CS682 Computer Vision
Write-up: P3 Matching Pipeline

Mithilaesh Jayakumar
G01206238

P3.1.1 Scaling and Rotating Features: Concepts

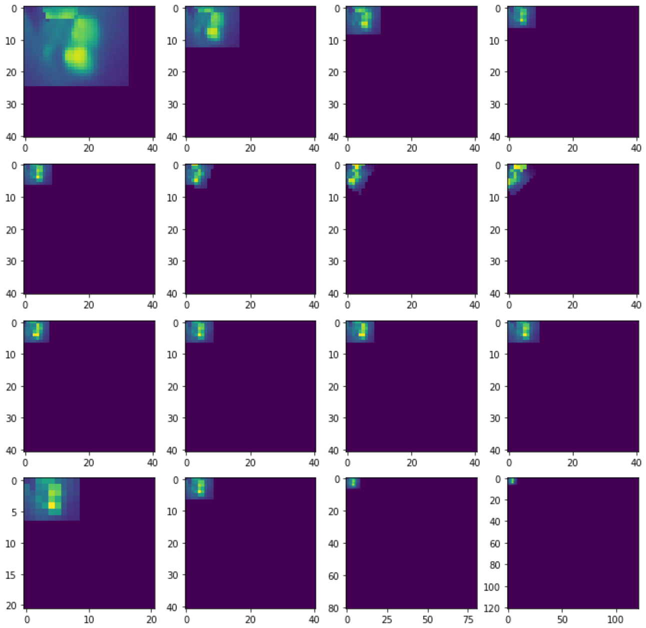
The homography matrix for the given scenario is given by trans_matrix below

```python
translate_origin = np.array([[1, 0, -feature_center_x], [0, 1, -feature_center_y], [0, 0, 1]])
unscale_matrix = np.array([[(1/feature_radius), 0, 0], [0, (1/feature_radius), 0], [0, 0, 1]])
unrotate_matrix = np.array([[np.cos(feature_orientation), np.sin(feature_orientation), 0],
                            [-np.sin(feature_orientation), np.cos(feature_orientation), 0],
                            [0, 0, 1]])

trans_matrix = np.linalg.inv(unrotate_matrix @ unscale_matrix @ translate_origin)
```

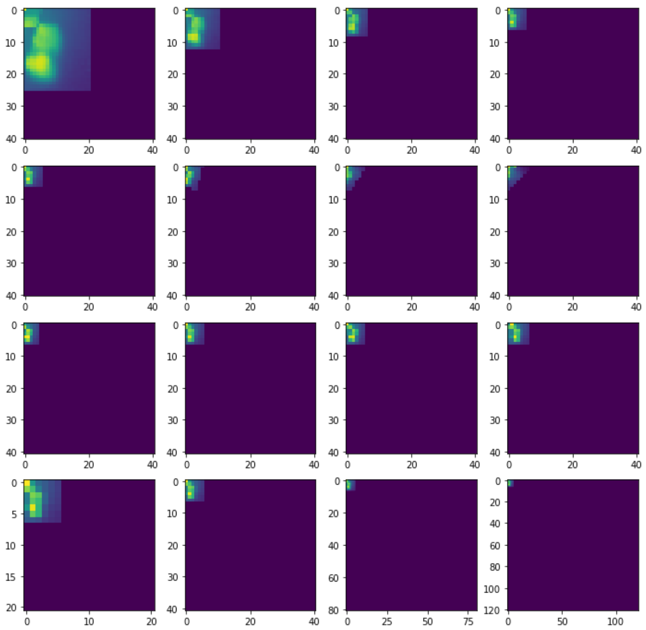
P3.1.2 Scaling and Rotating Features: Implementation

The below is the image generated for the two given feature co-ordinates using the H matrix above.

base_center_x = 800
base_center_y = 600



base_center_x = 500
base_center_y = 640



P3.2 Computing Homographies from Matches
P3.2.1 Computing Homographies from Perfect Matches

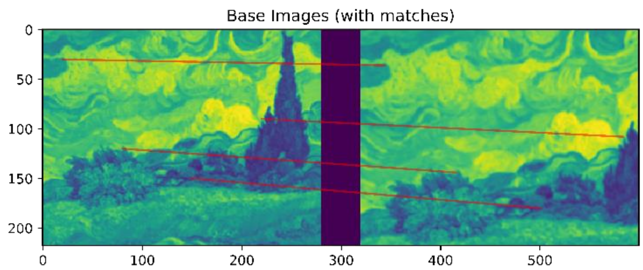
The below is the function used to compute the homography

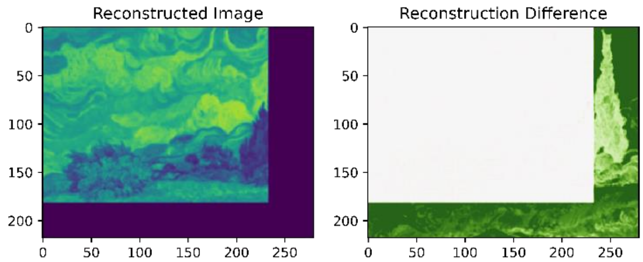
```python
def solve_homography(corrs):
    aList = []
    corrs = np.matrix(corrs)
    for corr in corrs:
        p1 = np.matrix([corr.item(0), corr.item(1), 1])
        p2 = np.matrix([corr.item(2), corr.item(3), 1])
        a2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2),
              p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1) * p1.item(2)]
        a1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2), 0, 0, 0,
              p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0) * p1.item(2)]
        aList.append(a1)
        aList.append(a2)
    matrixA = np.matrix(aList)
    u, s, v = np.linalg.svd(matrixA)
    h = np.reshape(v[8], (3, 3))
    h = (1/h.item(8)) * h
    return h
```

The following are the computed homography and the output plots for the given images.

1. Base Image

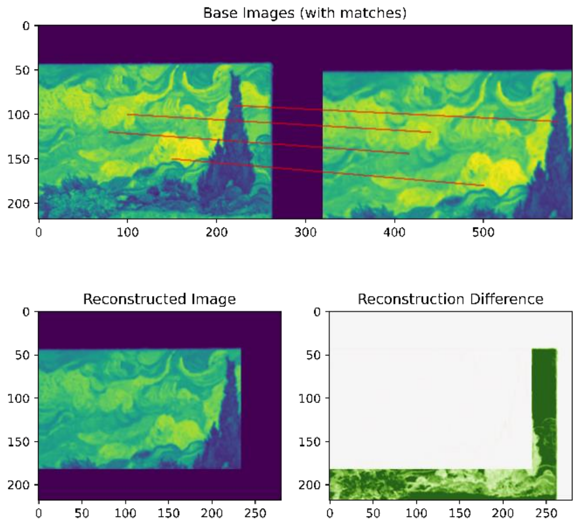
```
Computed Homography:
[[ 1.20000000e+00  1.05145343e-13  1.16066137e-12]
 [-1.01755716e-13  1.20000000e+00  4.86375697e-13]
 [-7.97958347e-16  7.67959488e-17  1.00000000e+00]]
```





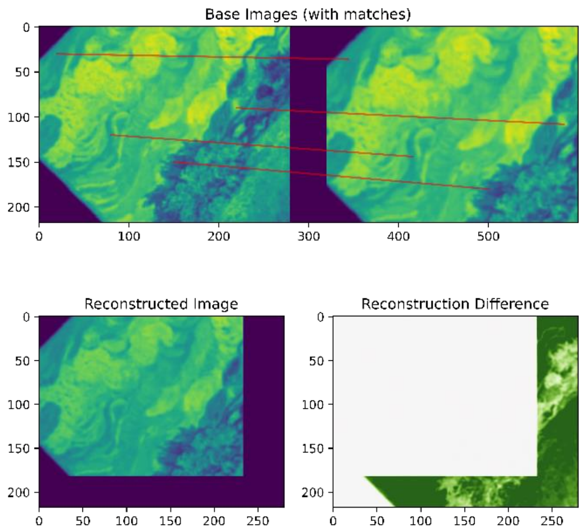
2. Transformed Image 1

```
Computed Homography:
[[ 1.20000000e+00  1.55158522e-14 -1.30447541e-11]
 [ 8.43597167e-14  1.20000000e+00 -3.77103467e-11]
 [ 7.18845908e-16  6.09501621e-16  1.00000000e+00]]
```



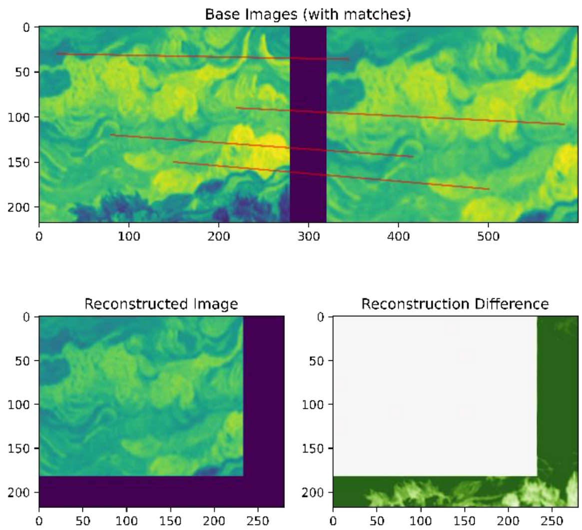
3. Transformed Image 2

```
Computed Homography:
[[ 1.20000000e+00  1.05145343e-13  1.16066137e-12]
 [-1.01755716e-13  1.20000000e+00  4.86375697e-13]
 [-7.97958347e-16  7.67959488e-17  1.00000000e+00]]
```



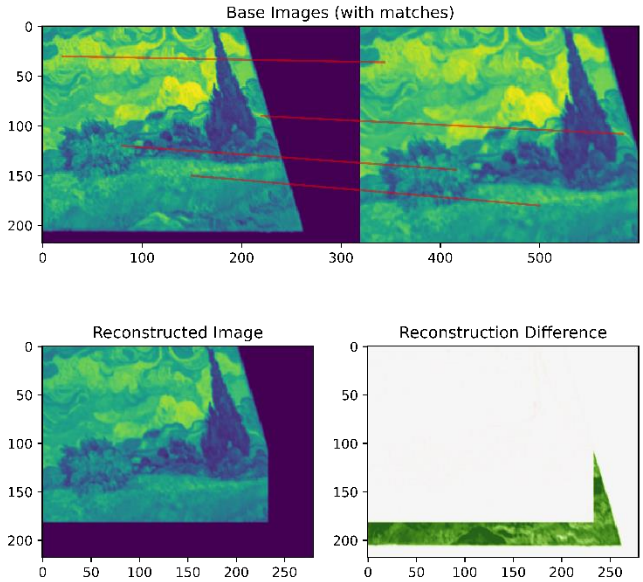
4. Transformed Image 3

```
Computed Homography:
[[ 1.20000000e+00  1.05145343e-13  1.16066137e-12]
 [-1.01755716e-13  1.20000000e+00  4.86375697e-13]
 [-7.97958347e-16  7.67959488e-17  1.00000000e+00]]
```



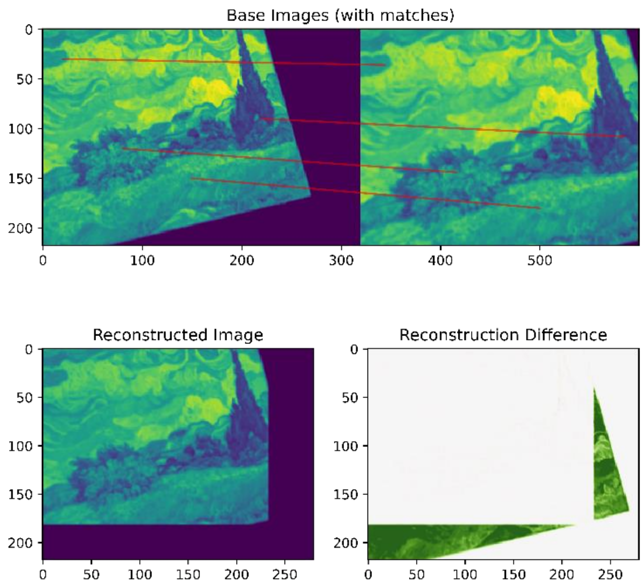
5. Transformed Image 4

```
Computed Homography:
[[ 1.20000000e+00  1.05145343e-13  1.16066137e-12]
 [-1.01755716e-13  1.20000000e+00  4.86375697e-13]
 [-7.97958347e-16  7.67959488e-17  1.00000000e+00]]
```



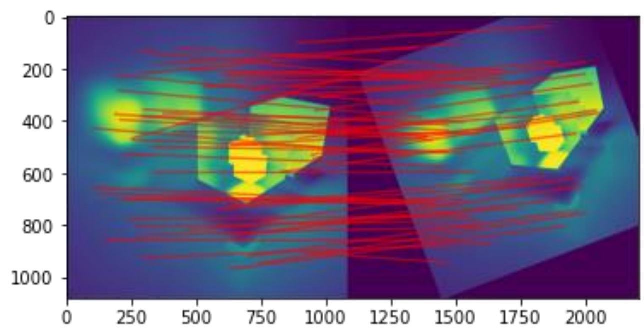
6. Transformed Image 5

```
Computed Homography:
[[ 1.20000000e+00  1.05145343e-13  1.16066137e-12]
 [-1.01755716e-13  1.20000000e+00  4.86375697e-13]
 [-7.97958347e-16  7.67959488e-17  1.00000000e+00]]
```



P3.2.2 Computing Homographies from Noisy Matches

The below shows the image with noisy matches



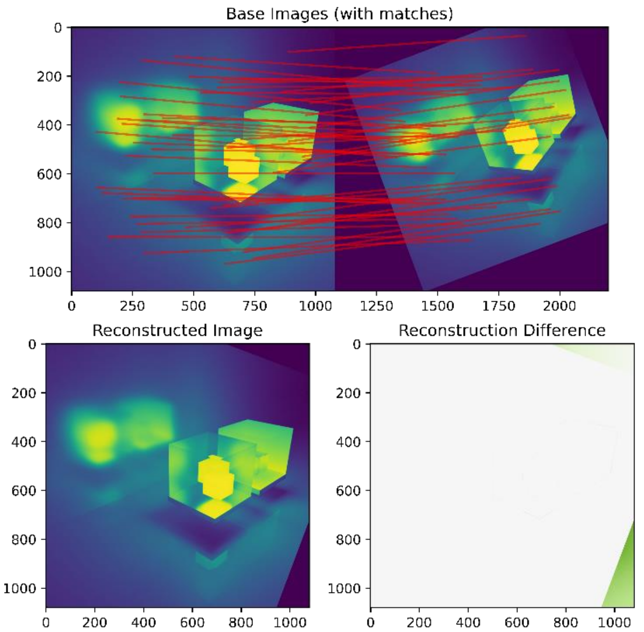
The below is the implementation of the RANAC procedure for removing outlier matches.

```
def geometricDistance(correspondence, h):
    p1 = np.transpose(np.matrix([correspondence[0].item(0), correspondence[0].item(1), 1]))
    estimatep2 = np.dot(h, p1)
    estimatep2 = (1/estimatep2.item(2))*estimatep2
    p2 = np.transpose(np.matrix([correspondence[0].item(2), correspondence[0].item(3), 1]))
    error = p2 - estimatep2
    return np.linalg.norm(error)

def solve_homograhy(corrs):
    aList = []
    for corr in corrs:
        p1 = np.matrix([corr.item(0), corr.item(1), 1])
        p2 = np.matrix([corr.item(2), corr.item(3), 1])
        a2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2),
              p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1) * p1.item(2)]
        a1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2), 0, 0, 0,
              p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0) * p1.item(2)]
        aList.append(a1)
        aList.append(a2)
    matrixA = np.matrix(aList)
    u, s, v = np.linalg.svd(matrixA)
    h = np.reshape(v[8], (3, 3))
    h = (1/h.item(8)) * h
    return h

def solve_homography_ransac(corr, rounds, sigma, s):
    corr = np.matrix(corr)
    maxInliers = []
    finalH = None
    for i in range(rounds):
        #find 4 random points to calculate a homography
        corr1 = corr[random.randrange(0, len(corr))]
        corr2 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((corr1, corr2))
        corr3 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr3))
        corr4 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr4))
        #call the homography function on those points
        h = solve_homograhy(randomFour)
        inliers = []
        for i in range(len(corr)):
            d = geometricDistance(corr[i], h)
            if d < s:
                inliers.append(corr[i])
        if len(inliers) > len(maxInliers):
            maxInliers = inliers
            finalH = h
        if len(maxInliers) > (len(corr)*sigma):
            break
    return finalH, inliers
```

The following is the computed homography using the RANSAC implementation and the Inlier matches.

```
Computed Homography:
[[ 8.00000000e-01  3.00000000e-01 -4.66728146e-10]
 [-3.00000000e-01  8.00000000e-01  2.20000000e+02]
 [ 1.23505490e-15 -1.91441765e-16  1.00000000e+00]]
Inlier Matches:
[[329.0, 499.0, 412.9, 520.5], [599.0, 246.0, 553.0, 237.10000000000002], [103.0, 428.0, 210.8, 531.5], [621.0, 214.0, 561.0, 2
04.90000000000003], [632.0, 842.0, 758.2, 704.0], [255.0, 472.0, 345.6, 521.1], [483.0, 202.0, 447.0, 236.70000000000002], [97
9.0, 310.0, 876.2, 174.3], [425.0, 120.0, 376.0, 188.5], [415.0, 442.0, 464.6, 449.1], [222.0, 526.0, 335.4, 574.2], [286.0, 83
9.0, 480.5, 805.4000000000001], [862.0, 264.0, 768.8000000000001, 172.60000000000005], [151.0, 857.0, 377.9, 860.300000000000
1], [211.0, 702.0, 379.4, 718.3], [250.0, 699.0, 409.7, 704.2], [287.0, 135.0, 270.1, 241.9], [201.0, 283.0, 245.7, 386.1], [74
5.0, 946.0, 879.8, 753.3000000000001], [974.0, 411.0, 902.5, 256.6], [574.0, 740.0, 681.2, 639.8], [198.0, 224.0, 225.600000000
00002, 339.8], [947.0, 529.0, 916.3, 359.1000000000001], [754.0, 784.0, 838.4000000000001, 621.0], [608.0, 225.0, 553.900000000
0001, 217.6], [695.0, 459.0, 693.7, 378.70000000000005], [828.0, 595.0, 840.9000000000001, 447.6], [637.0, 493.0, 657.5, 423.30
000000000007], [633.0, 968.0, 796.8, 804.5000000000001], [467.0, 819.0, 619.3, 735.1], [269.0, 805.0, 456.70000000000005, 783.
3], [960.0, 360.0, 876.0, 220.0], [299.0, 356.0, 346.0, 415.1], [706.0, 920.0, 840.8000000000001, 744.2], [770.0, 826.0, 863.8,
649.8000000000001], [952.0, 544.0, 924.8, 369.6000000000001], [337.0, 598.0, 449.0, 597.3000000000001], [375.0, 371.0, 411.3, 4
04.3], [189.0, 374.0, 263.40000000000003, 462.5], [106.0, 656.0, 281.6, 713.0], [623.0, 727.0, 716.5, 614.7], [298.0, 925.0, 51
5.9, 870.6], [891.0, 467.0, 852.9000000000001, 326.3], [530.0, 267.0, 504.1, 274.6], [735.0, 308.0, 680.4, 245.9], [244.0, 775.
0, 427.70000000000005, 766.8], [128.0, 678.0, 305.8, 724.0], [209.0, 395.0, 285.70000000000005, 473.3], [888.0, 101.0, 740.7, 3
4.40000000000034], [293.0, 380.0, 348.4, 436.1]]
```

The image reconstruction using RANSAC implementation



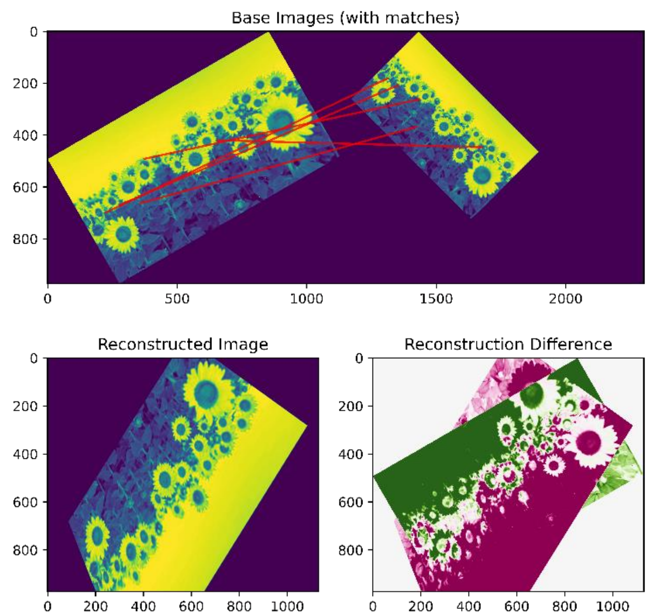
The above image was generated for the parameters of sigma, rounds and s shown below.

H_robust, matches = solve_homography_ransac(matches_noisy, rounds=100, sigma=5, s=5)

3.3 Feature Matching Pipeline

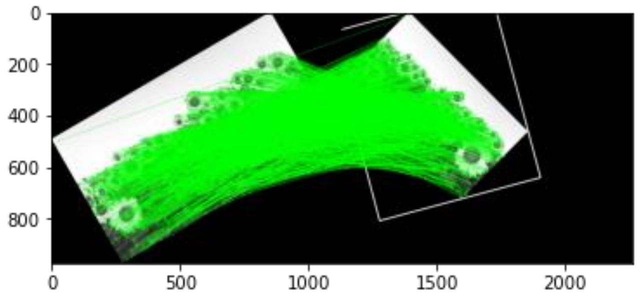
The below is the feature matching generated without using the OpenCV.

```
Computed Homography:
[[ 1.61917769e+00  1.42480685e-01 -2.37012432e+02]
 [ 3.46023094e-01 -6.60697870e-01  8.08658305e+02]
 [ 9.04922460e-04  6.02965950e-04  1.00000000e+00]]
Total Compute Time: 94.12769103050232
```



3.4 Feature Matching with OpenCV

The below is the output image generated for the feature matching problem using the OpenCV.



The following shows the execution time of the feature matching pipeline using OpenCV. We also show few statistics like the computed homography, total matches and inlier matches generated through OpenCV.

```
Total Compute Time: 0.6651182174682617

Computed Homography:
[[ 1.72788034e-01 -6.44181052e-01  5.78987181e+02]
 [ 6.44737866e-01  1.73169352e-01 -8.56356986e+01]
 [ 9.15452726e-07  8.64271606e-07  1.00000000e+00]]

Total Matches:
2926

Inlier Matches:
1584
```

The implementation using OpenCV is better than the manual implementation as the execution time is 140 times less when using OpenCV. Also, the number of matches is very few in manual implementation and not that accurate compared to OpenCV.