ECE 661: Homework 4

Monday, 1 October 2018

Dr. Avinash Kak

Naveen Madapana

1. Interest Point Extraction

1.1 Definition

A point in the image is considered as an interest point if its pixel intensity is a local maxima or a local minima w.r.t x axis, y axis and the z axis (scale). An ideal interest point is translation, rotation and scale invariant. There are several interest point detectors, namely, 1. Scale Invariant Feature Transform (SIFT), 2. Spped Up Robust Features (SURF), 3. Harris corners, etc.

1.2 SIFT

The main steps in SIFT are summarized below.

1. The theorem that directly relates Laplacian of Gaussian (LoG) with Difference of Gaussian (DoG) is central to SIFT and SURF. It is given as follows. It means that LoG can be approximated by estimating the DoG pyramid (DoG at various scales)

$$\frac{\partial f f(x,y,\sigma)}{\partial \sigma} = \sigma \Delta^2 f f(x,y,\sigma) \tag{1}$$

2. DoG pyramid consists of scale sapce representation of an image at several octaves, i.e. σ_0, $2\sigma_0$, $4\sigma_0$, $8\sigma_0$,.... Further, each level, multiple scales at minor variations of standard deviation are considered i.e. for instance at σ_0, the variations are $\sigma_0 + d\sigma_0$, $\sigma_0 + 2d\sigma_0$, $\sigma_0 + 3d\sigma_0$,$\sigma_0 + 4d\sigma_0$, ... All of this constitutes the DoG pyramd.

3. Given the DoG pyramid, we mark interest point as the pixels that are either a local maxima or a local minima i.e. the intensity of an interest point is larger than its 27 neighbors (9 - same scale, 9 - a scale above, 9 - a scale below).

4. As σ increases, the spatial resolution gets coarser. As a result, the image needs to be downsampled by half at each level of space representation. In order to find the corresponding points of local maxima/minima in the original image, following Taylor's expansion is used:

$$D(\bar{X}) \approx D(\bar{X_0}) + J^T(\bar{X_0})\bar{X} + \frac{1}{2}\bar{X}^T H(X_0)\bar{X} \tag{2}$$

   Where, J is the Jacobian matrix (gradient vector) evaluated at the points of extrema. H is the Hessian matrix (of double derivatives) evaluated at $\bar{X_0}$. Now, we will set $\frac{\partial D(\bar{X})}{\partial \bar{X}} = 0$. This gives the optimal $\bar{X} = -H^{-1}(\bar{X_0})J(\bar{X_0})$

5. Due to the errors in numerical precision and sampling, the value of $D(\bar{X}) < 0.1$ is thresholded to find more local extrema. We further make sure that the interest points are as far away from each other as possible.

6. Next, we find the dominant orientation at each interest point and consider a rectangle of size 8 x 16 in the dominant orientation at the interest point, and find a 128 dimensional descriptor vector at each interest point.

1.3 HARRIS Corners

In HARRIS corner detection, we aim to identify the points in the image at which the intensity change drastically both in x and y directions are considered as interest points. It utilizes the gradients in x and y computed using either haar filters or a simple Sobel operator. Further, this technique is applied at a particular scale. In contrast to SIFT, this technique is not invariant to scale. Nevertheless, these interest points are translation and rotation invariant.

Consider the following Haar filters to compute the gradients. The kernels are convolved across the images to obtain the gradients in x and y directions at a particular scale.

$$\frac{\partial}{\partial x} = \begin{bmatrix} -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \end{bmatrix}$$

$$\frac{\partial}{\partial y} = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Next, a $5\sigma \times 5\sigma$ sized region is considered. Matrix C is estimated using this region at every point in the image. C is given below. Note that Σ is the summation over all of the pixels in the region.

$$C = \begin{bmatrix} \Sigma(d_x)^2 & \Sigma dxdy \\ \Sigma dxdy & \Sigma(dy)^2 \end{bmatrix}$$

Now, the points where the matrix C is non singular are considered as interest points. When a matrix is non singular, the eigen values will be non zero. Further we note that,

$$trace(C) = \lambda_1 + \lambda 2 = \Sigma(d_x)^2 + \Sigma(d_y)^2 \tag{3}$$

$$det(C) = \lambda_1 \lambda 2 = \Sigma(d_x)^2 \Sigma(d_y)^2 - \Sigma(d_x dy)^2 \tag{4}$$

Let $r = \frac{\lambda_2}{\lambda_1}$, where λ_1 and λ_2 are eigenvalues and $\lambda_1 \geq \lambda_2$. Now,

$$\beta = \frac{det(C)}{trace(C)^2} = \frac{\lambda_1 \lambda_2}{(\lambda_1 + \lambda_2)^2} = \frac{r}{(1+r)^2} \tag{5}$$

For Harris corner detection, we threshold $\beta > 0.08$, to find the interest points. Next, we perform non maxima suppression to reduce the false positives.
At each interest point, a matrix of size 31 x 31 is considered which is centered at the interest point. A descriptor vector is considered as an array of pixel intensities in the 31 x 31 region.

Establishing Point Correspondences

Let u and v be the vectors of same length (k). Let μ_u and μ_v be the mean of u and v respectively.

2.1 Sum Squared Differences (SSD)

$$SSD(u, v) = \sum_i (u_i - v_i)^2 \tag{6}$$

2.2 Normalized Cross Correlation (NCC)

$$NCC(u,v) = \frac{\sum_i (u_i - \mu_u)(v_i - \mu_v)}{\sqrt{\sum_i (u_i - \mu_u)^2 \sum_i (v_i - \mu_v)^2}} \tag{7}$$

The results obtained by using NCC are better than SSD as the former is mean normalized and is not sensitive the uniform increase/decrease in the intensities. In experiments, the value of threshold NCC ranges any where from 0.97 to 0.999.

Experiments

SIFT (opencv), SURF (opencv) and HARRIS corner detection (own code) were used to find correspondences between four pairs of images. The first two pairs are given in the class, and the other two images were taken by me.
Further, the Harris corner detection is done two different scales (1.414 and 2.00). It was found that the quality of point correspondences is hihger for higher scales due to smoothing. Also, SSD performs better than NCC unless the average intensity and scale of the image does not change.
SURF provided comparatively large number of interest points at the same threshold of NCC. Hence higher NCC thresholds are used to reduce the number of interest points.

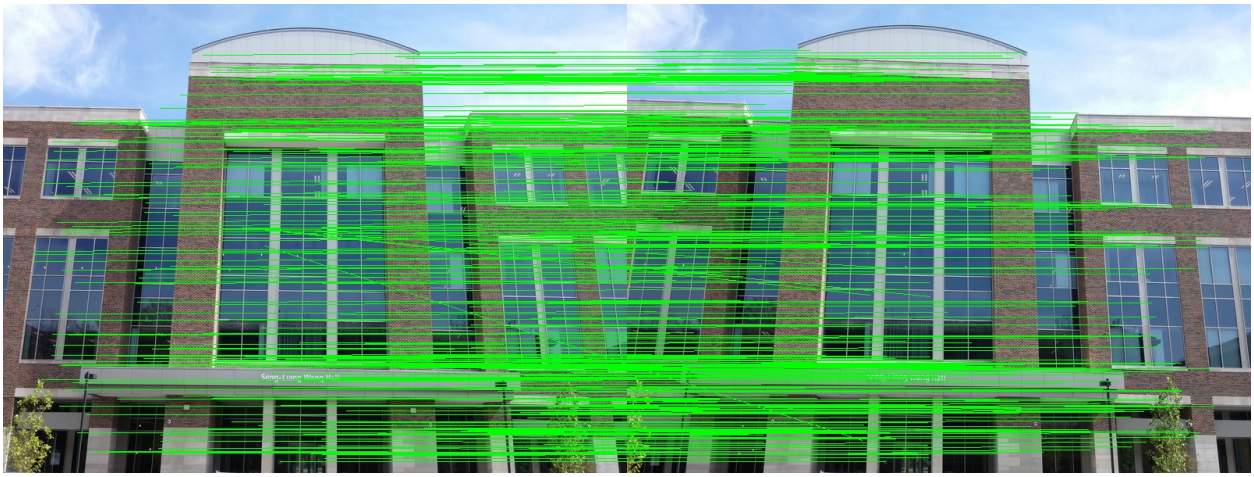
Pair 1

Original Images

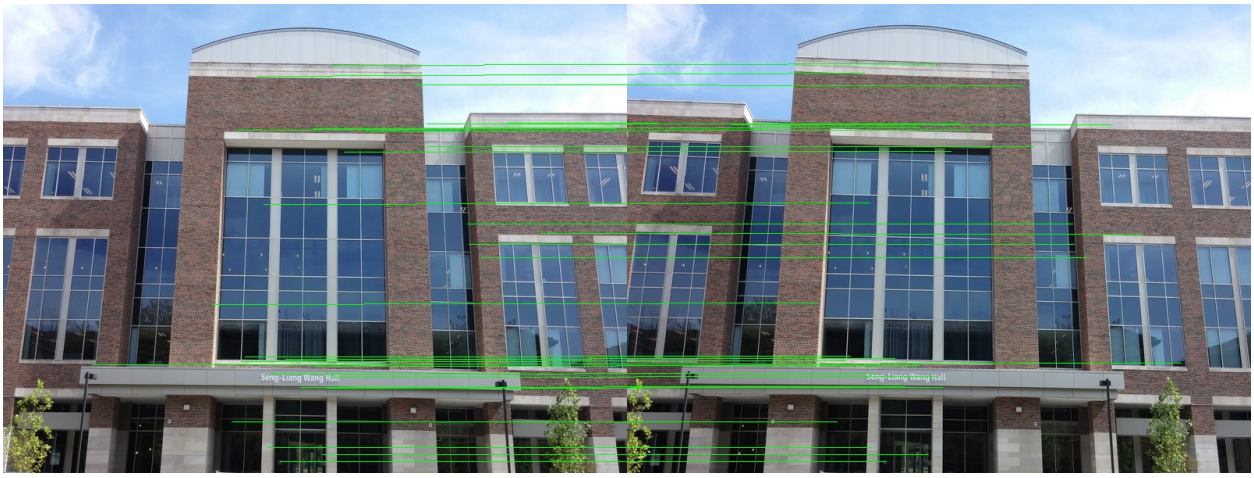
(a) Image 1

(b) Image 2

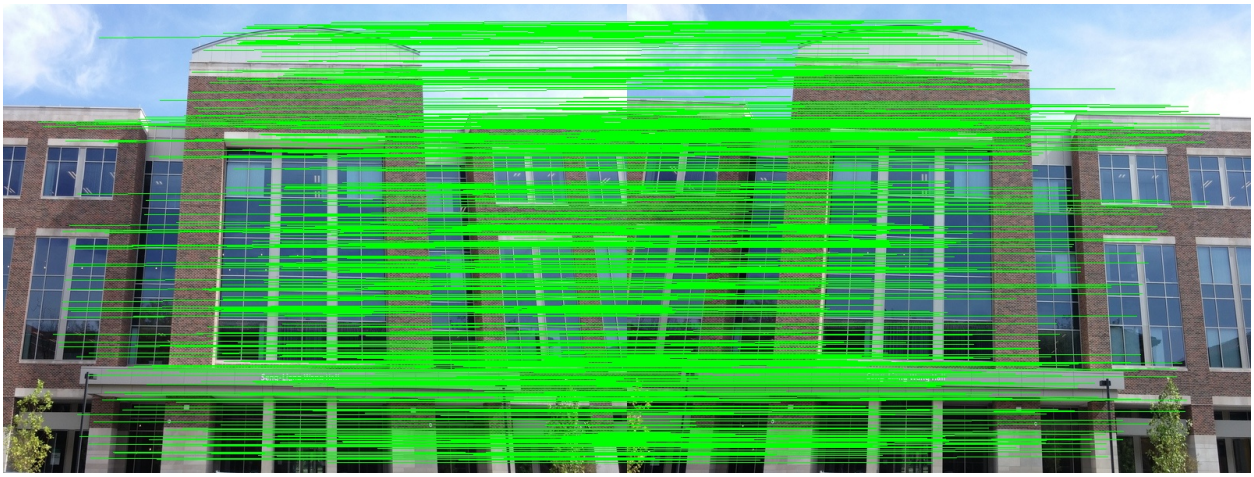
SIFT (thresh = 0.990)



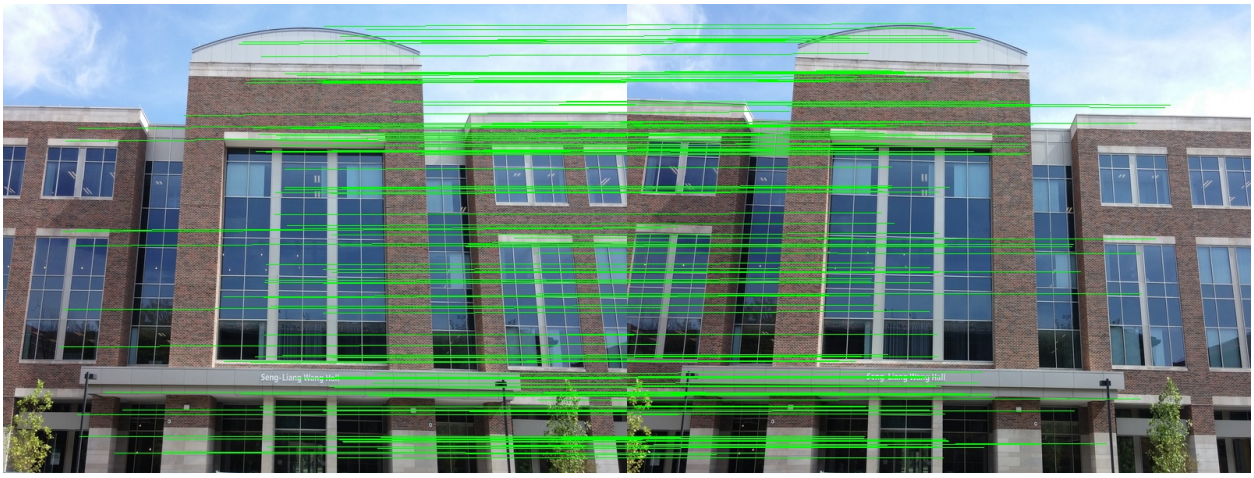
SIFT (thresh = 0.99)



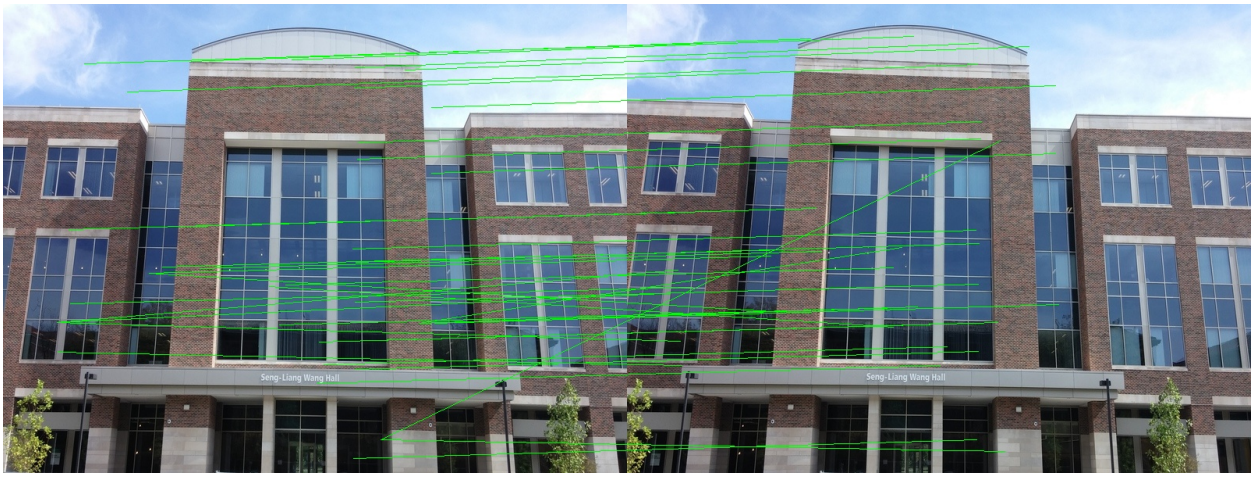
SURF (thresh = 0.9994)



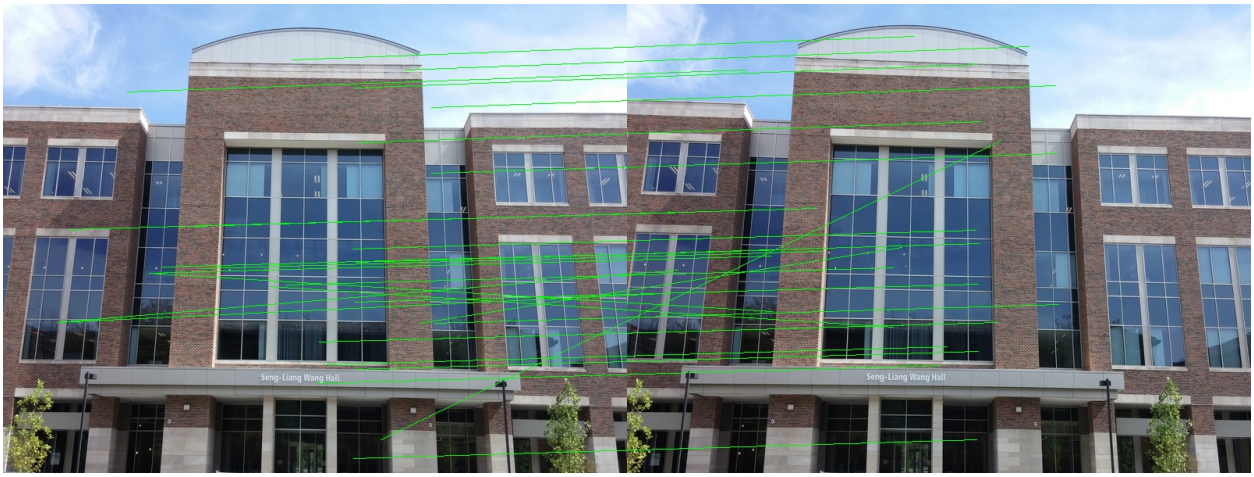
SURF (thresh = 0.9998)



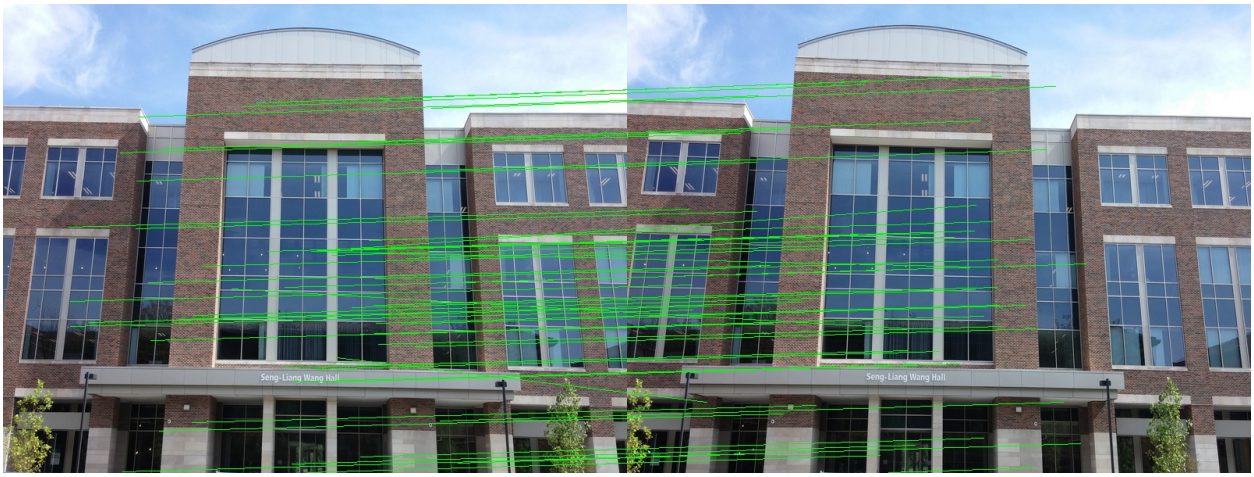
HARRIS (thresh = 0.990, sigma = 1.1414)



HARRIS (thresh = 0.993, sigma = 1.1414)



HARRIS (thresh = 0.99, sigma = 2.0)



Pair 2

Original Images

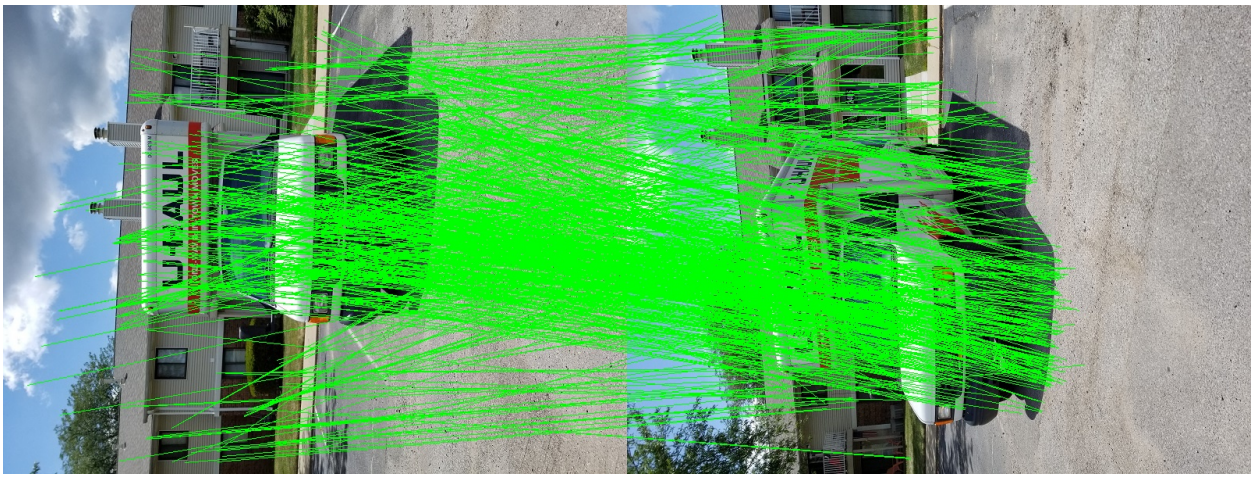
(a) Image 1

(b) Image 2

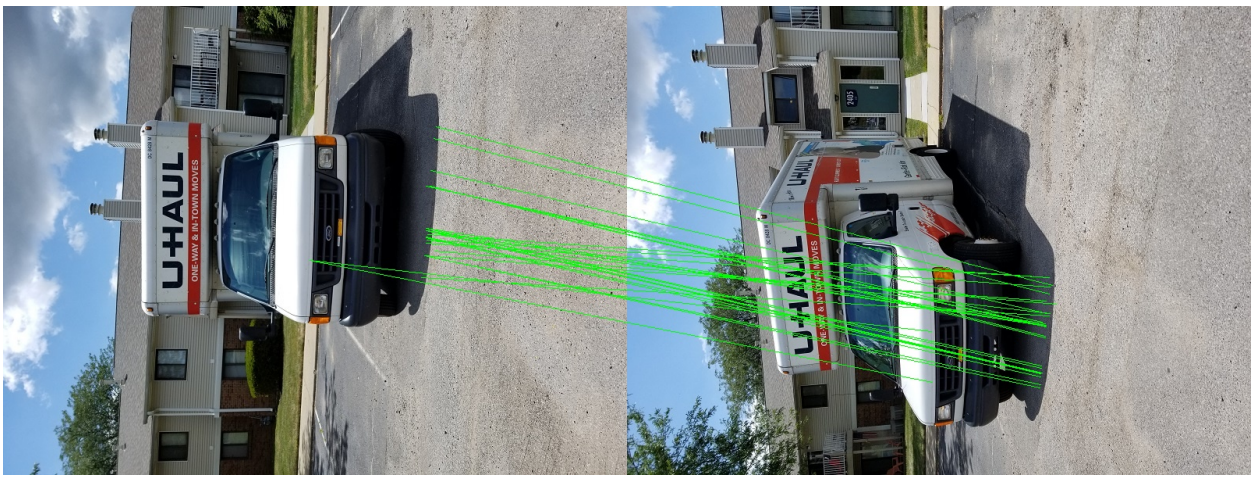
SIFT (thresh = 0.98)

SIFT (thresh = 0.99)

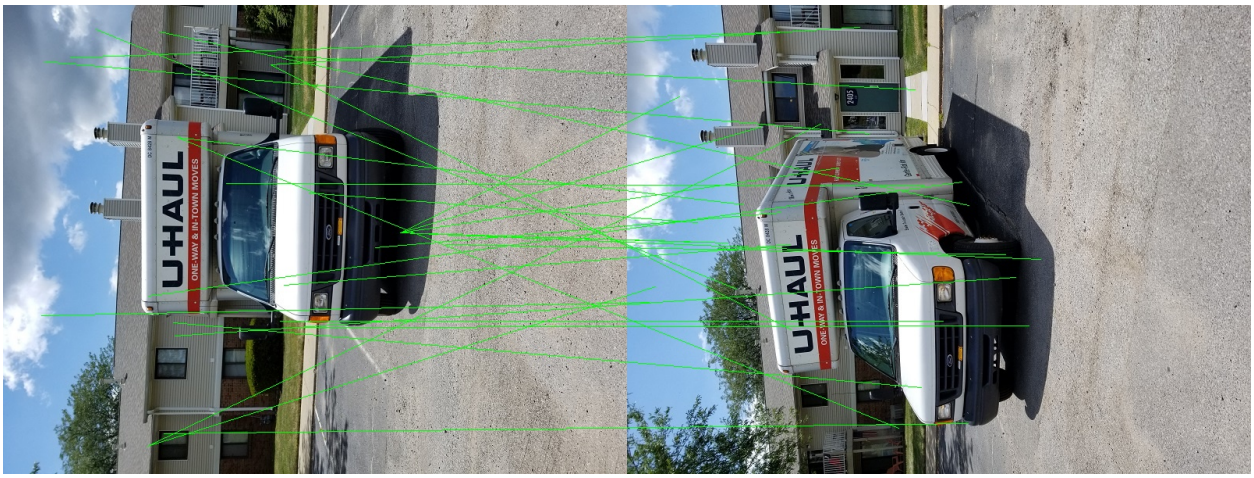
SURF (thresh = 0.99)



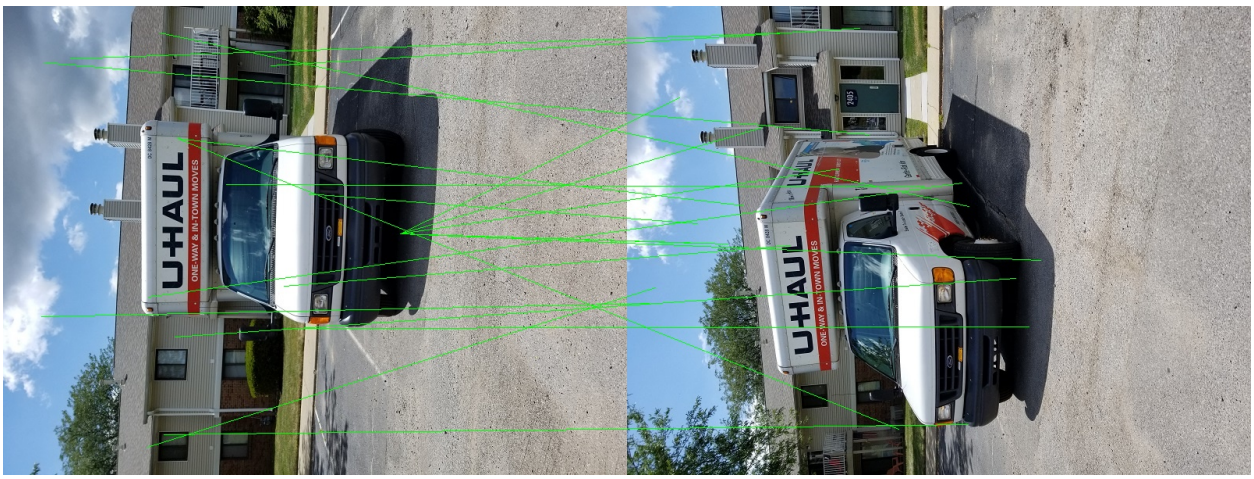
SURF (thresh = 0.9990)



HARRIS (thresh = 0.910, sigma = 1.1414)



HARRIS (thresh = 0.92, sigma = 1.1414)



HARRIS (thresh = 0.94, sigma = 2.0)

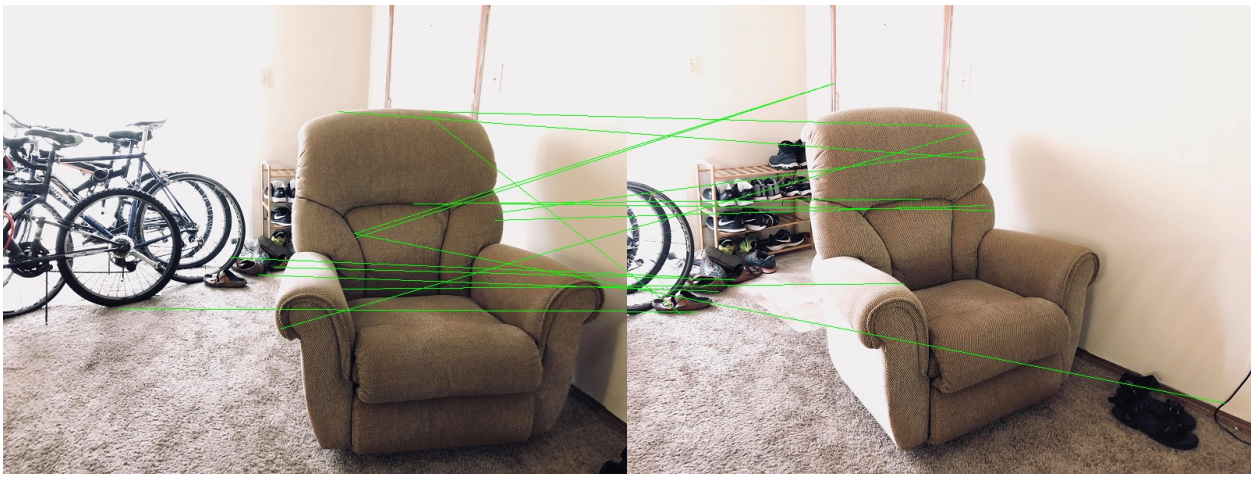
Pair 3

Original Images

(a) Image 1

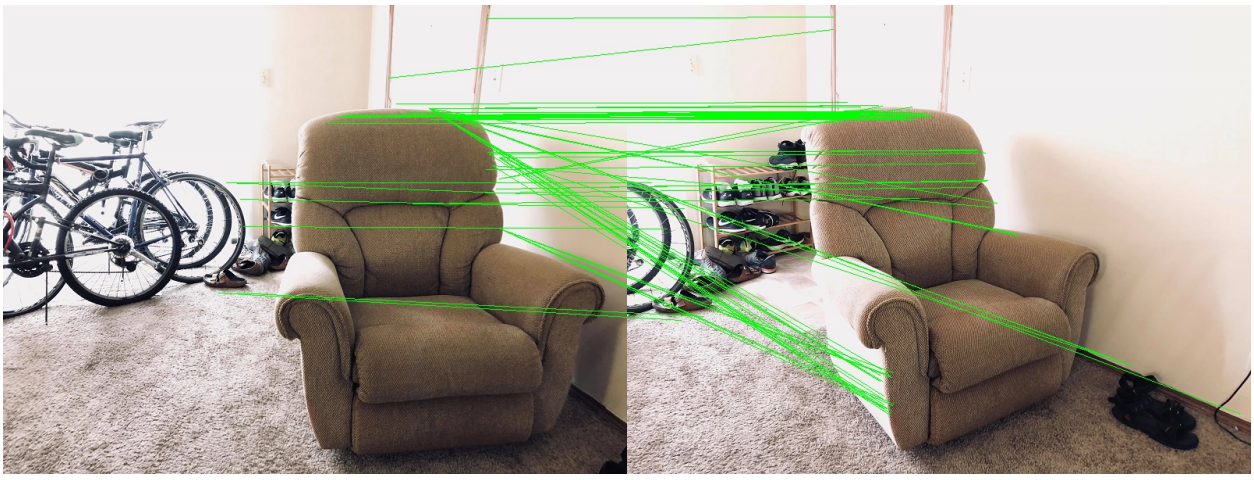
(b) Image 2

SIFT (thresh = 0.97)



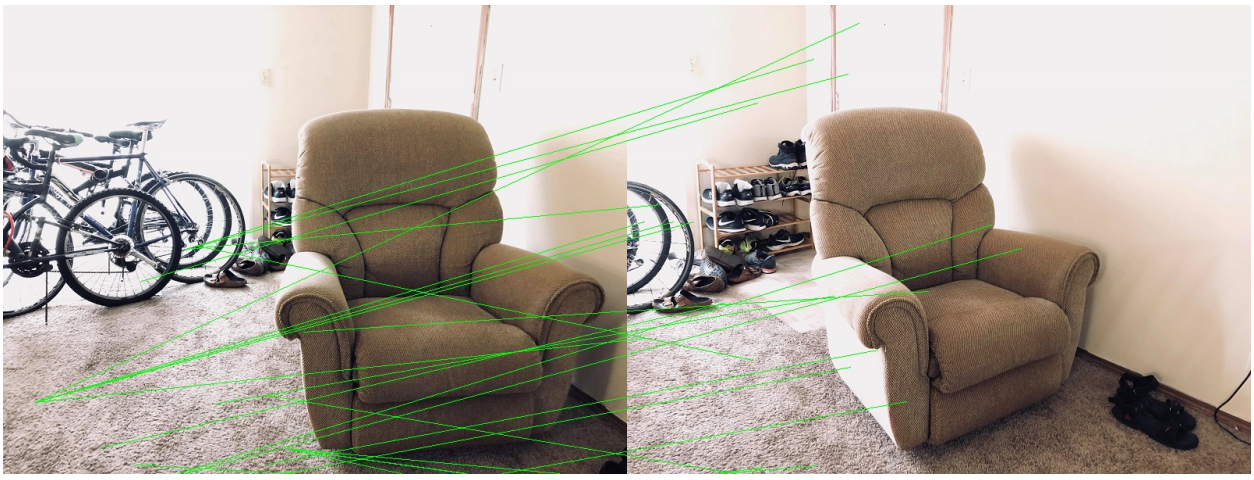
SIFT (thresh = 0.99)

SURF (thresh = 0.996)

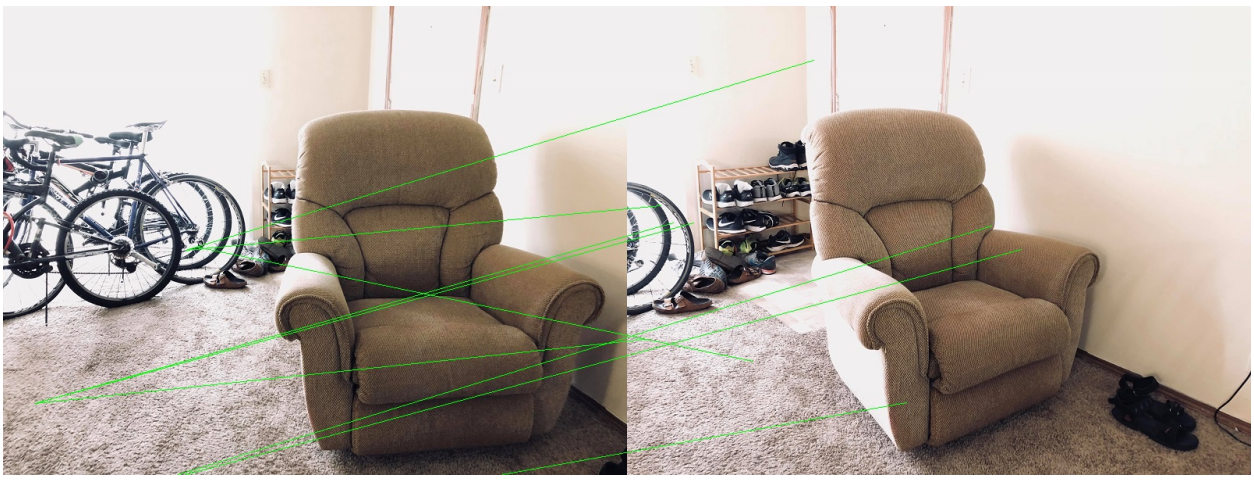


SURF (thresh = 0.9900)

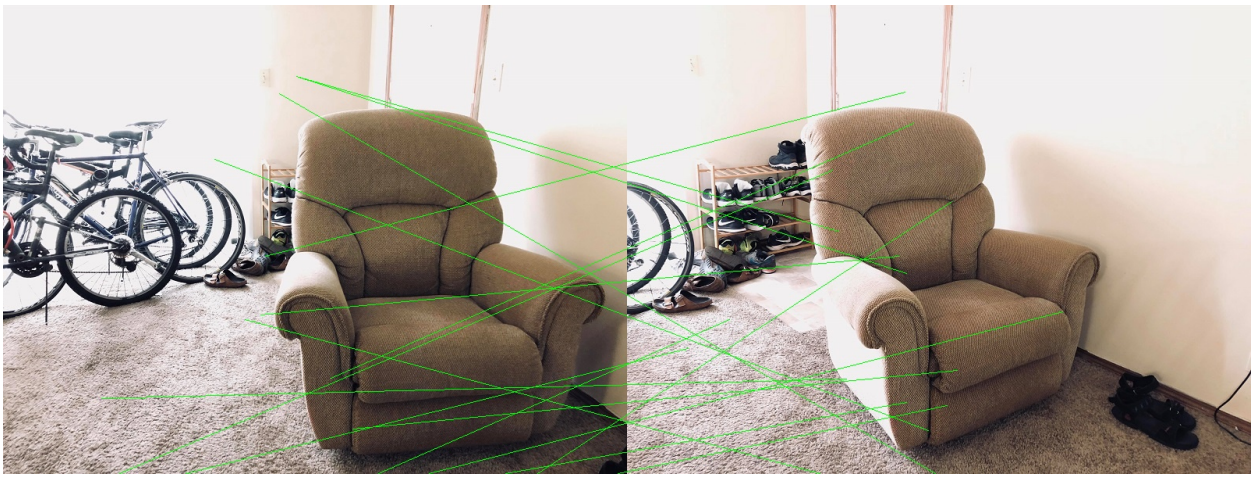
HARRIS (thresh = 0.910, sigma = 1.1414)



HARRIS (thresh = 0.94, sigma = 1.1414)



HARRIS (thresh = 0.92, sigma = 2.0)



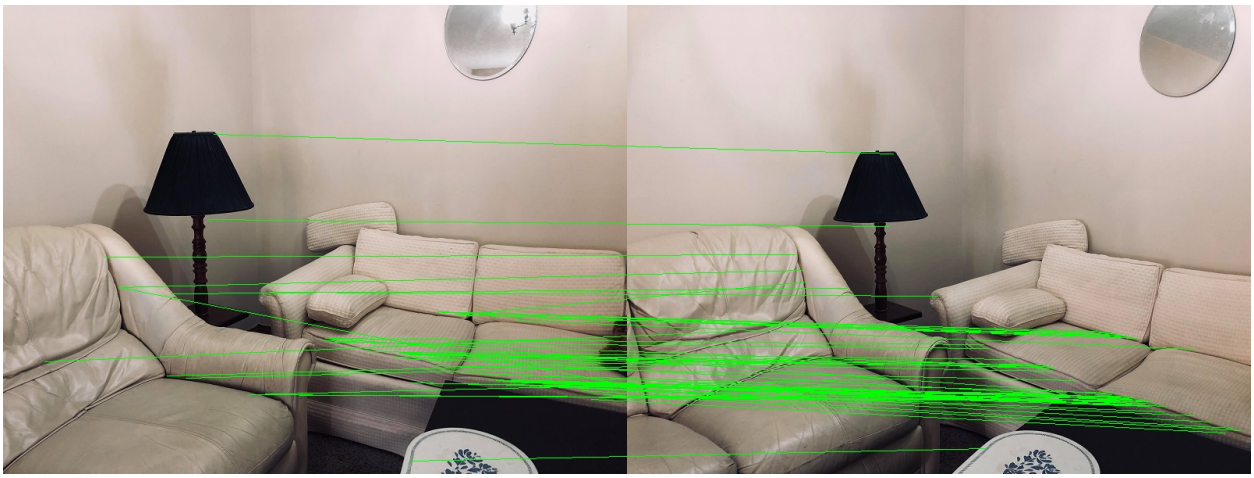
Pair 4

Original Images

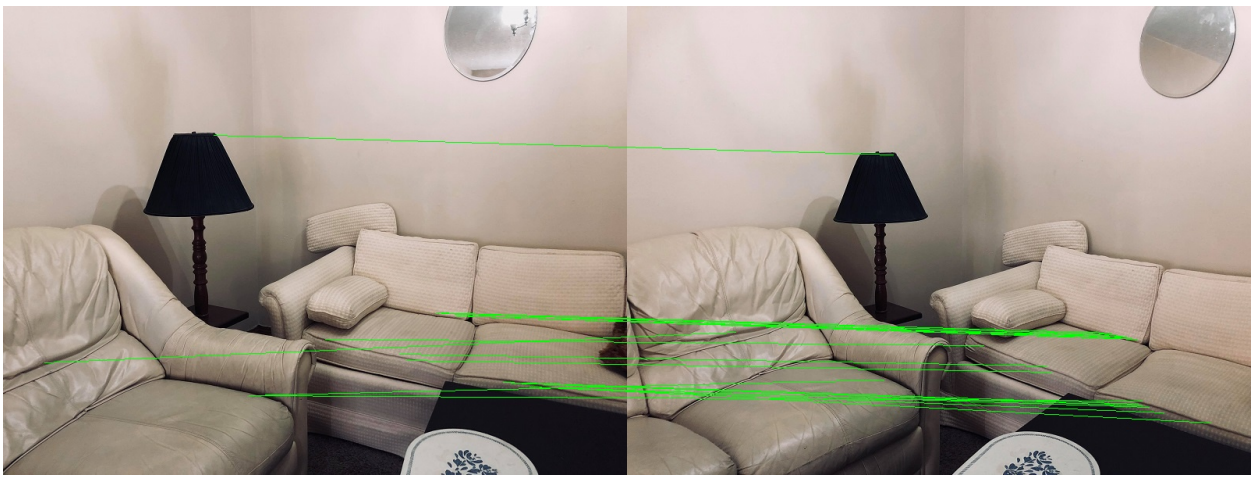
(a) Image 1

(b) Image 2

SIFT (thresh = 0.98)

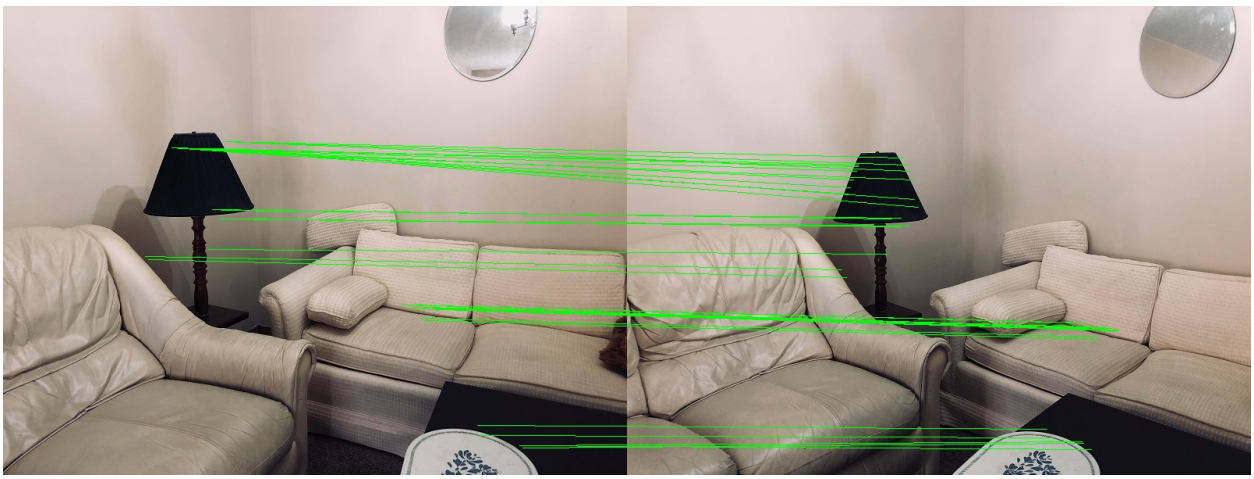


SIFT (thresh = 0.99)

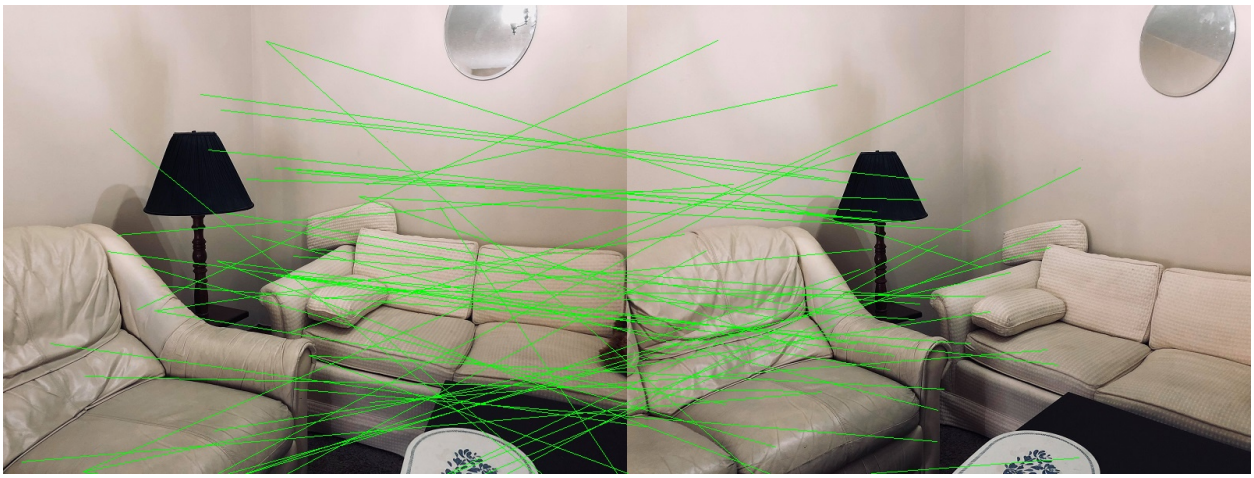


SURF (thresh = 0.99)

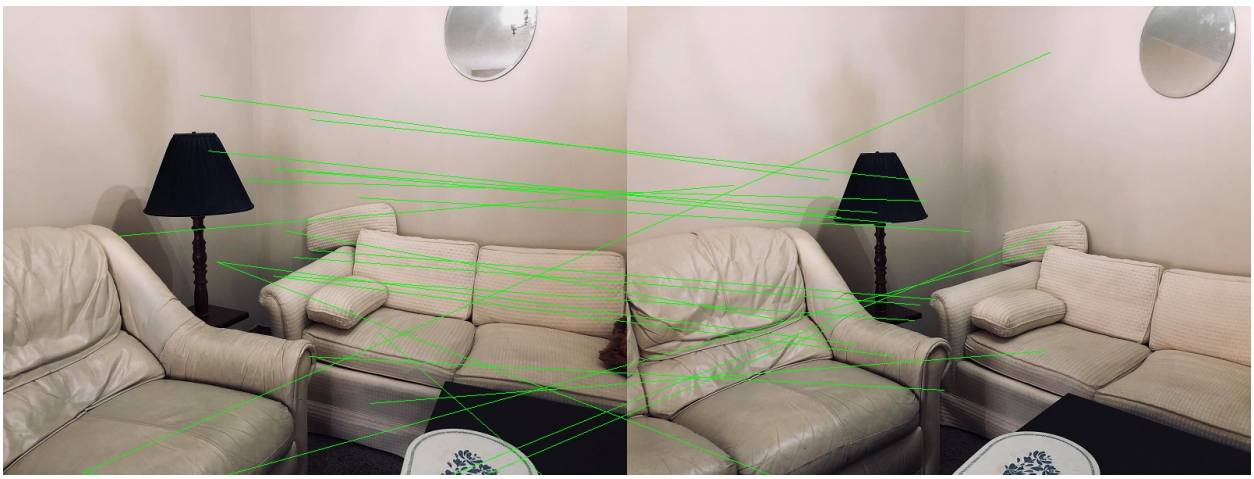
SURF (thresh = 0.9990)



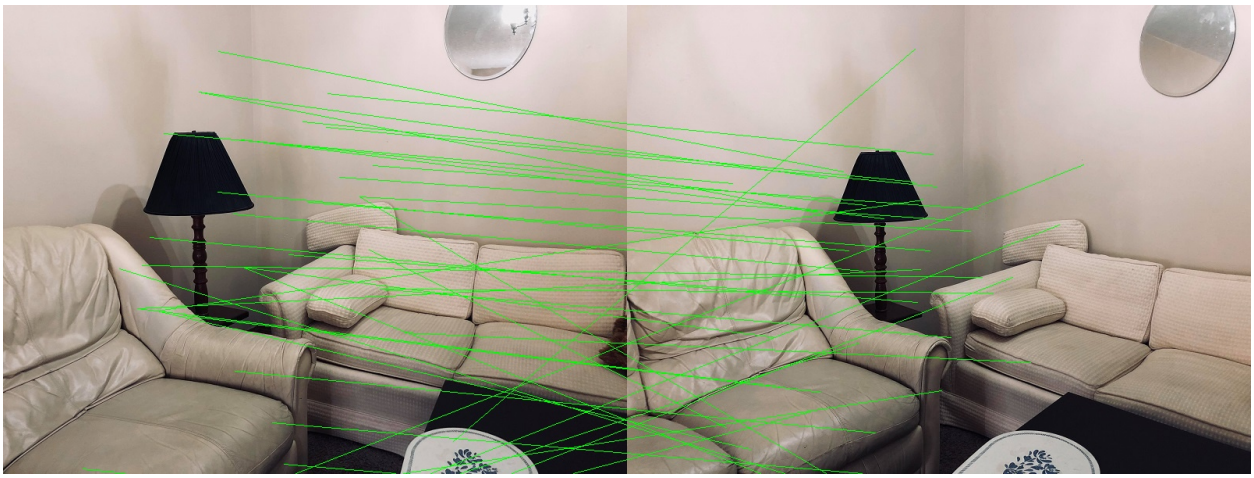
HARRIS (thresh = 0.910, sigma = 1.1414)



HARRIS (thresh = 0.94, sigma = 1.1414)



HARRIS (thresh = 0.94, sigma = 2.0)



Source Code

```
import cv2
import numpy as np
import time
from os.path import join, basename, splitext, dirname

def harris(img, sigma = 1.414, thresh = 0.08):
    # img is gray scale image

    # Size of Gaussian Kernel
    K = int(2 * np.floor(3 * 1.414 * sigma) + 1)
    # img = cv2.GaussianBlur(img, (K, K), 0)

    img = cv2.GaussianBlur(img, None, sigma)
    dx_img = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=K)
    dy_img = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=K)

    img_height, img_width = img.shape[:2]

    Nh = int(5 * sigma)
    if(Nh % 2 == 0): Nh += 1

    df_idx = (Nh-1)/2

    H = np.zeros((img_height-2*df_idx, img_width-2*df_idx))

    nms_win_size = 31 # non maximum suppression window size.
    nms_half = (nms_win_size-1)/2

    desc_size = 8 # window size is 31

    kps = []
    features = []
```

```
    for zr_idx, row_idx in enumerate(range(df_idx, img_height - df_idx)):
        row_ids = range(row_idx-df_idx, row_idx+df_idx)
        for zc_idx, col_idx in enumerate(range(df_idx, img_width - df_idx)):
            col_ids = range(col_idx-df_idx, col_idx+df_idx)
            M = img[row_ids, col_ids]
            dMx = dx_img[row_ids, col_ids]
            dMy = dy_img[row_ids, col_ids]

            c11 = np.sum(np.sum(dMx**2))
            c22 = np.sum(np.sum(dMy**2))
            c12 = np.sum(np.multiply(dMx, dMy).flatten())
            c21 = c12

            trace = c11 + c22
            det = c11 * c22 - c12 * c21

            if(trace < 1e-5): continue

            beta = det / (trace * trace) # Min = 0, Max = 0.25. Higher the better.

            if(beta > thresh):
                H[zr_idx, zc_idx] = beta

    ## Non Maximum supression
    for zr_idx, row_idx in enumerate(range(df_idx, img_height - df_idx)):
        for zc_idx, col_idx in enumerate(range(df_idx, img_width - df_idx)):

            # if did not pass previous threshold, continue
            if(H[zr_idx, zc_idx] == 0): continue

            # Boundary conditions. Ignore points on the boundary.
            if(zr_idx - nms_half < 0 or zc_idx - nms_half < 0): continue
            if(zr_idx + nms_half > img_height-1 or \
                zc_idx + nms_half > img_width-1): continue

            nms_win = H[zr_idx - nms_half:zr_idx + nms_half, \
                        zc_idx - nms_half:zc_idx + nms_half]

            # if beta is max in that window, then consider. Ignore otherwise.
            if(np.max(nms_win[:]) == H[zr_idx, zc_idx]):
                kps.append([row_idx, col_idx])
                features.append(img[row_idx-desc_size:row_idx+desc_size, \
                    col_idx-desc_size:col_idx+desc_size].flatten().tolist())

    return np.float32(kps), np.float32(features)

def extract_kps(image, ftype = 'sift', sigma = 1.414):
    # ftype - feature type 'sift' or 'surf'
    # image - np.ndarray (_ x _ x 3)
    # convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    if(ftype.lower() == 'sift'):
```

```
            descriptor = cv2.xfeatures2d.SIFT_create()
            # keypoints (cv2.KeyPoint object) and features (ndarray).
            (kps, features) = descriptor.detectAndCompute(image, None)
            kps = np.float32([kp.pt for kp in kps])
        elif ftype.lower() == 'surf':
            descriptor = cv2.xfeatures2d.SURF_create()
            # keypoints (cv2.KeyPoint object) and features (ndarray).
            (kps, features) = descriptor.detectAndCompute(image, None)
            kps = np.float32([kp.pt for kp in kps])
        elif ftype.lower() == 'harris':
            start = time.time()
            (kps, features) = harris(gray, sigma = sigma)
            print 'Harris corners: %.02f secs'%(time.time()-start)

        # kps: ndarray (_ x 2); features: ndarray (_ x 128)
        return (kps, features)

def mean_normalize(M, axis = 0):
        # Mean normalize matrix M (_ x k) in rows
        if(axis == 1): M = M.transpose() # (k x _)
        M -= np.mean(M, axis = 0)

        norms = np.linalg.norm(M, axis = 0)
        norms[norms == 0.0] = 1e-10

        M /= norms
        if(axis == 1): M = M.transpose()
        return M

def dist_mat_vec(M, vec, method = 'ncc'):
        # M : ndarray ( _ x k); vec: (1 x k)
        if(method.lower() == 'ssd'):
            return np.linalg.norm(M - vec, axis = 1)
        elif(method.lower() == 'ncc'):
            # Mean Normalizing rows of M
            M = mean_normalize(M, axis = 1)
            # Mean normalizing vec
            vec = vec - np.mean(vec)
            vect = vec / np.linalg.norm(vec)
            return np.dot(M, vec)
        elif(method.lower() == 'dot'):
            return np.dot(M, vec)

def dist_mat_mat(M1, M2, method = 'ncc'):
        # M1, M2 --> ndarray (y1 x k) and (y2 x k)
        D = np.zeros((M1.shape[0], M2.shape[0]))
        if(method.lower() == 'ncc'):
            M1 = mean_normalize(M1, axis = 1)
            M2 = mean_normalize(M2, axis = 1)
            method = 'dot'
        for idx2 in range(M2.shape[0]):
            D[:, idx2] = dist_mat_vec(M1, M2[idx2, :], method = method)
        return D
```

```
def filter_kps(kpA, kpB, featuresA, featuresB, method = 'ncc', thresh = 0.97):
    print len(featuresA), len(featuresB)
    if(method.lower() == 'ncc'):
        featuresA = mean_normalize(featuresA, axis = 1)
        featuresB = mean_normalize(featuresB, axis = 1)
        method = 'dot'

    matches = []
    for idxB in range(featuresB.shape[0]):
        temp = dist_mat_vec(featuresA, featuresB[idxB, :], method = method)
        temp[temp<thresh] = 0.0
        if(np.max(temp) == 0.0): continue
        else: matches.append((np.argmax(temp), idxB))

    return matches

def draw_matches(imageA, imageB, kpsA, kpsB, matches):
    # initialize the output visualization image
    (hA, wA) = imageA.shape[:2]
    (hB, wB) = imageB.shape[:2]
    vis = np.zeros((max(hA, hB), wA + wB, 3), dtype="uint8")
    vis[0:hA, 0:wA] = imageA
    vis[0:hB, wA:] = imageB

    # loop over the matches
    for queryIdx, trainIdx in matches:
        # only process the match if the keypoint was successfully
        # matched
        # draw the match
        ptA = (int(kpsA[queryIdx][0]), int(kpsA[queryIdx][1]))
        ptB = (int(kpsB[trainIdx][0]) + wA, int(kpsB[trainIdx][1]))
        cv2.line(vis, ptA, ptB, (0, 255, 0), 1)

    # return the visualization
    return vis

def run(image1_path, image2_path, ftype = 'sift', method = 'ncc', \
    thresh = 0.97, sigma = 1.414):
    img1 = cv2.imread(image1_path)
    img2 = cv2.imread(image2_path)

    kps1, features1 = extract_kps(img1, ftype = ftype, sigma = sigma)
    kps2, features2 = extract_kps(img2, ftype = ftype, sigma = sigma)

    start = time.time()
    matches = filter_kps(kps1, kps2, features1, features2, method = method, thresh = thresh)
    print 'Filter Kps: %.02f secs'%(time.time()-start)

    vis = draw_matches(img1, img2, kps1, kps2, matches)

    fname = splitext(basename(image1_path))[0] + '_' + splitext(basename(image2_path))[0]
    fname = fname + '_' + str(ftype) + '_' + str(int(sigma*1000)) + '_' + method + '.jpg'
```

```
    fname_path = join(dirname(image1_path), fname)
    print 'Writing to: ', fname_path

    cv2.imwrite(fname_path, vis)
    cv2.imshow('Visualization', vis)
    cv2.waitKey(0)

##########################
#          MAIN          #
##########################

if __name__ == '__main__':
    img1_path = 'pair3/1.jpg'
    img2_path = 'pair3/2.jpg'

    ftype = 'sift'
    method = 'ncc'
    thresh = 0.97 # SURF 0.9999

    vis = run(img1_path, img2_path, ftype = ftype, method=method, thresh = thresh)
```