
    # Step 1: Sort the XYR matrix by the R column, from largest to smallest R
    XYR = sorted(XYR, key=lambda x: x[2], reverse=True)
    XYR = np.asarray(XYR)
```

2. Iterate through the list and compute the minimum distance between each interest point(k) (k = 0 : N-1) and all other points(kn) (kn = 0 : k-1) ahead of it in the list (i.e. the points with greater response strength). (The first entry in the list is the global maximum, and will be chosen.)
3. Save each point and its minimum distance as a row of matrix XYD with 3 columns of x, y, and minimum distance values for the point;

```
    # Step 2: Iterate through the list and compute the min. distance between each inteest poi
nt and all other points ahead
    # x, y values of points
    x = XYR[:,0]
    y = XYR[:,1]

    # Max. number of points
    Nmax = 2500

    # Result matrix
    XYD = []
```

```
    for k in range(XYR.shape[0]):
        min_distance = 2*image.shape[0] # maximum possible distance x sqrt(2)

        for kn in range(k):
            dx = x[k] - x[kn]
            dy = y[k] - y[kn]

            # Compute distance between point(k) and neibour point(kn)
            distance = np.sqrt(dx**2 + dy**2)

            # Obtain min. distance
            if distance < min_distance:
                min_distance = distance

        # Step 3: Save results
        XYD.append([x[k], y[k], min_distance])
```

4. Sort all points (XYD matrix) by the minimum distance (D column), from largest to smallest distances;

```
    # Step 4: Sort the XYD matrix by the D column (min. distance), from largest to smallest D
    XYD = sorted(XYD, key=lambda x: x[2], reverse=True)
    XYD = np.asarray(XYD)
```

5. Return the top Nmax (Nmax = 2500) points as the final detected points;

```
    # Step 5: Extract the top Nmax points as the final detected points
    XYD = XYD[0:Nmax,:]

    # Return the x, y coordinates of the final detected points
    x = XYD[:,0].astype(int)
    y = XYD[:,1].astype(int)
```

2.2. Implantation of SIFT-like algorithm (get_features function)

The next step is describing local features of interest points in both images using a SIFT-like algorithm. This is accomplished within the “get_features” function:

image1_features = get_features(image1_bw, x1, y1, feature_width, scales1)

image2_features = get_features(image2_bw, x2, y2, feature_width, scales2)

Implantation of SIFT-like algorithm [1,2]:

1. Compute the horizontal and vertical derivatives of the image (Ix, Iy) by convolving with the horizontal and vertical Sobel filters (ksize = 3);

```
# Step 1: Compute the horizontal and vertical derivatives of the image (Ix, Iy)
Ix = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
Iy = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)
```

2. Compute magnitudes (I_magnitudes) and angles (I_angles) and of the image using Ix and Iy through following relations:

$$I\_magnitudes = \sqrt{I_x^2 + I_y^2}; \qquad I\_angles = \tan^{-1}(Iy, Ix)$$

```
# Step 2: Compute magnitudes (I_magnitudes) and angles (I_angles) and of the image
I_magnitudes = np.sqrt(Ix ** 2 + Iy ** 2)
I_angles = np.arctan2(Iy, Ix)
```

3. Iterate through interest points(k) (k = 0 : N-1)

   For each point(k):

   Extract the 16x16 window (point(k) is in the center of this window) from I_magnitudes (magnitudesWin16x16) and I_angles (anglesWin16x16)

```
# Step 3: Iterate through interest points(k)
for k in range(x.shape[0]):
    # Extract the 16x16 window (point(k) is in the center of this window)
    xc = int(x[k])
    yc = int(y[k])

    fw_by2 = int(feature_width/2)
    x_start = max(xc - fw_by2, 0)
    x_end = min(xc + fw_by2, image.shape[1])
    y_start = max(yc - fw_by2, 0)
    y_end = min(yc + fw_by2, image.shape[0])

    # Obtain magnitudesWin16x16) & anglesWin16x16
    magnitudesWin16x16 = I_magnitudes[y_start:y_end, x_start:x_end]
    anglesWin16x16 = I_angles[y_start:y_end, x_start:x_end]
```

4. Divide the 16x16 windows to sixteen 4x4 windows (magnitudesWin4x4 , anglesWin4x4);

   For each 4x4 window (magnitudesWin4x4 , anglesWin4x4):

Compute 8-bin histogram of anglesWin4x4 weighted by magnitudesWin4x4 (hist_vector size = 8);

```
        # Result matrix
        siftFeatures_vector = []

        # Step 4: Divide the 16x16 windows to 16*4x4 windows (magnitudesWin4x4 , anglesWin4x4
);
        fw_by4 = int(feature_width/4)
        for i in range(0, feature_width, 4):
            for j in range(0, feature_width, 4):
                # Obtain magnitudesWin4x4 & anglesWin4x4
                magnitudesWin4x4 = magnitudesWin16x16[i:i + fw_by4, j:j + fw_by4]
                anglesWin4x4 = anglesWin16x16[i:i + fw_by4, j:j + fw_by4]

                # Compute 8-bin histogram of anglesWin4x4 weighted by magnitudesWin4x4
                hist_vector, _ = np.histogram(anglesWin4x4, bins=8, range=(-
np.pi, np.pi), weights=magnitudesWin4x4)
                siftFeatures_vector.extend(hist_vector)
```

5. Arrange and save 16 hist_vector as the siftFeatures_vector of point(k) (siftFeatures_vector size = 128);

   Normalize siftFeatures_vector;

   Arrange and save N siftFeatures_vector as fv matrix (fv size = Nx128)

```
        # Step 5: Arrange and save 16 hist_vector as the siftFeatures_vector of point(k)
        siftFeatures_vector = np.array(siftFeatures_vector)

        # Normalize siftFeatures_vector;
        siftFeatures_vector = siftFeatures_vector/np.sqrt(np.sum(siftFeatures_vector**2))

        # Arrange and save N siftFeatures_vector as fv matrix
        fv.append(siftFeatures_vector)
```

2.3. Implantation of Feature Matching algorithm (match_features function)

The next step is comparing and matching features in both images using “ration test” which is accomplished within the “match_features” function:

matches, confidences = match_features(image1_features, image2_features, x1, y1, x2, y2)

Implantation of Feature Matching algorithm [1,2]:

Iterate through features1(k1) (k1 = 0 : N-1)
For each feature1(k1):

1. Iterate through features2(k2) (k2 = 0 : N-1)

   For each feature2(k2):

2. Compute Euclidean distance between feature1(k1) and feature2(k2) and save as fv_distance(k1,k2)

```
    # Step 1.Iterate through features1(k1)
    for k1 in range(features1.shape[0]):

        # Step 2.Iterate through features2(k2)
        for k2 in range(features2.shape[0]):
            feature1 = features1[[k1],:]
            feature2 = features2[[k2],:]

            # Step 3.Compute Euclidean distance between feature1(k1) and feature2(k2)
            feature_distance = np.sqrt(np.sum((feature1 - feature2)**2))
            fv_distance[k1,k2] = feature_distance
```

3. For each row k1 of fv_distance matrix, obtain the indexes of first and second minimum distances (between feature1(k1) and feature2(k2)) as min_ind_1 and min_ind_2 and compute corresponding distances and their ratio

feat_dis_min1 = first minimum distance between feature1(k1) and feature2(min_ind_1)

feat_dis_min2 = second minimum distance between feature1(k1) and feature2(min_ind_2)

```
        # Step 4.Obtain the indexes of first and second minimum distances
        feature2_sorted_index = np.argsort(fv_distance[k1,:])
        min_ind_1 = feature2_sorted_index[0]
        min_ind_2 = feature2_sorted_index[1]
        feat_dis_min1 = fv_distance[k1,min_ind_1]
        feat_dis_min2 = fv_distance[k1,min_ind_2]
```

4. Compute the ratio between first and second minimum distances (between feature1(k1) and feature2(k2))

ratio = feat_dis_min1/feat_dis_min2

```
        # Step 5.Compute the ratio between first and second minimum distances
        ratio = feat_dis_min1/feat_dis_min2
```

5. Accept feature1(k1) and feature2(min_ind_1) as matching features if ratio < 0.8

```
        # Step 6.Accept feature1(k1) and feature2(min_ind_1) as matching features if ratio <
0.8
        if ratio < 0.8:  #Change to 0.9 while running Mount Rushmore
            feature1_index.append(k1)
            feature2_index.append(min_ind_1)
            feature1_2_distance.append(feat_dis_min1)
```

6. Arrange and save matches as matches matrix of two columns of feature1_index and feature2_index
   Arrange and save their corresponding distances confidences matrix of one column of feature1_2_distance

```
    # Step 7.Arrange and save matches and confidences
    matches = np.stack((np.asarray(feature1_index), np.asarray(feature2_index)), axis = -1)
    confidences = np.asarray(feature1_2_distance)
```

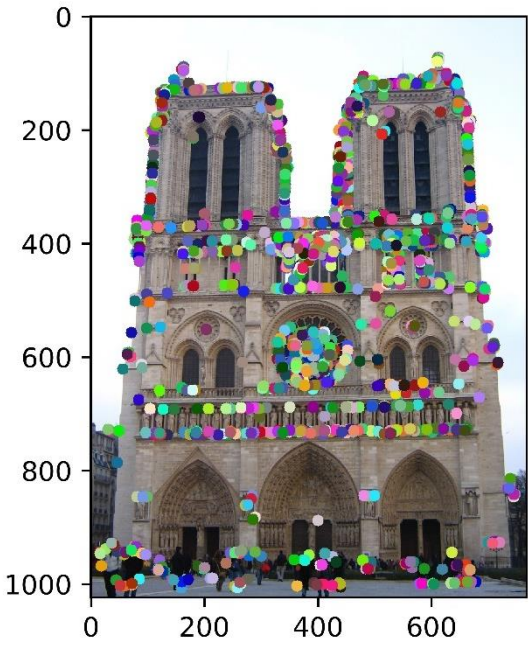
3. Results and Discussion

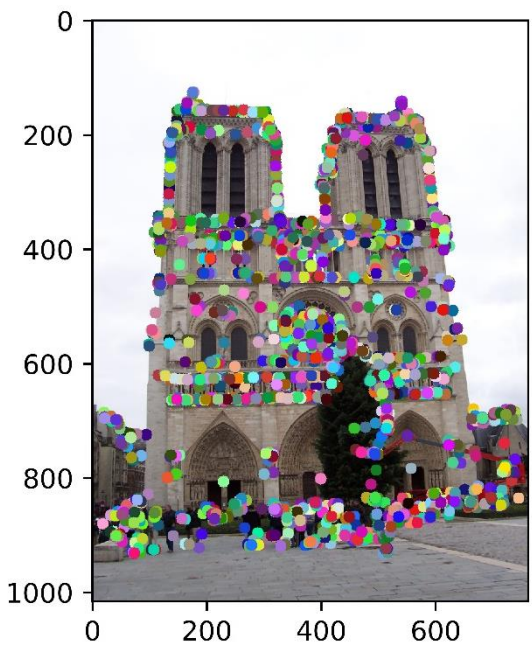
3.1. Best Local Feature Matching Results for Provided Images

This section presents the results of local feature matching for the image pairs provided as input data. Matching ratio and scale values have been tuned to obtain best accuracies.

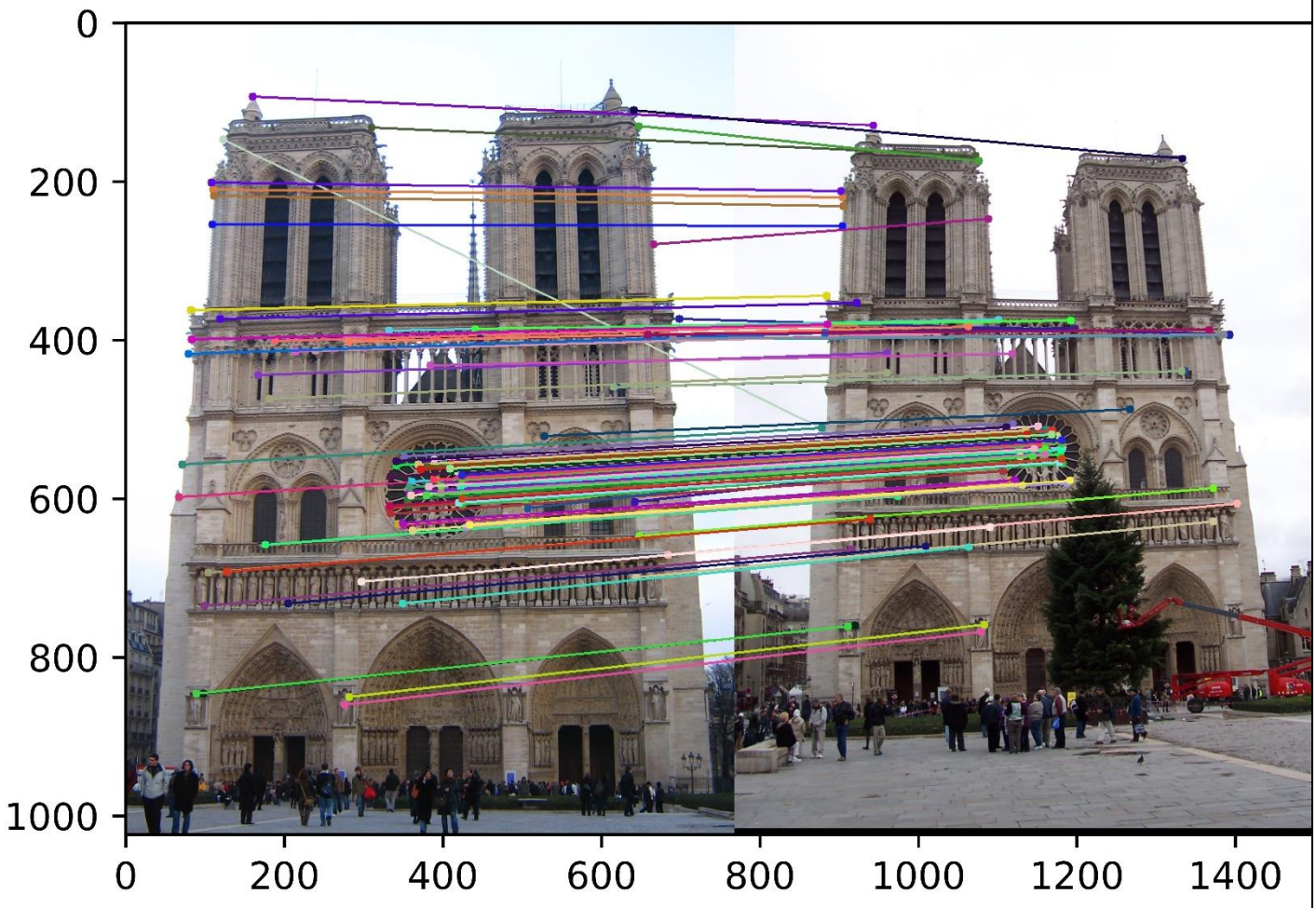
Table 1. Best Local Feature Matching Results for Notre Dame Images @ scale = 0.5 & matching ratio = 0.8

Interest Points (Corners) in Image1 & Image2





Corresponding Features between Image1 & Image2



Correspondence Evaluation Results for Image1 & Image2

Found matches: 100/100 required matches

Accuracy = 0.90

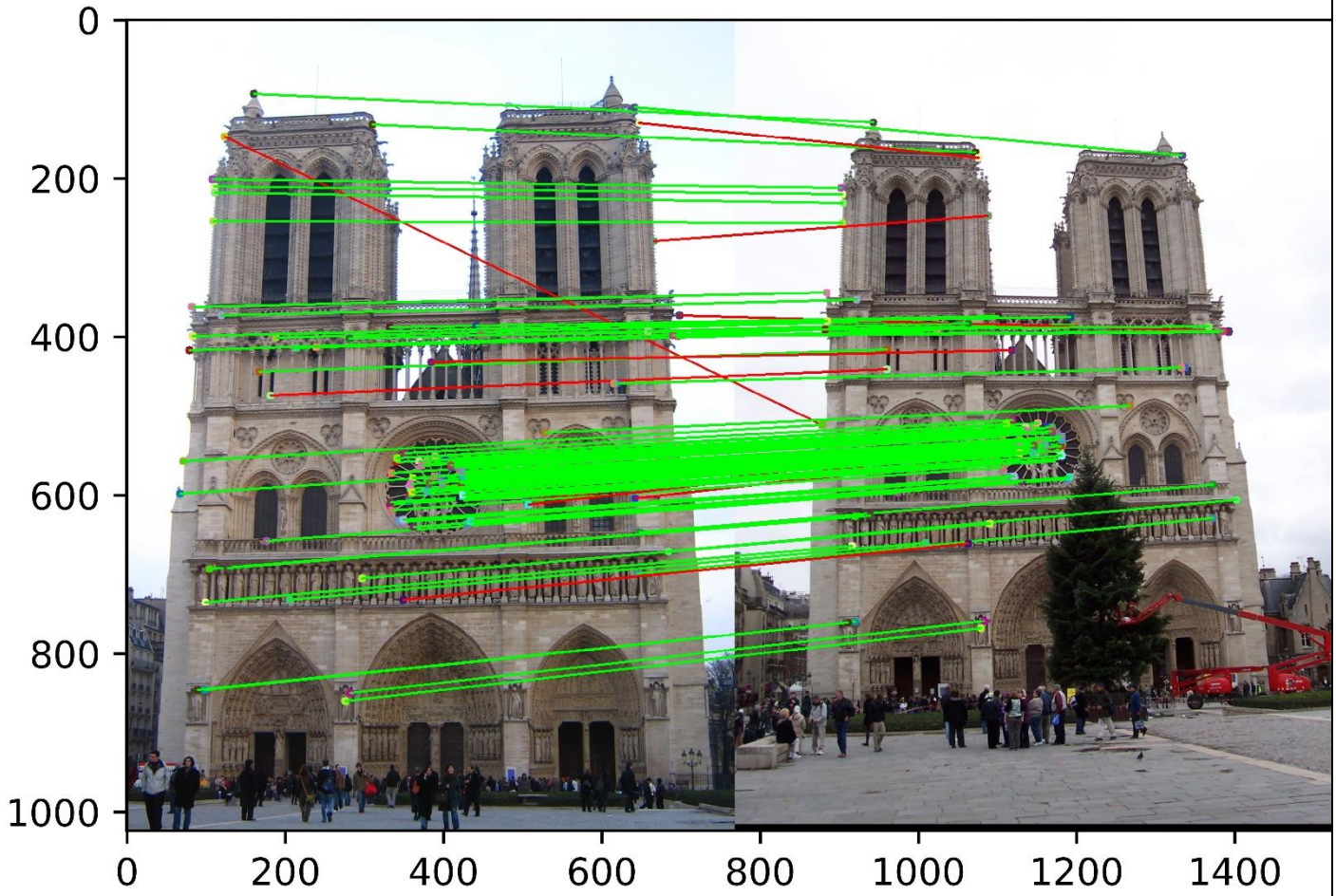
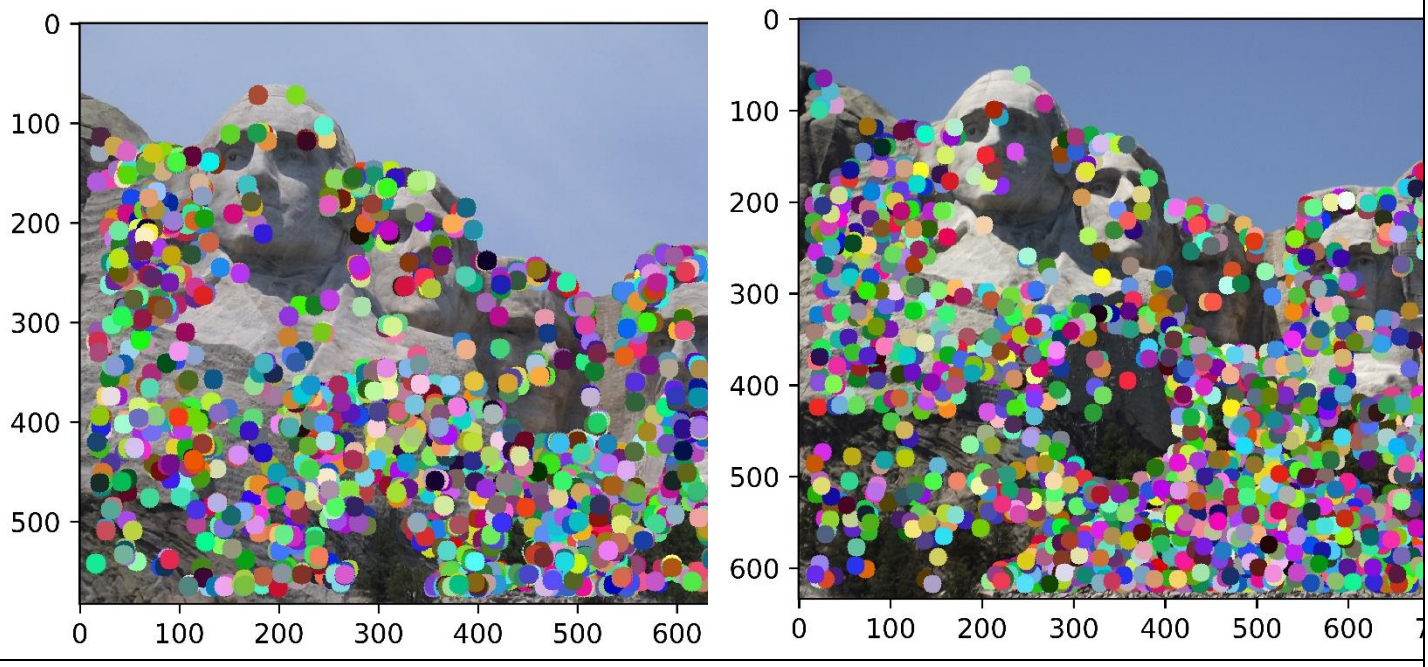
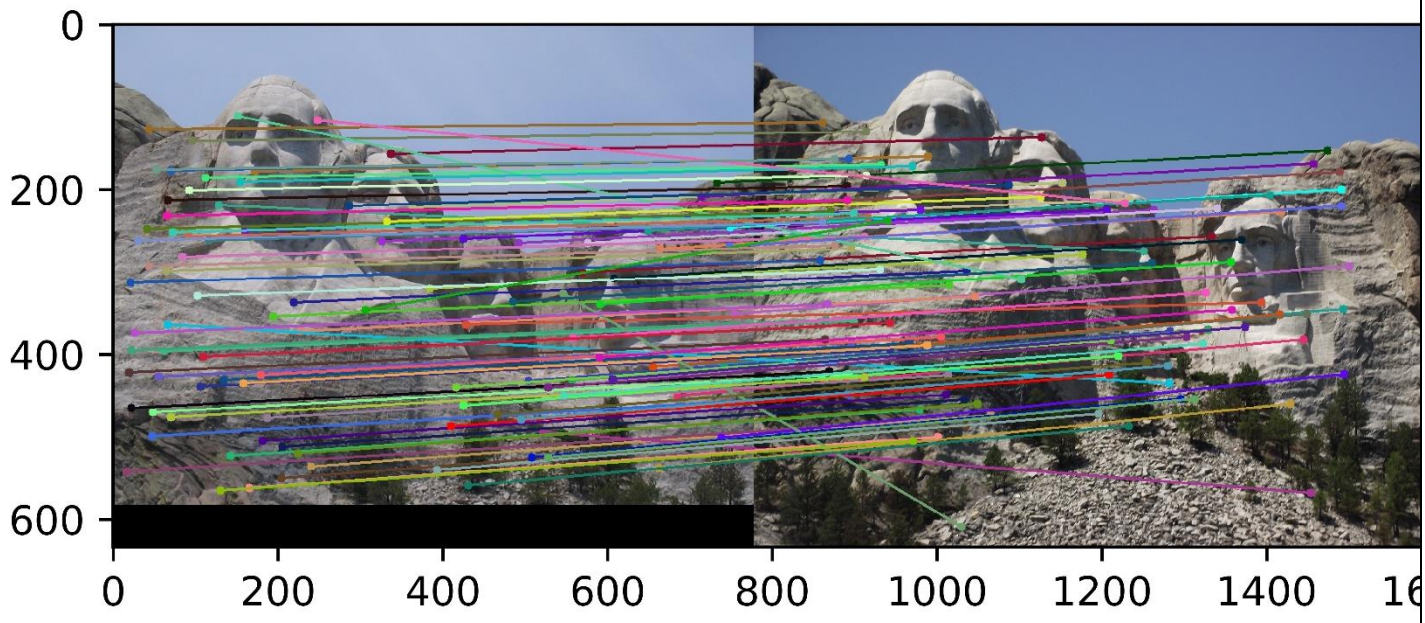


Table 2. Best Local Feature Matching Results for Mount Rushmore Images @ scale = 0.3 & matching ratio = 0.9

Interest Points (Corners) in Image1 & Image2



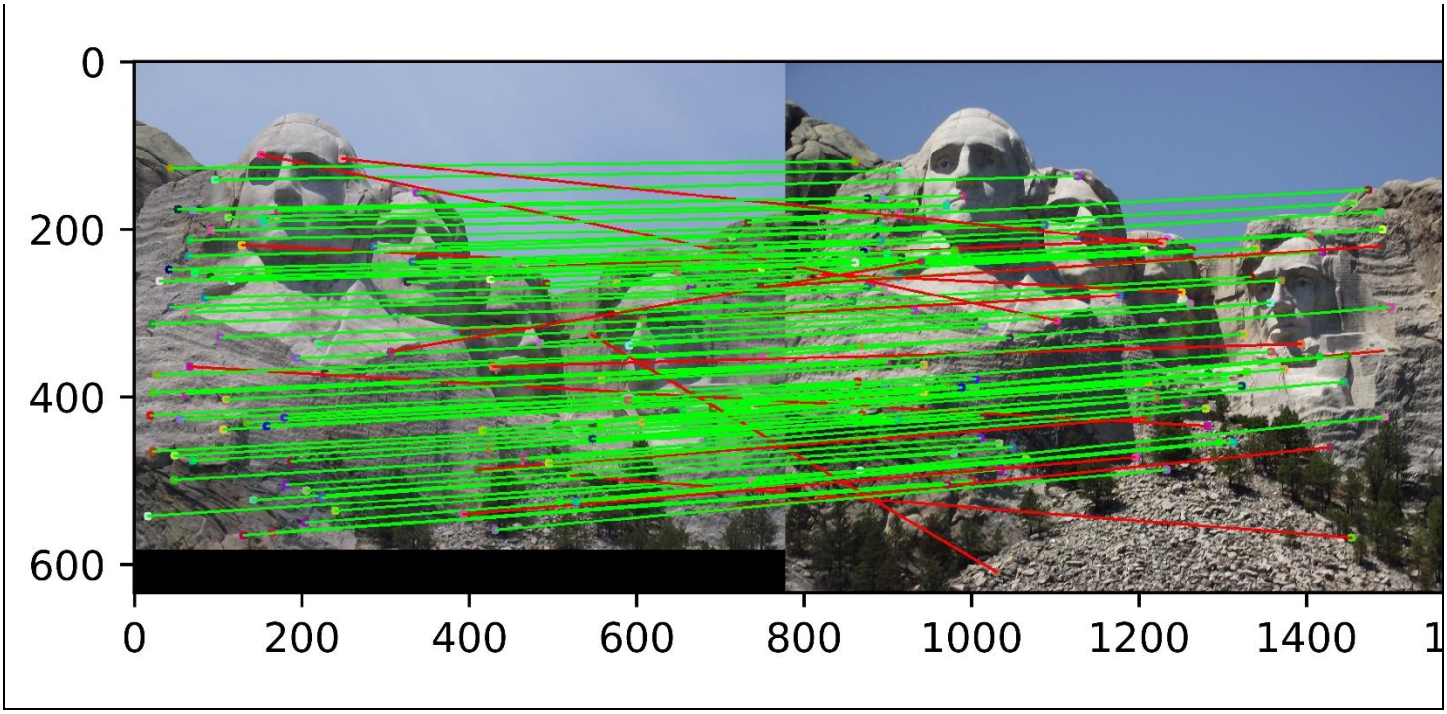
Corresponding Features between Image1 & Image2



Correspondence Evaluation Results for Image1 & Image2

Found matches: 100/100 required matches

Accuracy = 0.85



3.2. Effect of Scale Values on Correspondence Evaluation Results

In order to analyse the effect of matching ratios and scale values on correspondence evaluation results, different values were tested.

Remarks on the effect of scale values:

- For Notre Dame images, best accuracy was 90% obtained at scale = 0.5 and matching ratio = 0.8;
- For Mount Rushmore images, best accuracy was 85% obtained at scale = 0.3 and matching ratio = 0.9;

Table 3. Effect of cut-off frequency on the low/high frequency perception

	Correspondence Evaluation Results			
Image	**Matching Ratio**	**Scale = 0.3**	**Scale = 0.5**	**Scale = 0.7**
Notre Dame	0.8	Found matches: 61/100 Accuracy:	***Found matches:** **100/100** **Accuracy:**	Found matches: 100/100 Accuracy:

		0.55	**0.90**	0.85
Mount Rushmore	0.9	***Found matches:** **100/100** **Accuracy:** **0.85**	Found matches: 100/100 Accuracy: 0.72	Found matches: 100/100 Accuracy: 0.50

Extra Works

Following tasks were done outside assignment requirements:

- Testing different scales and matching ratio. The best results are presented in section 3.1. The correspondence evaluation results for all scales are presented in table 4.

References

[1] Assignment 01 description by Dr. Kin-Choong Yow
[2] Szeliski, R. (2010). Computer vision: algorithms and applications. Springer Science & Business Media.