# ECE 661: Computer Vision

Homework 4, Fall 2018

**Arindam Bhanja Chowdhury**

abhanjac@purdue.edu

## 1 Overview

This homework is about finding the interest points from two photos of the same object taken from different viewpoints and then automatically establish correspondence between the interest points in the two images. The pair of original images provided are shown in Fig 1, 2 and 3, 4.

Interest points are the points in the image that stays invariant to change in viewpoints, illumination, scale and x and y variations. So they can be very important to find out or localize an object in different scenes.

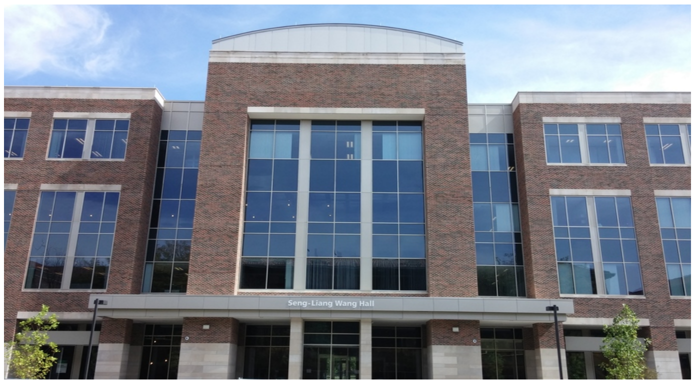

Figure 1: Reference image 1 of homework from viewpoint 1.

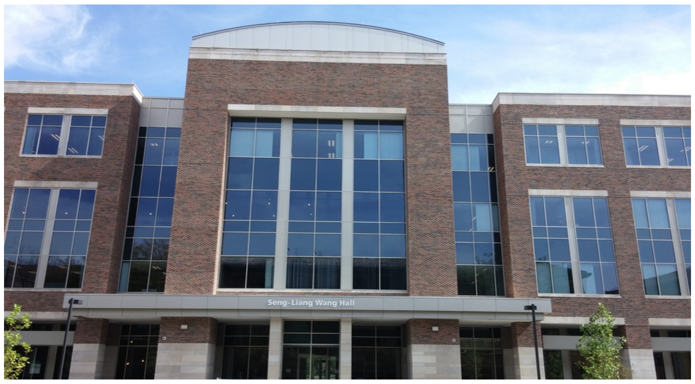

Figure 2: Reference image 1 of homework from viewpoint 2.

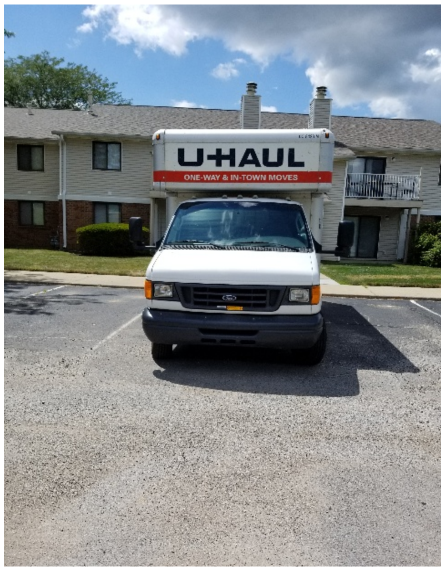

Figure 3: Reference image 2 of homework from viewpoint 1.

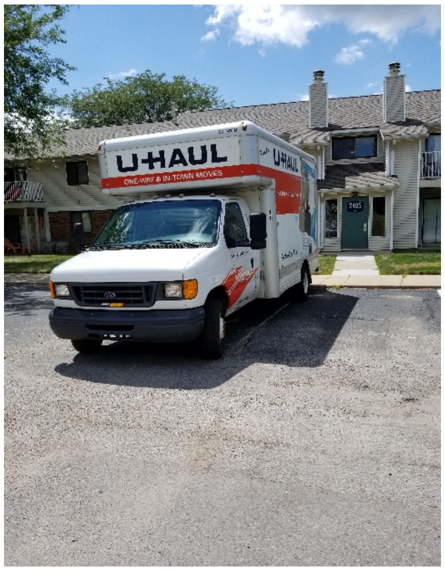

Figure 4: Reference image 2 of homework from viewpoint 2.

## 2 Harris Corner Interest Points

Harris corners are extracted from a grayscale image by finding the pixels near which there is a significant variation in both the x and y directions.

First the x-derivative ($D_x$) and y-derivative ($D_y$) images are extracted. For finding these, the convolution filter kernel used is calculated using Haar Wavelet Filters. The size of the kernel is defined as equal to the smallest even integer greater than $4\sigma$. So for $\sigma = 1.2$ the kernel size is $6 \times 6$. The filters used for $\sigma = 1.2$ to find the $D_x$ and $D_y$ images, by convolving with the grayscale version of the input image, are as follows.

$$D_x = \begin{bmatrix} -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \end{bmatrix}$$

$$D_y = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

Similar filters can be constructed for different $\sigma$ values.

Then for each pixel location, a $C$ matrix is constructed from the derivative values obtained from $D_x$ and $D_y$ images using the pixels from a $5\sigma \times 5\sigma$ neighbohood of the corresponding pixel. The value of $5\sigma$ is rounded up to the nearest odd interger. The $C$ matrix is shown in the following equation.

$$C_{2\times 2} = \begin{bmatrix} \Sigma D_x^2 & \Sigma D_x D_y \\ \Sigma D_x D_y & \Sigma D_y^2 \end{bmatrix} \tag{1}$$

The $D_x^2$ is the square of the $D_x$ value, not the double derivative with respect to x. This is a symmetric matrix. Now in a real corned there is always variation in both the x and y directions. So the $D_x D_y$ terms are non-zero in the corresponding $C$ matrix. Hence, at a proper corner, this $C$ matrix will have a rank of 2. For example, near the edges, along the y direction, the $D_y$ terms will be zero and hence the rank of the $C$ matrix is not 2 as the terms with $D_y$ in them will be all zeros. So, at a proper corner, the $C$ matrix will have two eigen values. These eigen values are found at each pixel location from the corresponding $C$ matrix and then the ratio $\lambda_2/\lambda_1$ (assuming that $\lambda_1 >= \lambda_2$). This ratio is can be used as a threshold to find good corners. However, explicit calculation of the eigen values is not needed, as the the trace and determinant of $C$ can be used to create another parameter which can instead be used as a threshold ($k$).

$$k = \frac{Det(C)}{Trace(C)^2} = \frac{\lambda_1 \lambda_2}{(\lambda_1 + \lambda_2)^2} \tag{2}$$

So, after calculating the determinant and trace at every pixel location, an $k$ value is calculated as the average of the ratio $Det(C)/Trace(C)^2$ at all pixel locations in the image. This $k$ is then used to rewrite the equation 2 as follows,

$$Det(C) - k\ Trace(C)^2 >= 0 \tag{3}$$

Now, the LHS of the equation 2 can be written as $R$ and this is the Harris corner detector Response image. Only the points in this image which are greater than zero (as also shown by the equation) are potential corners.

Now, there can be a number of clusters of pixels which are very bright in this $R$ image. So a non-maximum suppression operation is done by sliding a window over $R$, to replace all the high pixel values with zeros and only keeping the brightest pixel alive. This final image gives the Harris corners.

The Harris corners from two viewpoints of an image are compared using Sum of Squared Differences (SSD) or Normalized Cross Correlation (NCC) metric and only those which can pass a certain threshold are kept as good corners. The expressions for SSD and NCC are given as follows.

$$SSD = \Sigma_x \Sigma_y (f_1(x,y) - f_2(x,y))^2 \tag{4}$$

$$NCC = \frac{\Sigma_x \Sigma_y (f_1(x,y) - m_1)(f_2(x,y) - m_2)}{\sqrt{(\Sigma_x \Sigma_y (f_1(x,y) - m_1)^2)(\Sigma_x \Sigma_y (f_2(x,y) - m_2)^2)}} \tag{5}$$

## 3 Procedure of Harris Corner Detection

Harris corners are detected in the images pairs shown earlier in the Fig, 1, 2, 3, 4 in the same manner as explained in the theory. The matching corners are extracted based on the SSD and NCC values. These are shown in the result section. The neighborhood used to perform non-maximum suppression is $29 \times 29$ and the neighborhood used to calculate the SSD and NCC values are $21 \times 21$ in size.

The Harris Corner detector is also tested on two separate image pairs other than the ones given in the problem set. Those are also shown in the result section.

## 4 Results of Harris Corner Detection

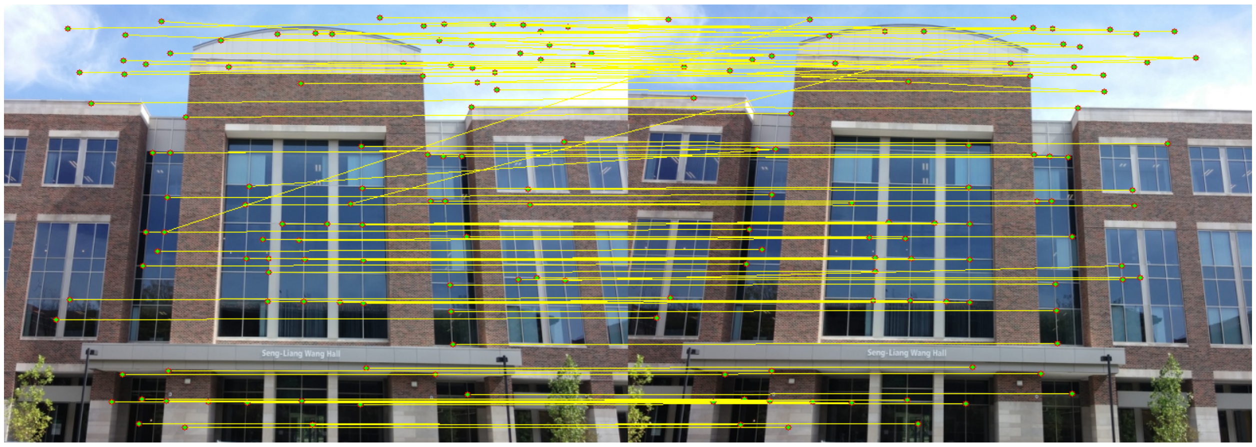

Figure 5: SSD on the Harris Corners on image pair 1 with $\sigma = 0.707$.

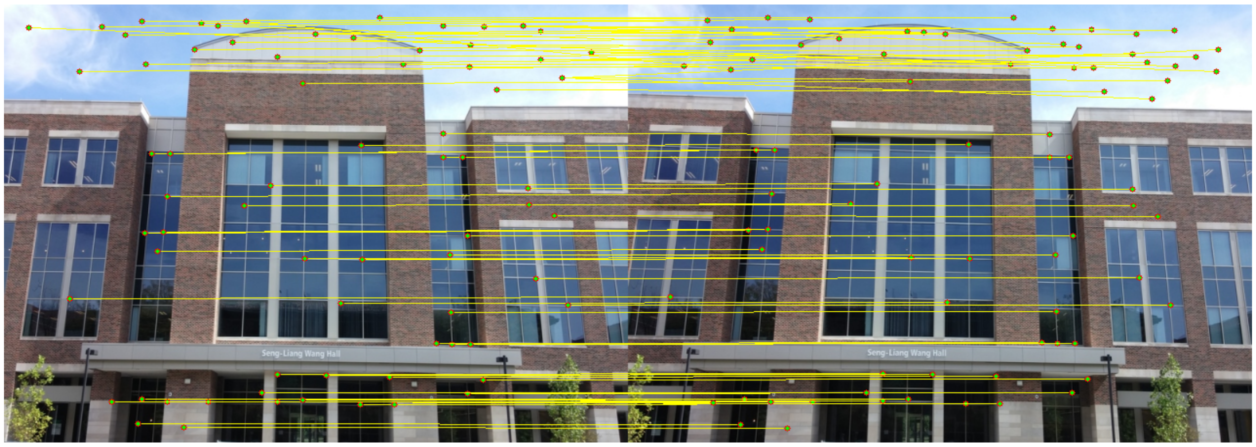

Figure 6: SSD on the Harris Corners on image pair 1 with $\sigma = 1$.

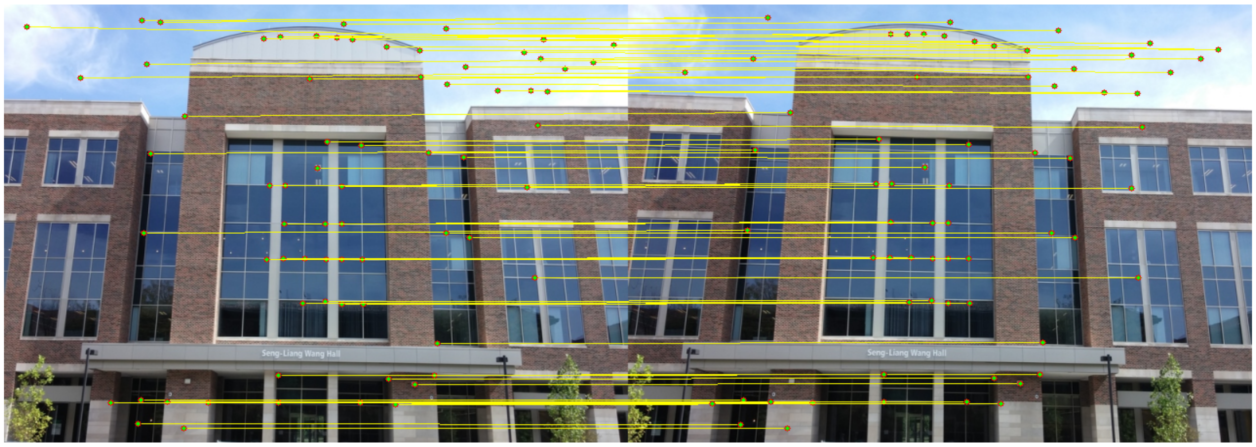

Figure 7: SSD on the Harris Corners on image pair 1 with $\sigma = 1.414$.

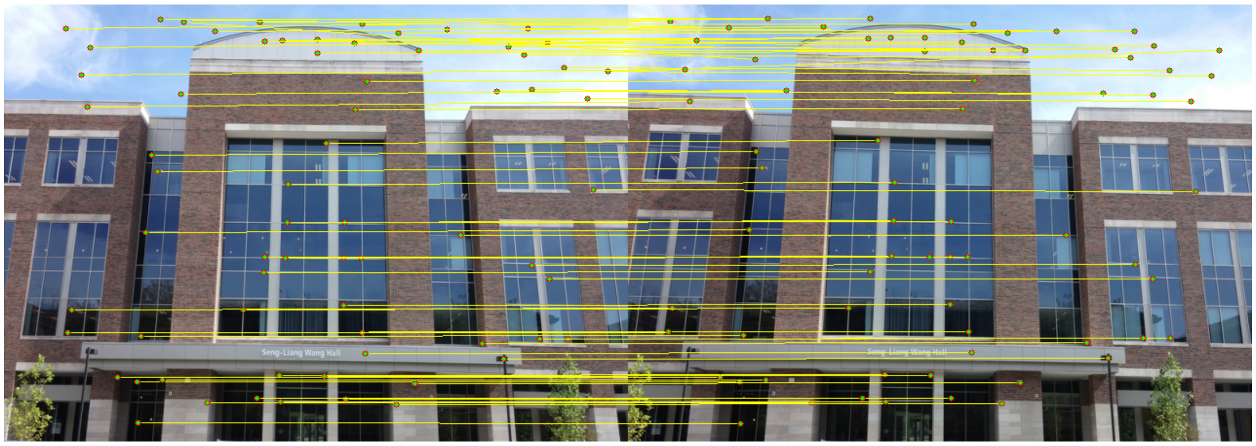

Figure 8: SSD on the Harris Corners on image pair 1 with $\sigma = 2$.

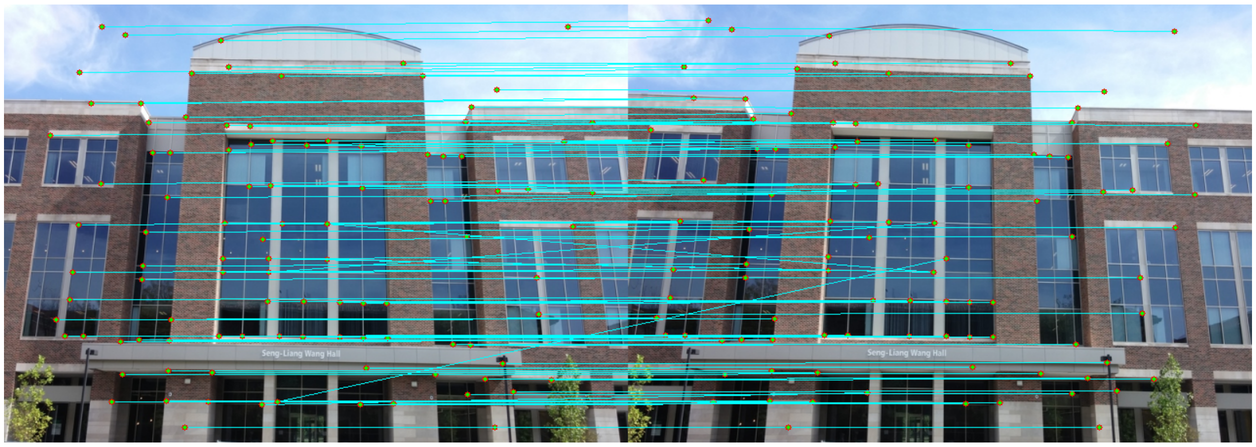

Figure 9: NCC on the Harris Corners on image pair 1 with $\sigma = 0.707$.

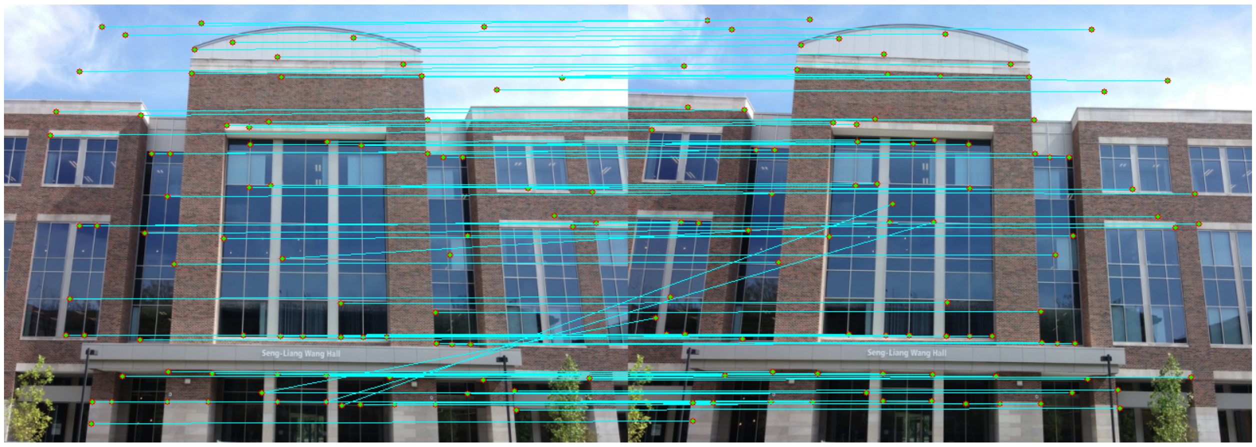

Figure 10: NCC on the Harris Corners on image pair 1 with $\sigma = 1$.

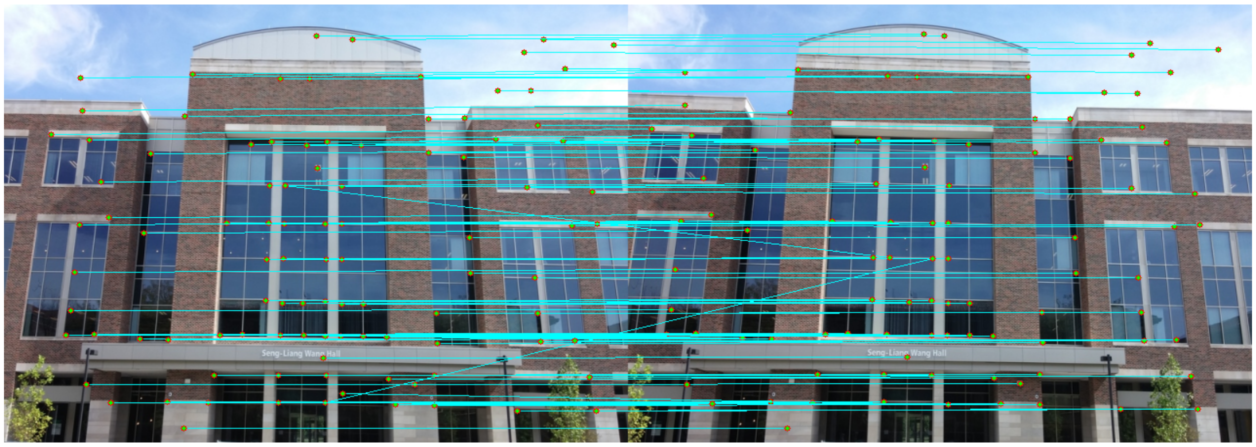

Figure 11: NCC on the Harris Corners on image pair 1 with $\sigma = 1.414$.

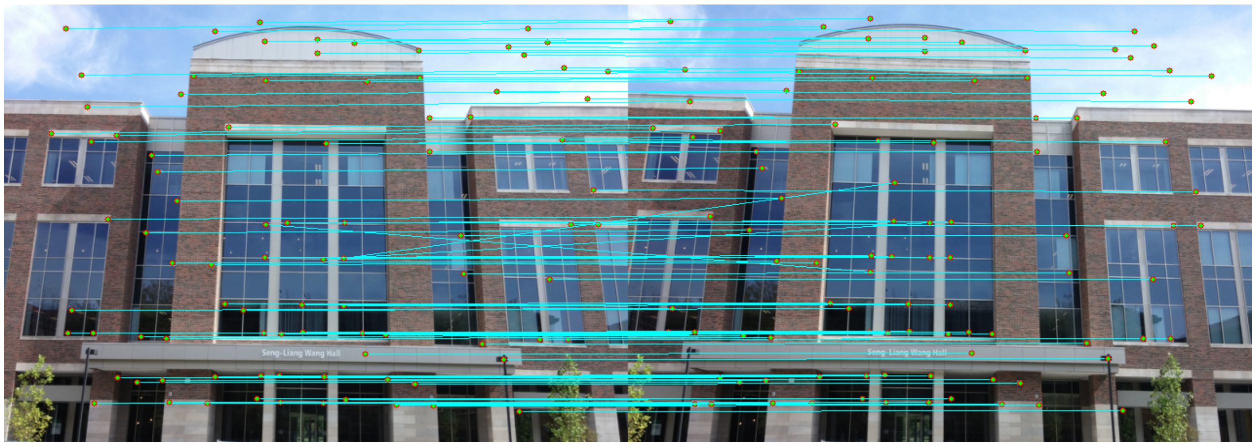

Figure 12: NCC on the Harris Corners on image pair 1 with $\sigma = 2$.

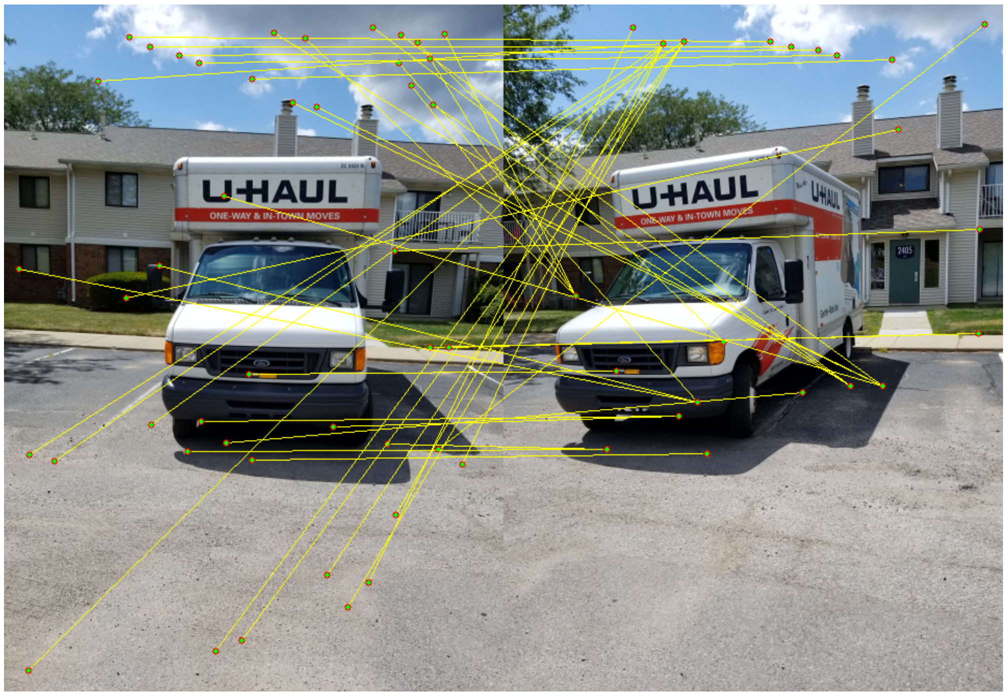

Figure 13: SSD on the Harris Corners on image pair 2 with $\sigma = 0.707$.

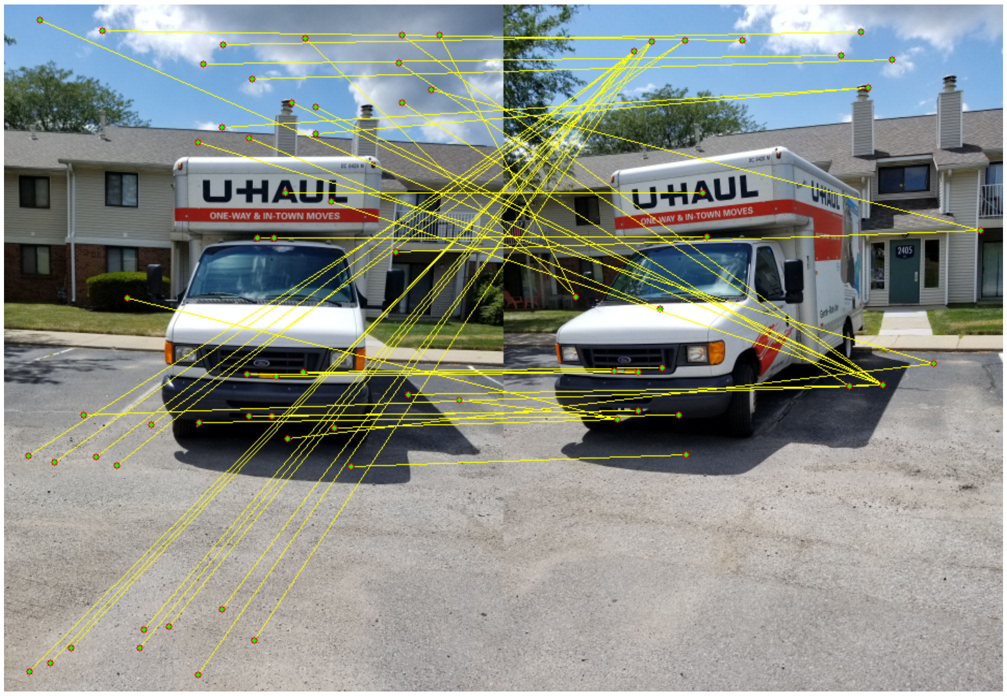

Figure 14: SSD on the Harris Corners on image pair 2 with $\sigma = 1$.

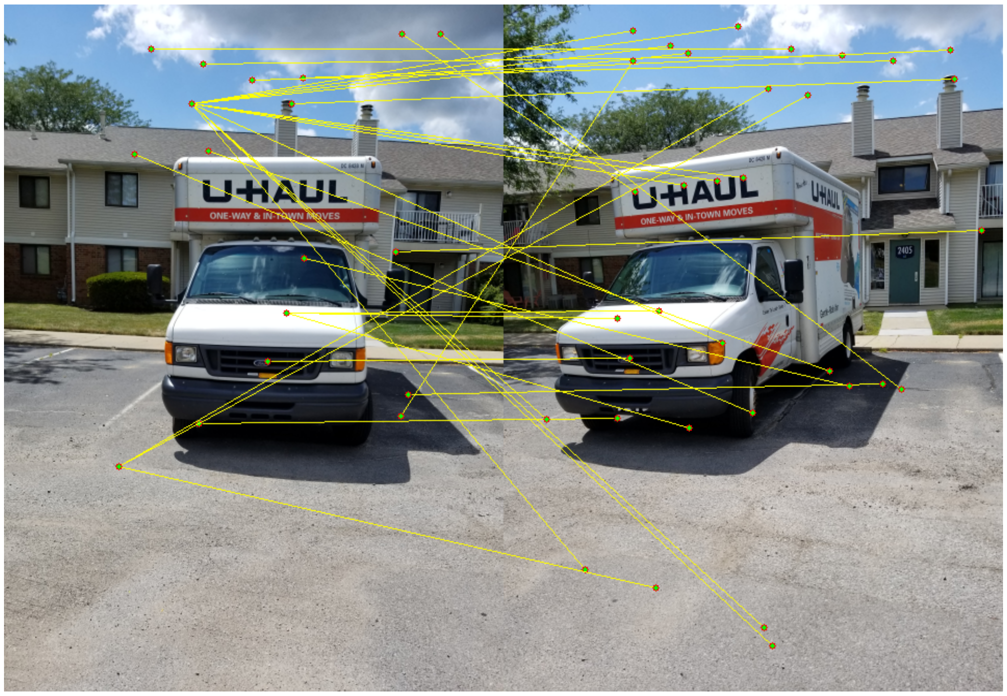

Figure 15: SSD on the Harris Corners on image pair 2 with $\sigma = 1.414$.

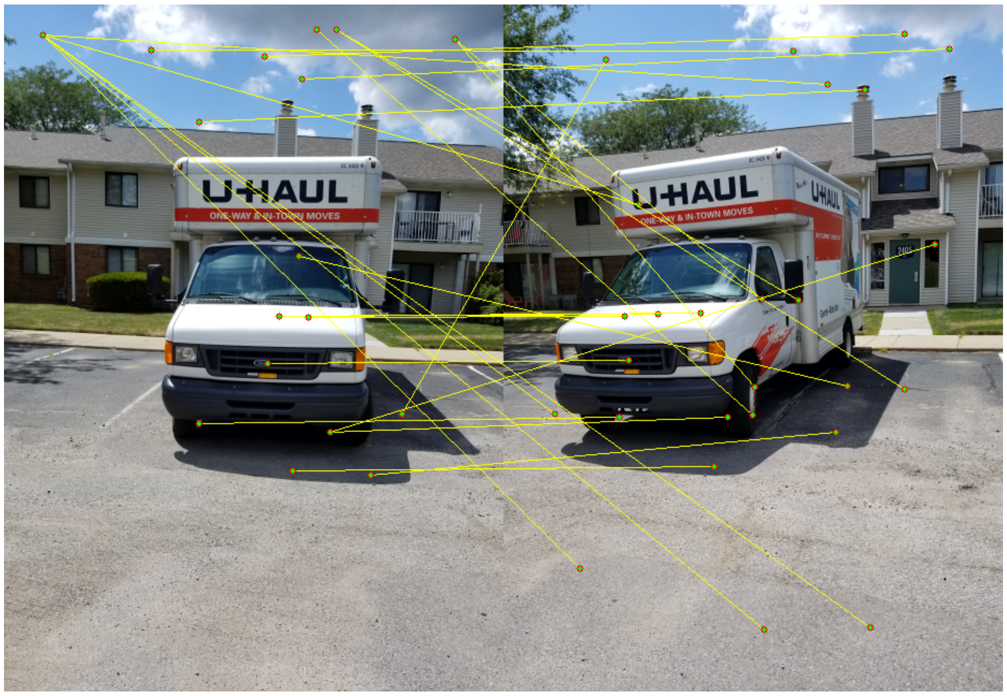

Figure 16: SSD on the Harris Corners on image pair 2 with $\sigma = 2$.

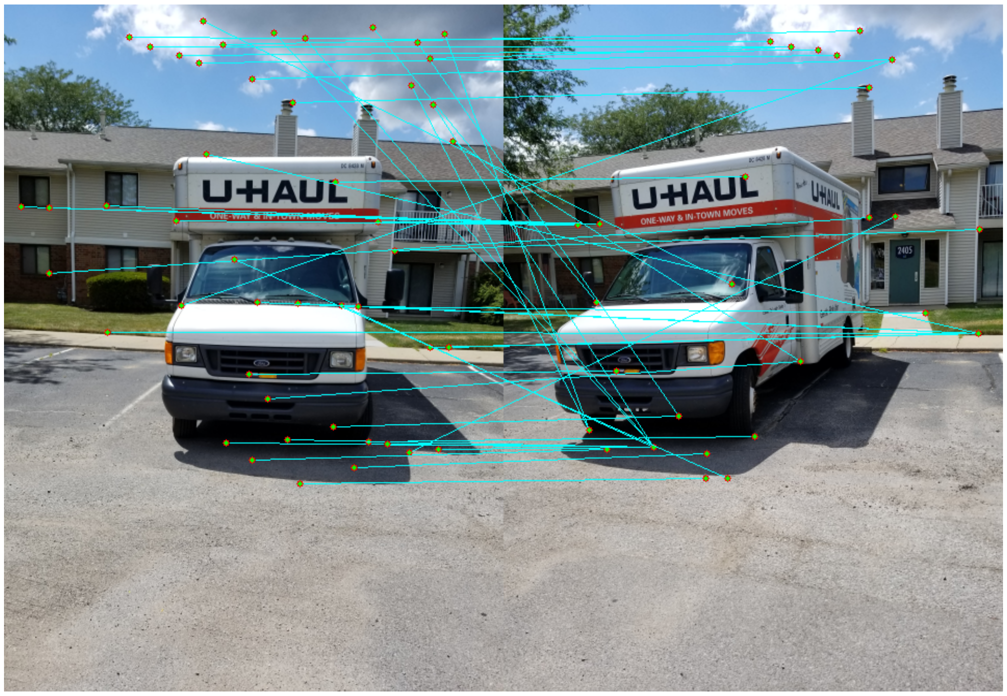

Figure 17: NCC on the Harris Corners on image pair 2 with $\sigma = 0.707$.

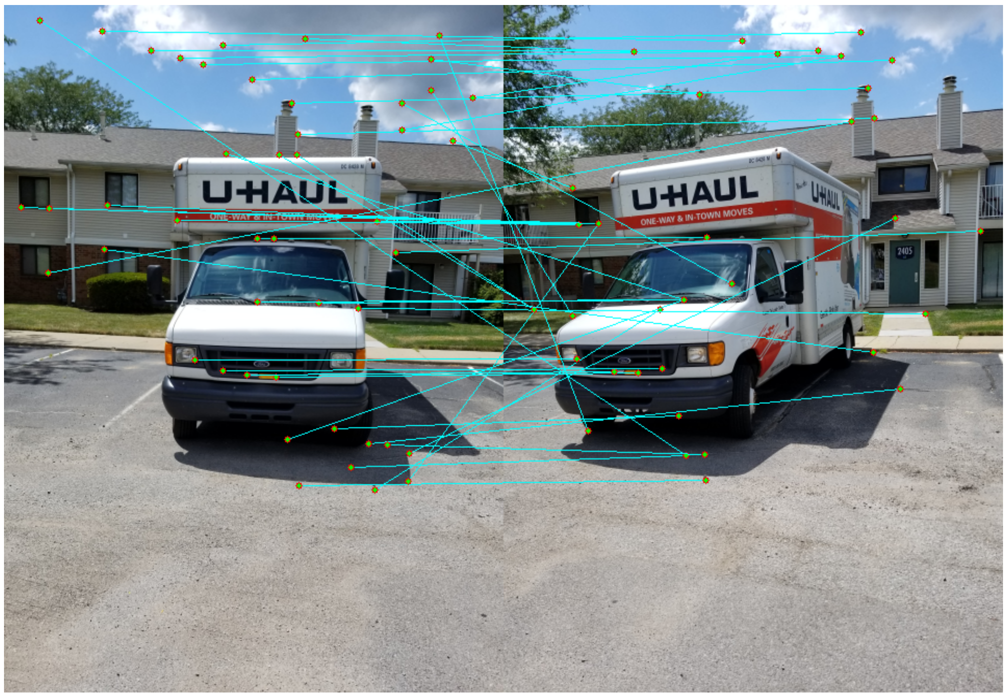

Figure 18: NCC on the Harris Corners on image pair 2 with $\sigma = 1$.

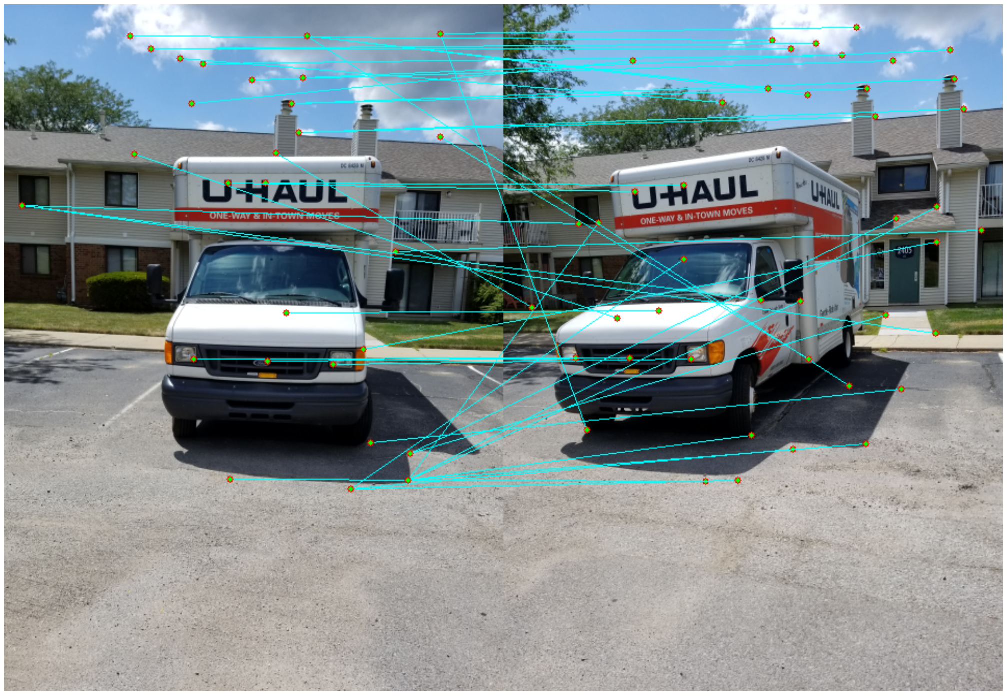

Figure 19: NCC on the Harris Corners on image pair 2 with $\sigma = 1.414$.

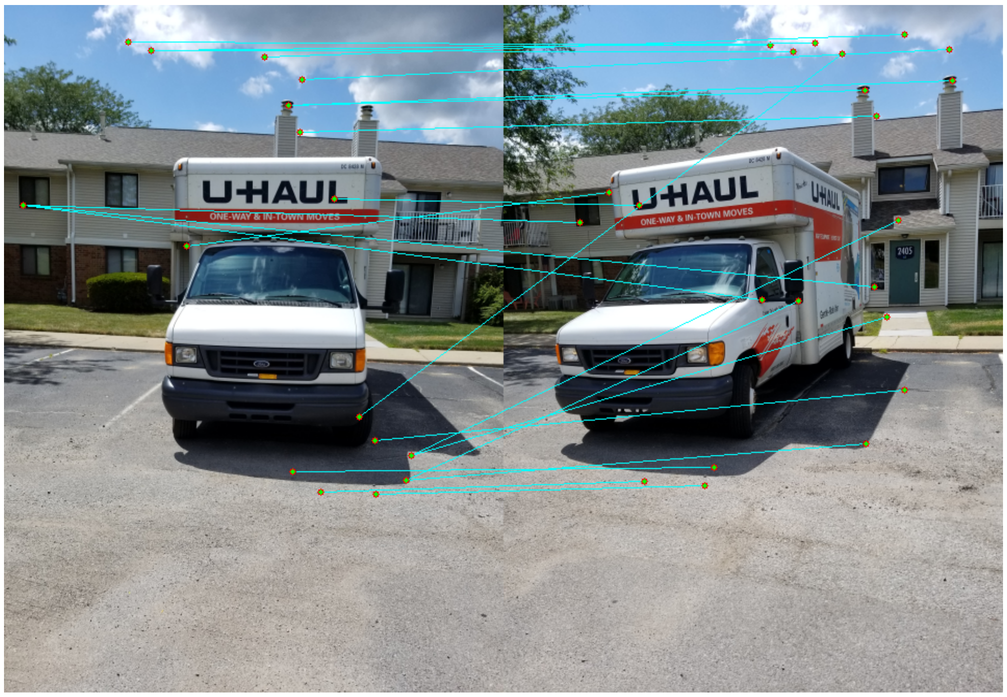

Figure 20: NCC on the Harris Corners on image pair 2 with $\sigma = 2$.

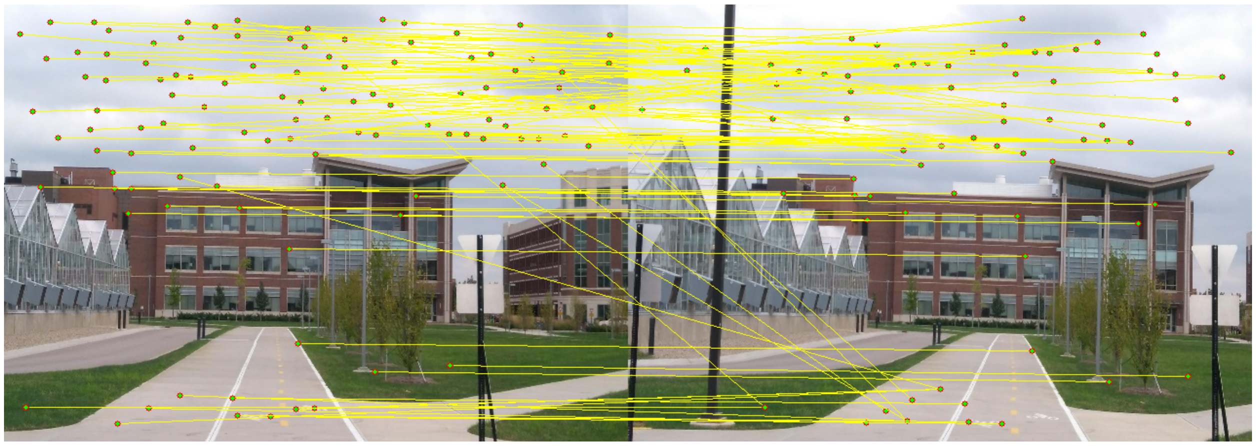

Figure 21: SSD on the Harris Corners on image pair 3 with $\sigma = 0.707$.

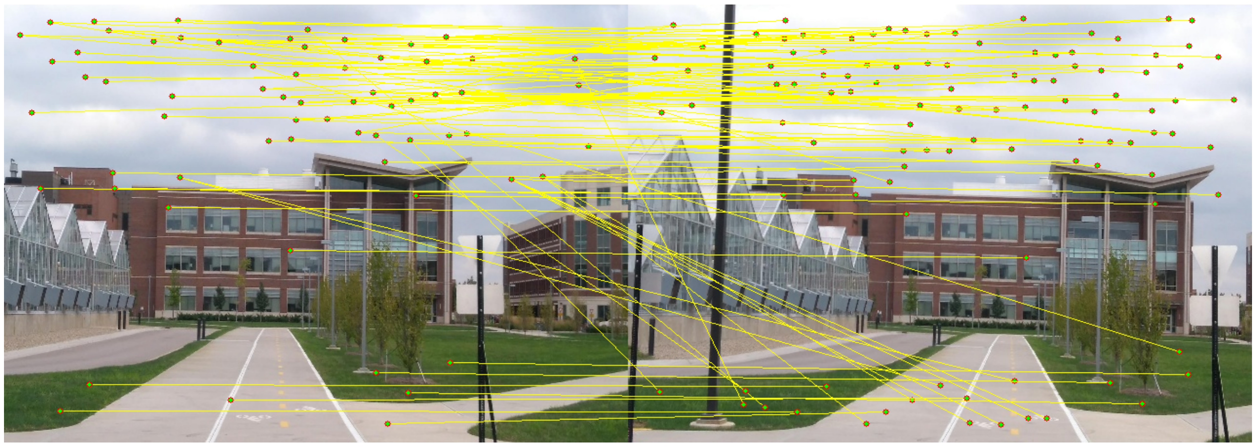

Figure 22: SSD on the Harris Corners on image pair 3 with $\sigma = 1$.

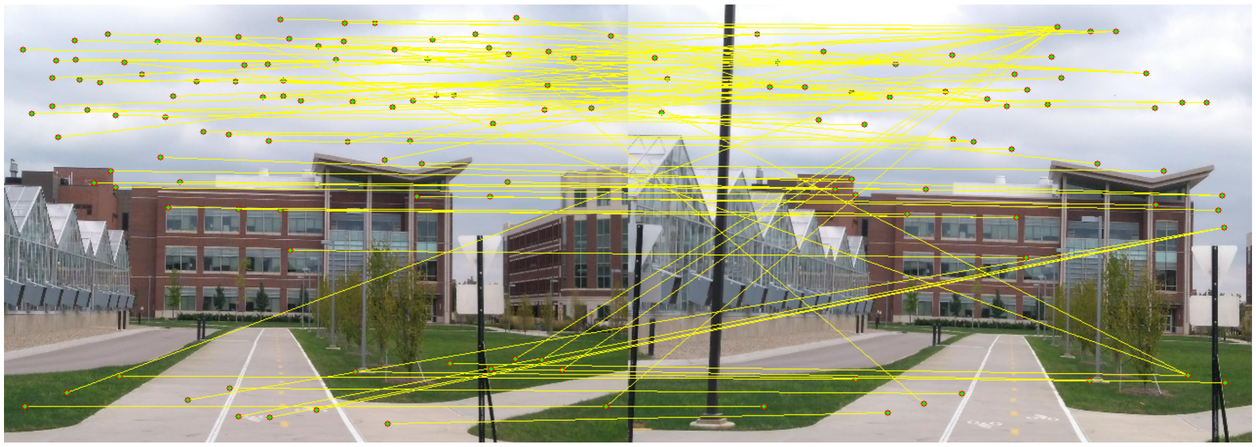

Figure 23: SSD on the Harris Corners on image pair 3 with $\sigma = 1.414$.

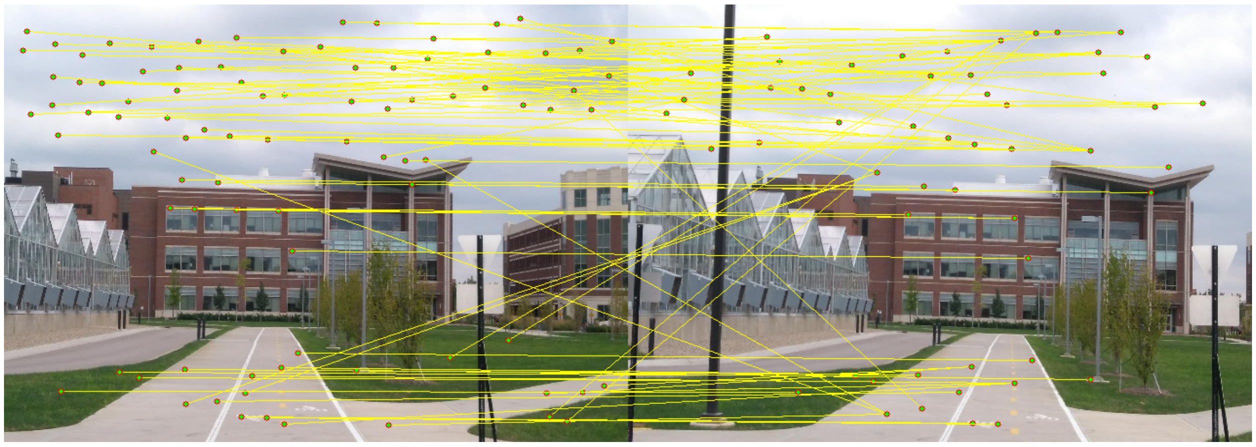
Figure 24: SSD on the Harris Corners on image pair 3 with $\sigma = 2$.

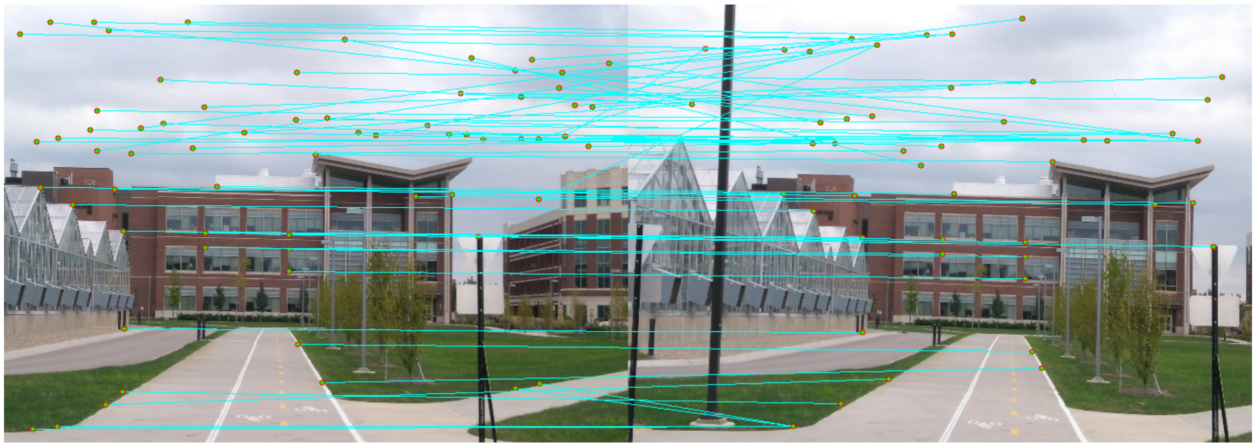
Figure 25: NCC on the Harris Corners on image pair 3 with $\sigma = 0.707$.

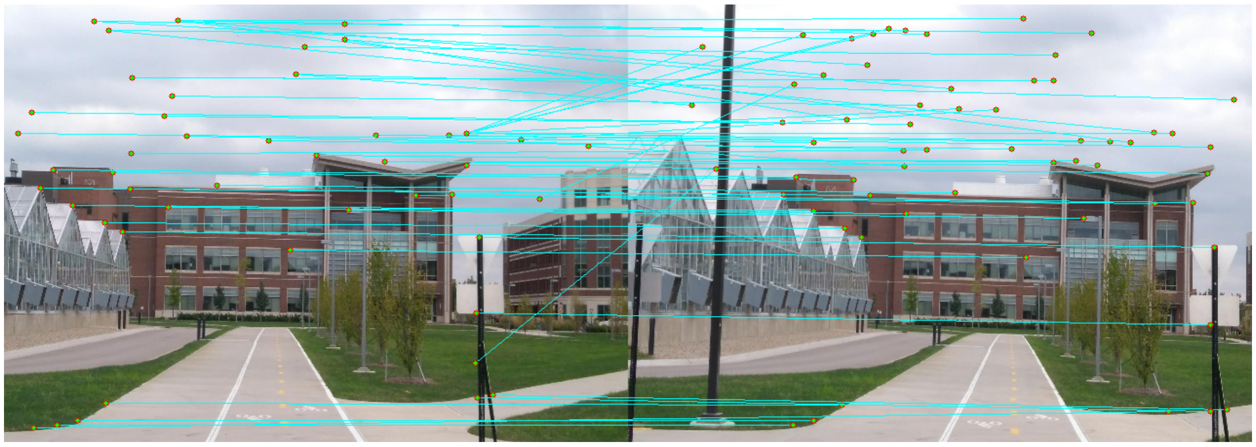
Figure 26: NCC on the Harris Corners on image pair 3 with $\sigma = 1$.

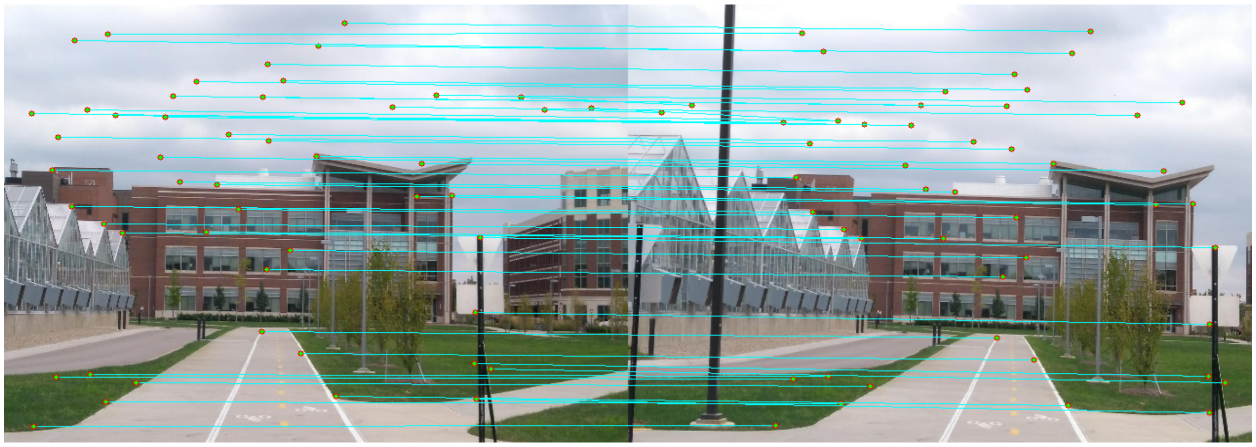

Figure 27: NCC on the Harris Corners on image pair 3 with $\sigma = 1.414$.

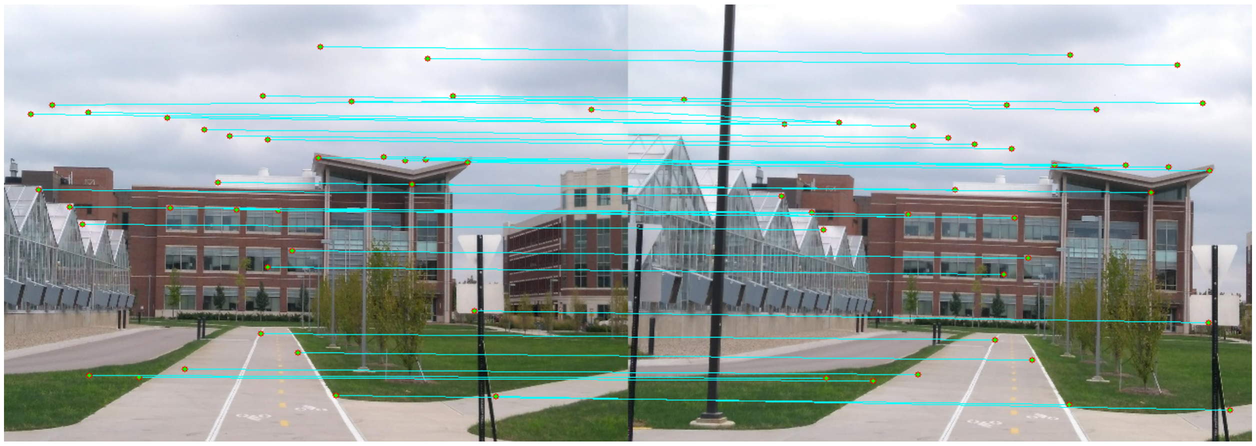

Figure 28: NCC on the Harris Corners on image pair 3 with $\sigma = 2$.

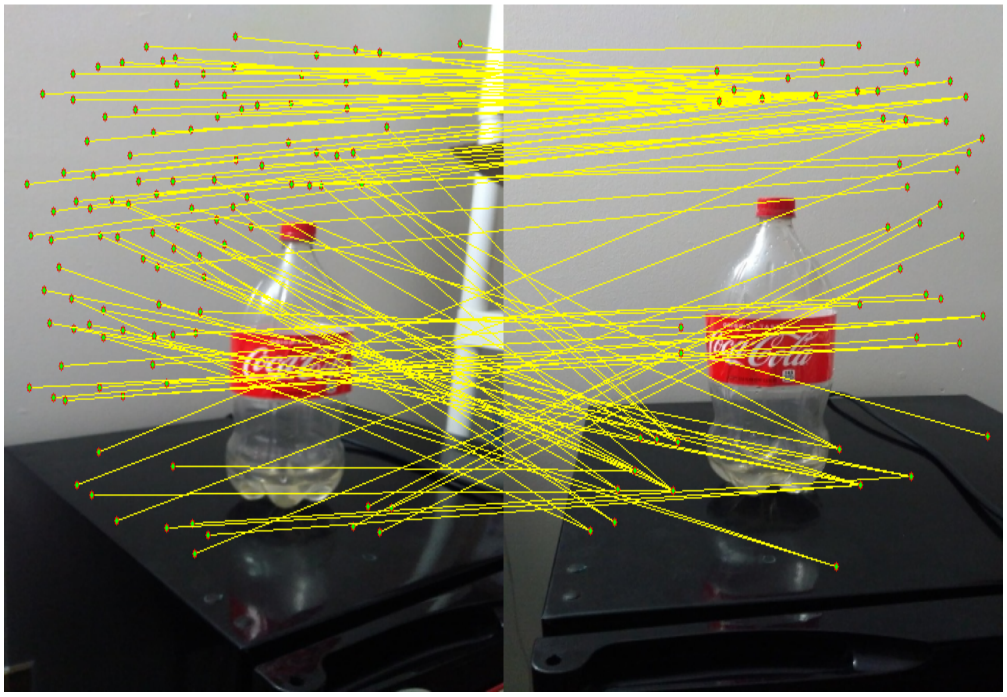

Figure 29: SSD on the Harris Corners on image pair 4 with $\sigma = 0.707$.

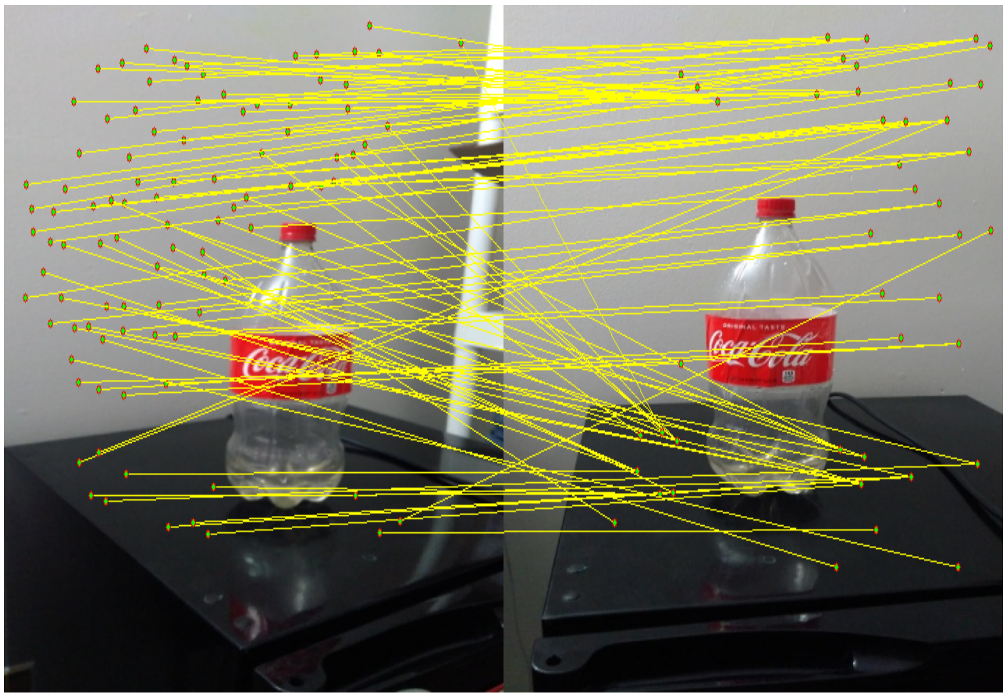

Figure 30: SSD on the Harris Corners on image pair 4 with $\sigma = 1$.

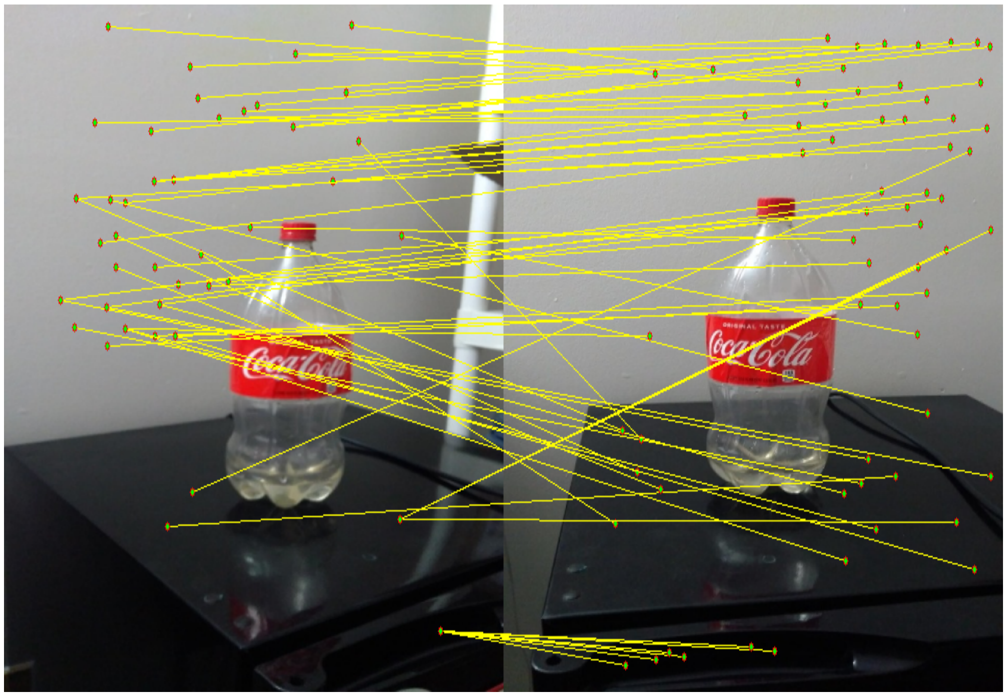
Figure 31: SSD on the Harris Corners on image pair 4 with $\sigma = 1.414$.

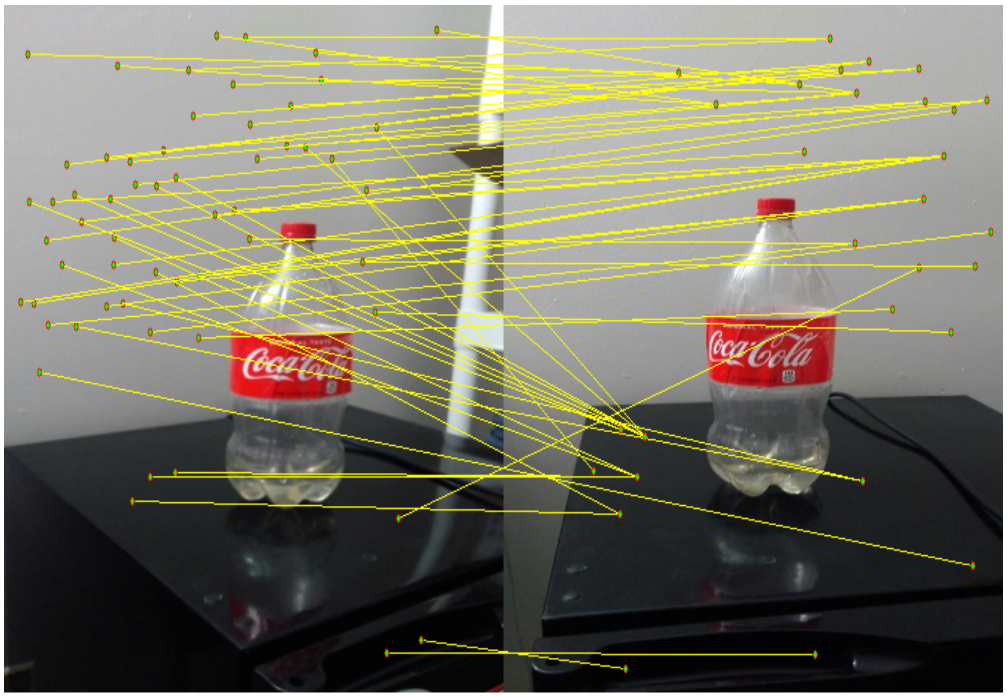
Figure 32: SSD on the Harris Corners on image pair 4 with $\sigma = 2$.

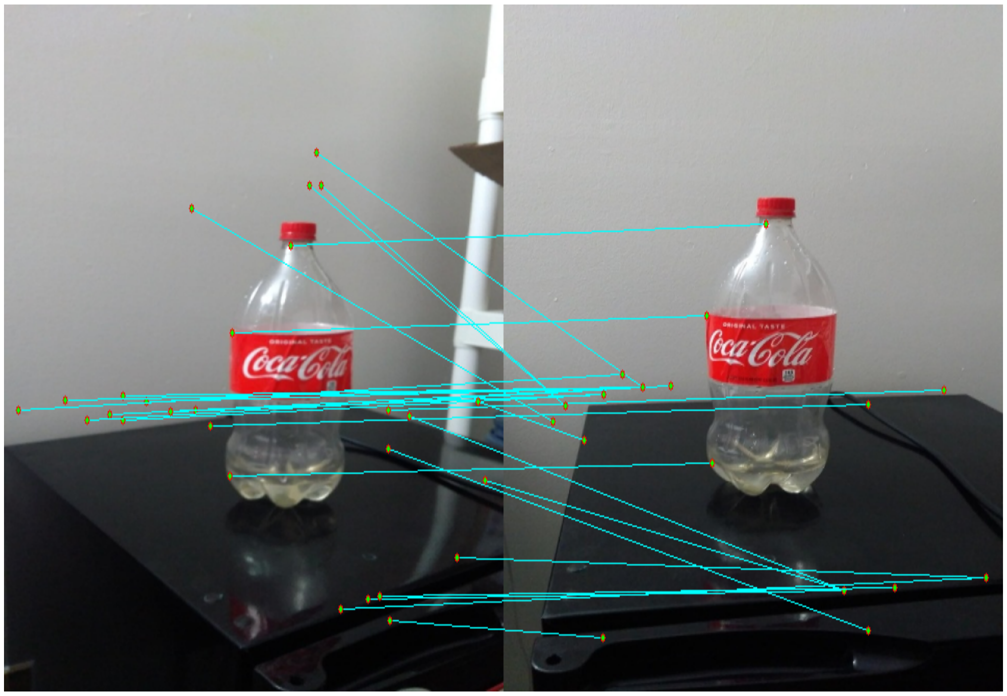

Figure 33: NCC on the Harris Corners on image pair 4 with $\sigma = 0.707$.

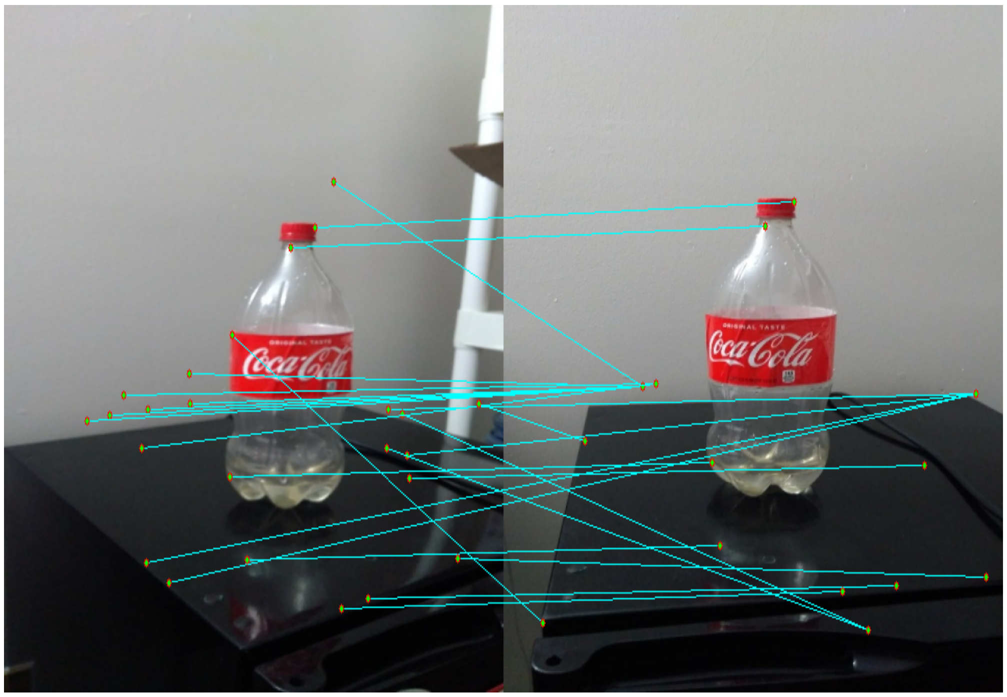

Figure 34: NCC on the Harris Corners on image pair 4 with $\sigma = 1$.

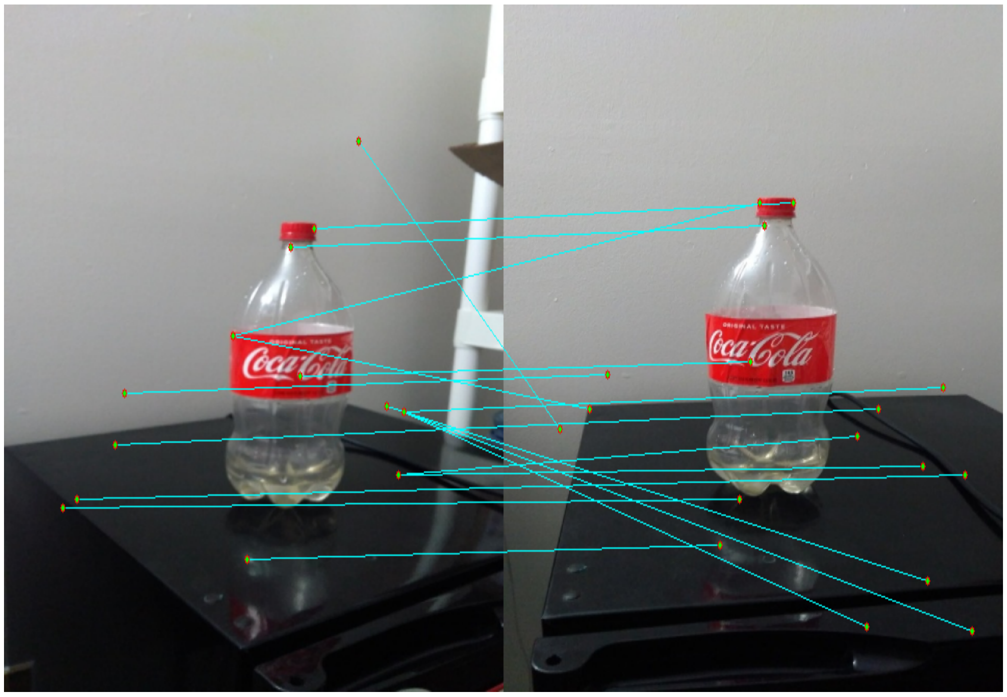

Figure 35: NCC on the Harris Corners on image pair 4 with $\sigma = 1.414$.

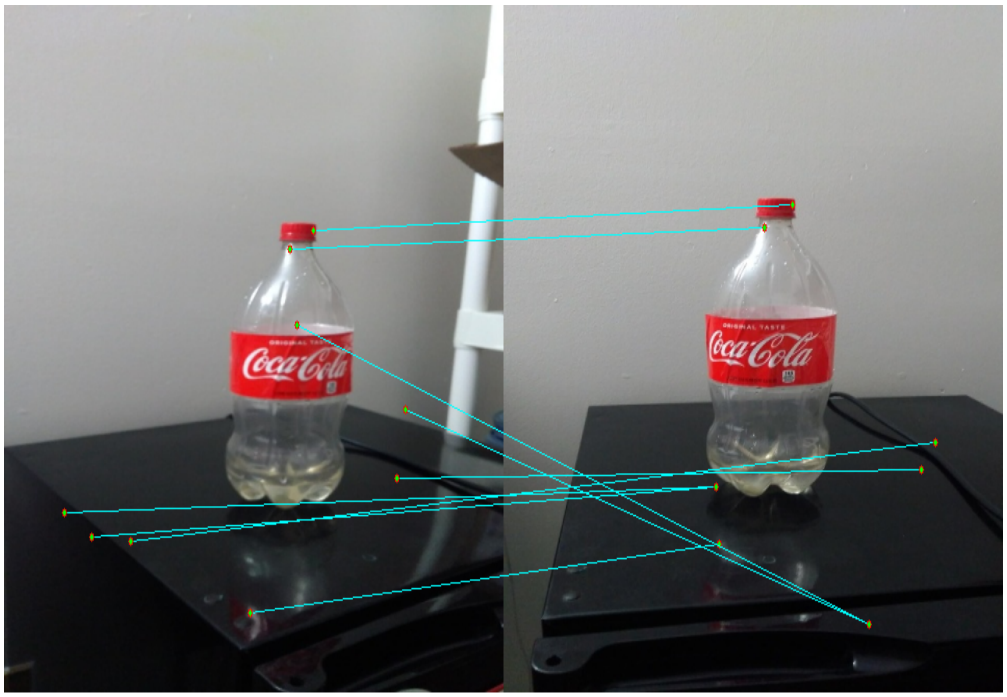

Figure 36: NCC on the Harris Corners on image pair 4 with $\sigma = 2$.

## 5 Discussion and Overview of Results

An overview of the results of Harris corner detection is also given in the following tables. Note that the k value and the number of corners found is not dependent on SSD or NCC as they are calculated before applying the SSD and NCC and applying the thresholds.

| | **Pair 1 Image 1** | **Pair 1 Image 2** | **Pair 2 Image 1** | **Pair 2 Image 2** |
|---|---|---|---|---|
| **Method** | SSD | SSD | SSD | SSD |
| **Threshold** | 12000 | 12000 | 25000 | 25000 |
| **k value** ($\sigma = 0.707$) | 0.083 | 0.082 | 0.105 | 0.112 |
| **Corners Found** ($\sigma = 0.707$) | 260 | 266 | 240 | 255 |
| **Good Matches** ($\sigma = 0.707$) | 96 | 96 | 44 | 44 |
| **k value** ($\sigma = 1$) | 0.102 | 0.101 | 0.130 | 0.138 |
| **Corners Found** ($\sigma = 1$) | 253 | 247 | 221 | 229 |
| **Good Matches** ($\sigma = 1$) | 78 | 78 | 49 | 49 |
| **k value** ($\sigma = 1.414$) | 0.095 | 0.093 | 0.121 | 0.128 |
| **Corners Found** ($\sigma = 1.414$) | 214 | 217 | 207 | 206 |
| **Good Matches** ($\sigma = 1.414$) | 70 | 70 | 33 | 33 |
| **k value** ($\sigma = 2$) | 0.101 | 0.099 | 0.122 | 0.129 |
| **Corners Found** ($\sigma = 2$) | 205 | 204 | 180 | 171 |
| **Good Matches** ($\sigma = 2$) | 71 | 71 | 26 | 26 |

Table 1: Overview of performance for SSD on image pair 1 and pair 2

| | **Pair 1 Image 1** | **Pair 1 Image 2** | **Pair 2 Image 1** | **Pair 2 Image 2** |
|---|---|---|---|---|
| **Method** | NCC | NCC | NCC | NCC |
| **Threshold** | 0.96 | 0.96 | 0.82 | 0.82 |
| **k value** ($\sigma = 0.707$) | 0.083 | 0.082 | 0.105 | 0.112 |
| **Corners Found** ($\sigma = 0.707$) | 260 | 266 | 240 | 255 |
| **Good Matches** ($\sigma = 0.707$) | 102 | 102 | 47 | 47 |
| **k value** ($\sigma = 1$) | 0.102 | 0.101 | 0.130 | 0.138 |
| **Corners Found** ($\sigma = 1$) | 253 | 247 | 221 | 229 |
| **Good Matches** ($\sigma = 1$) | 89 | 89 | 46 | 46 |
| **k value** ($\sigma = 1.414$) | 0.095 | 0.093 | 0.121 | 0.128 |
| **Corners Found** ($\sigma = 1.414$) | 214 | 217 | 207 | 206 |
| **Good Matches** ($\sigma = 1.414$) | 99 | 99 | 43 | 43 |
| **k value** ($\sigma = 2$) | 0.101 | 0.099 | 0.122 | 0.129 |
| **Corners Found** ($\sigma = 2$) | 205 | 204 | 180 | 171 |
| **Good Matches** ($\sigma = 2$) | 95 | 95 | 22 | 22 |

Table 2: Overview of performance for NCC on image pair 1 and pair 2

| | **Pair 3 Image 1** | **Pair 3 Image 2** | **Pair 4 Image 1** | **Pair 4 Image 2** |
|---|---|---|---|---|
| **Method** | SSD | SSD | SSD | SSD |
| **Threshold** | 14000 | 14000 | 1200 | 1200 |
| **k value** ($\sigma = 0.707$) | 0.078 | 0.076 | 0.099 | 0.117 |
| **Corners Found** ($\sigma = 0.707$) | 257 | 264 | 267 | 271 |
| **Good Matches** ($\sigma = 0.707$) | 107 | 107 | 105 | 105 |
| **k value** ($\sigma = 1$) | 0.1 | 0.096 | 0.126 | 0.144 |
| **Corners Found** ($\sigma = 1$) | 251 | 248 | 235 | 253 |
| **Good Matches** ($\sigma = 1$) | 106 | 106 | 94 | 94 |
| **k value** ($\sigma = 1.414$) | 0.086 | 0.084 | 0.114 | 0.132 |
| **Corners Found** ($\sigma = 1.414$) | 211 | 225 | 232 | 224 |
| **Good Matches** ($\sigma = 1.414$) | 89 | 89 | 63 | 63 |
| **k value** ($\sigma = 2$) | 0.086 | 0.085 | 0.102 | 0.119 |
| **Corners Found** ($\sigma = 2$) | 174 | 180 | 167 | 180 |
| **Good Matches** ($\sigma = 2$) | 88 | 88 | 57 | 57 |

Table 3: Overview of performance for SSD on image pair 3 and pair 4

| | **Pair 3 Image 1** | **Pair 3 Image 2** | **Pair 4 Image 1** | **Pair 4 Image 2** |
|---|---|---|---|---|
| **Method** | NCC | NCC | NCC | NCC |
| **Threshold** | 0.95 | 0.95 | 0.9 | 0.9 |
| **k value** ($\sigma = 0.707$) | 0.078 | 0.076 | 0.099 | 0.117 |
| **Corners Found** ($\sigma = 0.707$) | 257 | 264 | 267 | 271 |
| **Good Matches** ($\sigma = 0.707$) | 65 | 65 | 27 | 27 |
| **k value** ($\sigma = 1$) | 0.1 | 0.096 | 0.126 | 0.144 |
| **Corners Found** ($\sigma = 1$) | 251 | 248 | 235 | 253 |
| **Good Matches** ($\sigma = 1$) | 61 | 61 | 24 | 24 |
| **k value** ($\sigma = 1.414$) | 0.086 | 0.084 | 0.114 | 0.132 |
| **Corners Found** ($\sigma = 1.414$) | 211 | 225 | 232 | 224 |
| **Good Matches** ($\sigma = 1.414$) | 49 | 49 | 17 | 17 |
| **k value** ($\sigma = 2$) | 0.086 | 0.085 | 0.102 | 0.119 |
| **Corners Found** ($\sigma = 2$) | 174 | 180 | 167 | 180 |
| **Good Matches** ($\sigma = 2$) | 36 | 36 | 9 | 9 |

Table 4: Overview of performance for NCC on image pair 3 and pair 4

## 6 Overview of SIFT

This will be a very brief overview of the Scale Invariant Feature Transform (SIFT) algorithm. This is one of the most popular interest point detectors. It finds the interest points using the following steps.

- **Constructing a scale space:** This is the initial preparation. We create internal representations of the original image to ensure scale invariance. This is done by generating a ”scale space”.
- **LOG Approximation:** The Laplacian of Gaussian is great for finding interesting points (or key points) in an image. But it’s computationally expensive. So we approximate it using the Difference of Gaussian (DOG).
- **Finding keypoints:** With the super fast approximation, we now try to find key points. These are maxima and minima in the Difference of Gaussian image we calculate in step 2.
- **Get rid of bad key points** Edges and low contrast regions are bad keypoints. Eliminating these makes the algorithm efficient and robust. A technique similar to the Harris Corner Detector is used here.
- **Assigning an orientation to the keypoints** An orientation is calculated for each key point. Any further calculations are done relative to this orientation. This effectively cancels out the effect of orientation, making it rotation invariant.
- **Generate SIFT features** Finally, with scale and rotation invariance in place, one more representation is generated. This helps uniquely identify features. Lets say you have 50,000 features. With this representation, you can easily identify the feature you’re looking for.

That was an overview of the entire algorithm. We will be using the SIFT package in opencv for implementing the algorithm.

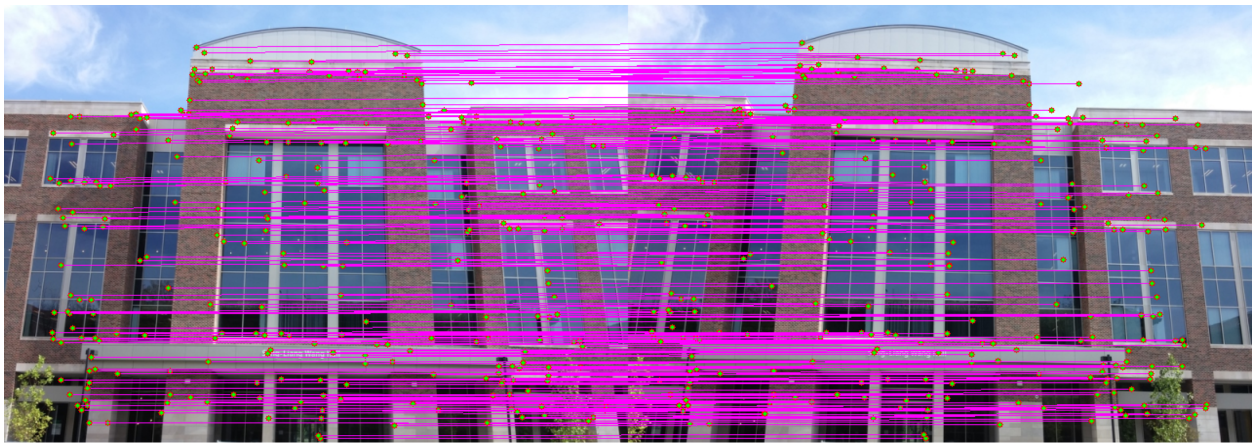

Figure 37: SIFT on image pair 1.

Number of good matches for Sift is 276 out of 961 total matches (with threshold: 50).

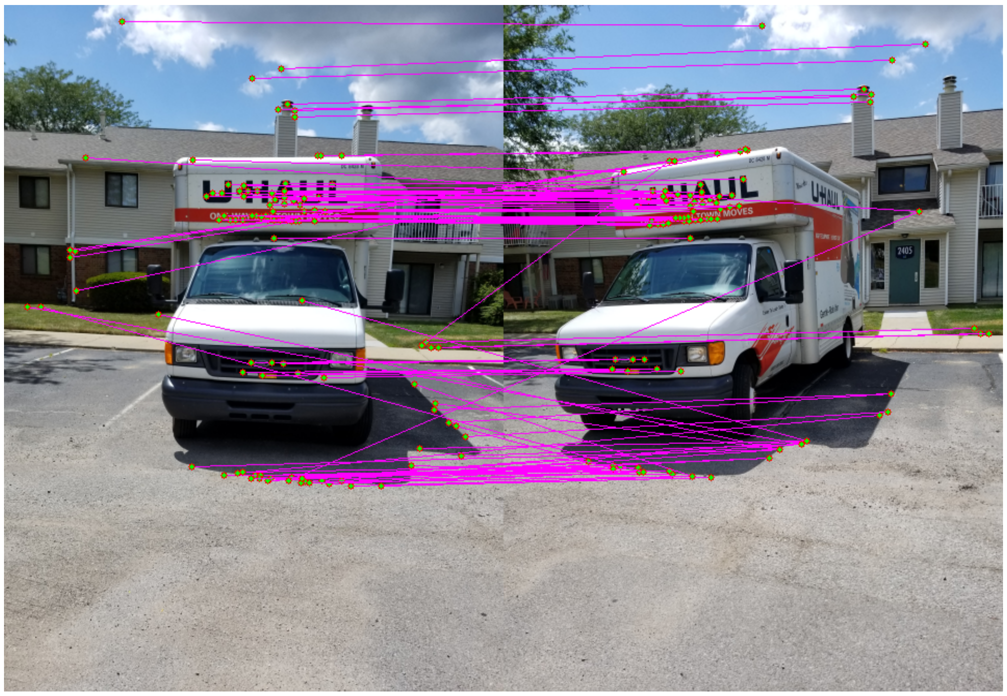

Figure 38: SIFT on image pair 2.

Number of good matches for Sift is 123 out of 991 total matches (with threshold: 125).

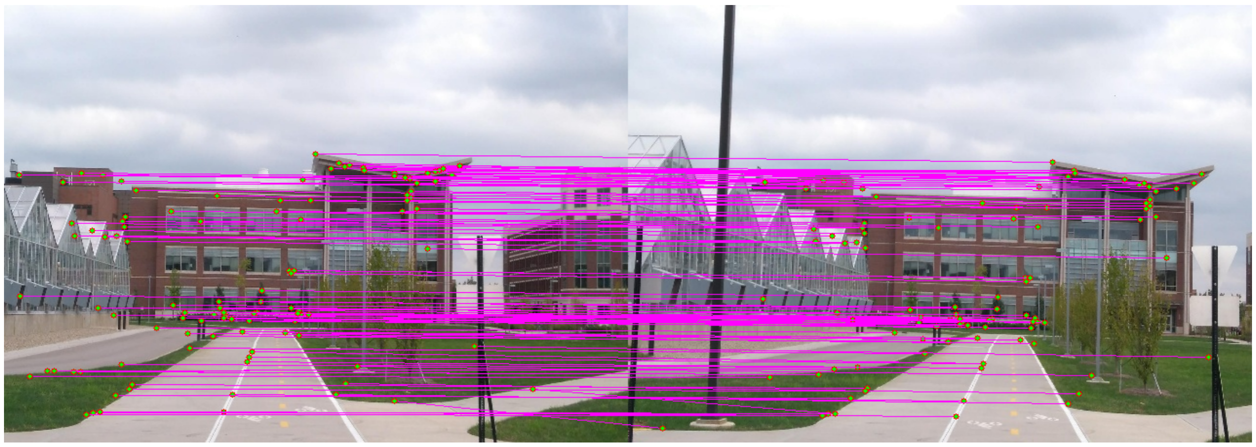

Figure 39: SIFT on image pair 3.

Number of good matches for Sift: 115 out of 924 total matches (with threshold: 75).
Number of good matches for Sift: 41 out of 191 total matches (with threshold: 230).

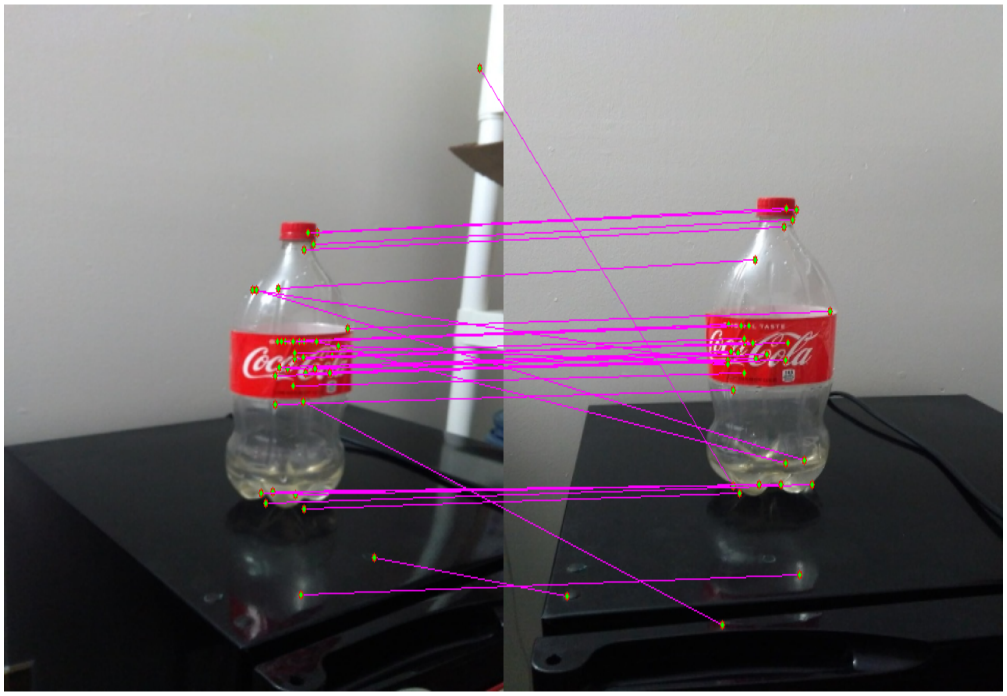

Figure 40: SIFT on image pair 4.

## 7 Comments

- The number of corners found also decreases as $\sigma$ increases due to increased smoothing.
- The overall performance of Harris corner detector using SSD and NCC stays more or less at the same level and comparable for almost all the image pairs except for the image pair 3.
- In image pair 3 the Harris Corners detected using NCC performed much better than those detected using SSD.
- It seems that the performance of the Harris corner detector is better for the image 1 pair. This may be because, the level of change in viewpoint in the image pair 2 is more than that of image pair 1.
- SIFT seems to far outperform the Harris Corner detector in all the image pairs. This is because of the DOG calculation used in SIFT. It takes care of scale invariance in a very sophisticated manner. So the keypoints detected are far more reliable and robust in this case.

```
#!/usr/bin/env python

import numpy as np, cv2, os, time, math, copy
from scipy import signal

#===============================================================================
# ARINDAM BHANJA CHOWDHURY
# abhanjac@purdue.edu
# ECE 661 FALL 2018, HW 4
#===============================================================================

#===============================================================================
# FUNCTIONS CREATED IN HW4.
#===============================================================================

def calculateHaarFilter( sigma=None ):
    '''
    Calculates the Haar Wavelet filter from given sigma.
    '''

    # Calculating the kernel size of the filter.
    ksize = math.ceil( sigma * 4 )
    ksize = int( ksize )
    ksize = ksize + 1 if ksize % 2 > 0 else ksize

    dxFilter = np.ones( (ksize, ksize) )
    dxFilter[ :, : int( ksize / 2 ) ] = -1

    dyFilter = np.ones( (ksize, ksize) )
    dyFilter[ int( ksize / 2 ) :, : ] = -1

    return dxFilter, dyFilter

#===============================================================================

# Function to normalize input image to the range of 0 to 1
# (provided all the elements dont have the same values, in which case it returns
# the original array).
normalize = lambda x: (x - np.min(x)) / (np.max(x) - np.min(x)) \
                                if np.max(x) > np.min(x) else x

#===============================================================================

def calculateHarrisCorners( img=None, sigma=None, k=None ):
    '''
    This function takes in the input image and a sigma and finds out the harris
    corners from it.
    '''
    if len(img.shape) == 3:     grayImg = cv2.cvtColor( img, cv2.COLOR_BGR2GRAY )
    else:           grayImg = img

    h, w = grayImg.shape

    # Calculating the derivative images.
    dxFilter, dyFilter = calculateHaarFilter( sigma=sigma )     # Calculate kernel.

    dxImg = signal.convolve2d( grayImg, dxFilter, mode='same' ) # Derivative image along x.
    dyImg = signal.convolve2d( grayImg, dyFilter, mode='same' ) # Derivative image along y.

    dx2Img = dxImg * dxImg
    dy2Img = dyImg * dyImg
    dxyImg = dxImg * dyImg

#-------------------------------------------------------------------------------
```

```
    # Calculating the sum of the derivatives around each pixel.
    # This is done using a convolution filter that acts like a sum.
    kint = int(5 * sigma)
    ksize = kint if kint % 2 > 0 else kint + 1  # Rounding to nearest odd integer.
    kernel = np.ones( (ksize, ksize) )

    sumx2Img = signal.convolve2d( dx2Img, kernel, mode='same' )   # Sum of dx2.
    sumy2Img = signal.convolve2d( dy2Img, kernel, mode='same' )   # Sum of dy2.
    sumxyImg = signal.convolve2d( dxyImg, kernel, mode='same' )   # Sum of dxy.

    traceImg = sumx2Img + sumy2Img      # lambda1 + lambda2 image.
    detImg = sumx2Img * sumy2Img - sumxyImg * sumxyImg      # lambda1 * lambda2 image.

#-------------------------------------------------------------------------------

    if k == None:       # Find the average k if k is not specified.
        kImg = detImg / ( traceImg * traceImg + 0.000001 )     # k = r / (1+r)^2.
        k = np.sum( kImg ) / (h * w)    # Finding average k value over the image.
        print( f'k: {k}' )

    # Response of the harris corner detector.
    R = detImg - k * traceImg * traceImg

    # Removing the negative pixels and also the ones less than k value.
    R = R * np.asarray( R > 0, dtype=np.uint8 )
    #R = np.asarray( R, dtype=np.uint8 )

#-------------------------------------------------------------------------------

    # Non-max suppression.
    nmsR = np.zeros( R.shape, dtype=np.uint8 )
    kernel = 29
    win = int(kernel / 2)

    for x in range( win, w-win, 1 ):
        for y in range( win, h-win, 1 ):
            # Finding the max value inside the kernel window.
            neighborhood = R[ y-win : y+win+1, x-win : x+win+1 ]
            maxVal = np.amax( neighborhood )

            # Only keeping the maxValue pixel and making all others 0.
            if R[ y, x ] == maxVal:
                nmsR[ y, x ] = maxVal
            else:   continue

    # Now extract these maximum points from the nmsR image.
    # These points are the non-zero points in nmsR as all the other non max points
    # are made 0.
    listOfCorners = []      # List of corner coordinates. Format (x, y), like opencv.
    for x in range( w ):
        for y in range( h ):
            if nmsR[ y, x ] > 0:    # Append all the non zero points.
                listOfCorners.append( [x,y] )

    #cv2.imshow( 'img', R )
    #cv2.waitKey(0)

    listOfCorners = sorted( listOfCorners, key=lambda x: x[1] )

    return listOfCorners

#===============================================================================

def distanceHarris( img1=None, kp1=None, img2=None, kp2=None, kernel=21, mode=None ):
    '''
    Calculate the Sum of Squared Difference (SSD) between the keypoints of image1 with
    those of image2 in the neighborhood of size kernel x kernel around the keypoint
    locations. This is when the mode is 'SSD'.
    The distance is calculated as Normalized Cross Correlation if the mode is 'NCC'.
    '''
    if mode == None:
        print( 'mode not specified. Aborting...' )
        return

    if len(img1.shape) == 3:     grayImg1 = cv2.cvtColor( img1, cv2.COLOR_BGR2GRAY )
    else:            grayImg1 = img1

    if len(img2.shape) == 3:     grayImg2 = cv2.cvtColor( img2, cv2.COLOR_BGR2GRAY )
    else:            grayImg2 = img2

    h, w = grayImg1.shape

#-------------------------------------------------------------------------------

    win = int( kernel / 2 )

    # Neighborhood of the kp1.
    x1lw, x1up = kp1[0] - win, kp1[0] + win
    if x1lw < 0:    x1lw = 0
    if x1up >= w:    x1up = w-1

    y1lw, y1up = kp1[1] - win, kp1[1] + win
    if y1lw < 0:    y1lw = 0
    if y1up >= h:    y1up = h-1

    # Neighborhood of the kp2.
    x2lw, x2up = kp2[0] - win, kp2[0] + win
    if x2lw < 0:    x2lw = 0
    if x2up >= w:    x2up = w-1

    y2lw, y2up = kp2[1] - win, kp2[1] + win
    if y2lw < 0:    y2lw = 0
    if y2up >= h:    y2up = h-1

#-------------------------------------------------------------------------------

    # It may happen that one of the keypoint is in the edge of one image and the
    # other is not. In that case the smaller of the two neighborhoods around the
    # two keypoints are considered in both the images.
    leftBound1 = kp1[0] - x1lw
    leftBound2 = kp2[0] - x2lw
    if leftBound1 < leftBound2:     x2lw = kp2[0] - leftBound1
    elif leftBound2 < leftBound1:   x1lw = kp1[0] - leftBound2

    rightBound1 = x1up - kp1[0]
    rightBound2 = x2up - kp2[0]
    if rightBound1 < rightBound2:     x2up = kp2[0] + rightBound1
    elif rightBound2 < rightBound1:   x1up = kp1[0] + rightBound2

    topBound1 = kp1[1] - y1lw
    topBound2 = kp2[1] - y2lw
    if topBound1 < topBound2:     y2lw = kp2[1] - topBound1
    elif topBound2 < topBound1:   y1lw = kp1[1] - topBound2

    botBound1 = y1up - kp1[1]
    botBound2 = y2up - kp2[1]
    if botBound1 < botBound2:     y2up = kp2[1] + botBound1
    elif botBound2 < botBound1:   y1up = kp1[1] + botBound2

    neighborhood1 = grayImg1[ y1lw : y1up, x1lw : x1up ]
```

```python
    neighborhood2 = grayImg2[ y2lw : y2up, x2lw : x2up ]

    #print( x1lw, x1up, y1lw, y1up )
    #print( x2lw, x2up, y2lw, y2up )

#-------------------------------------------------------------------------------

    if mode == 'SSD':
        diff = neighborhood1 - neighborhood2
        ssd = np.sum( diff * diff )

        return ssd

#-------------------------------------------------------------------------------

    if mode == 'NCC':
        winMean1 = np.mean( neighborhood1 )
        winMean2 = np.mean( neighborhood2 )

        num1 = neighborhood1 - winMean1
        num2 = neighborhood2 - winMean2
        num = np.sum( num1 * num2 )

        den1 = ( num1 * num1 )
        den1 = np.sum( den1 )
        den2 = ( num2 * num2 )
        den2 = np.sum( den2 )
        den = np.sqrt( den1 * den2 )

        ncc = num / ( den + 0.000001 )

        return ncc

#===============================================================================

def findGoodMatches( img1=None, listOfCorners1=None, img2=None, listOfCorners2=None, \
                     kernel=21, mode=None, matchThresh=None ):
    '''
    Calculate the matching keypoints between the two input images given the
    set of keypoints as lists. Sum of Squared Difference (SSD) between the keypoints
    is used (on a neighborhood of size kernel x kernel around the keypoint)
    when the mode is 'SSD'. Normalized Cross Correlation is used if the mode is 'NCC'.

    The matchThresh is the threshold value of the matched distance. If it is 200,
    then for SSD mode, all the matched keypoint pairs whose distance is above 200
    are ignored as bad matches, and the rest is returned.
    If the mode is NCC mode, then for a distance of 200, all matched keypoint pairs
    whose distance is below 200 are ignored as bad matches, and the rest is returned.
    '''
    if mode == None:
        print( 'mode not specified. Aborting...' )
        return

#-------------------------------------------------------------------------------

    # Taking the smaller set among the two listOfCorners. Such that the
    # no. of corners in A image is always less than that of B image.
    if len( listOfCorners1 ) < len( listOfCorners2 ):
        cornersA = listOfCorners1
        imgA = copy.deepcopy( img1 )
        cornersB = listOfCorners2
        imgB = copy.deepcopy( img2 )
        Ais1 = True     # Shows if imgA is img1 or not.
        # Used later to return matched pair of points.
    else:
        cornersA = listOfCorners2
        imgA = copy.deepcopy( img2 )
        cornersB = listOfCorners1
        imgB = copy.deepcopy( img1 )
        Ais1 = False     # Shows if imgA is img1 or not.
        # Used later to return matched pair of points.

    # Creating lists of same lenth as cornersA
    matchedCornersB = copy.deepcopy( cornersA )
    distValue = copy.deepcopy( cornersA )

#-------------------------------------------------------------------------------

    if mode == 'SSD':        # Matching by SSD.

        # Scanning all the corners of A and finding the SSD with each of B to get the
        # match (i.e. the corner in B with which it has the minimum ssd value).
        for idxa, a in enumerate(cornersA):
            minSSD = 100000
            for idxb, b in enumerate(cornersB):
                ssd = distanceHarris( imgA, a, imgB, b, kernel=21, mode='SSD' )

                if minSSD > ssd:
                    minSSD = ssd    # Calculating the minimum ssd value.
                    matchedCornersB[ idxa ] = cornersB[ idxb ]    # Storing best match.
                    distValue[ idxa ] = minSSD      # Storing the distance values.

#-------------------------------------------------------------------------------

        # The matchedPairs1To2 list stores the tuple of matched points of image 1 to 2.
        # And not the other way round. So the first element of each of the tuples is
        # a keypoint of image 1 and the 2nd element is a keypoint of image 2.
        # This is done using the flag Ais1.
        if Ais1:
            matchedPairs1To2 = [ ( cornersA[i], matchedCornersB[i] ) for i in range( \
                                                                 len( cornersA ) ) ]
        else:
            matchedPairs1To2 = [ ( matchedCornersB[i], cornersA[i] ) for i in range( \
                                                                 len( cornersA ) ) ]

        # Sorting the list with ascending order of SSD value, such that the best
        # lowest SSD value is at the beginning.
        matchedPairs1To2 = sorted( matchedPairs1To2, \
                                   key=lambda x: distValue[ matchedPairs1To2.index(x) ] )
        distValue = sorted( distValue )

        # If no matchThresh is specified then all points are returned.
        matchThresh = 100000 if matchThresh == None else matchThresh
        #print(matchThresh)

        goodMatches1to2 = [ matchedPairs1To2[i] for i in range( len(matchedPairs1To2) ) \
                                        if distValue[i] < matchThresh ]

#-------------------------------------------------------------------------------

    elif mode == 'NCC':     # Matching by NCC.

        # Scanning all the corners of A and finding the NCC with each of B to get the
        # match (i.e. the corner in B with which it has the maximum ncc value).
        for idxa, a in enumerate(cornersA):
            maxNCC = 0
            for idxb, b in enumerate(cornersB):
                ncc = distanceHarris( imgA, a, imgB, b, kernel=21, mode='NCC' )

                if maxNCC < ncc:
                    maxNCC = ncc    # Calculating the maximum ncc value.
                    matchedCornersB[ idxa ] = cornersB[ idxb ]    # Storing best match.
                    distValue[ idxa ] = maxNCC      # Storing the distance values.

#-------------------------------------------------------------------------------

        # The matchedPairs1To2 list stores the tuple of matched points of image 1 to 2.
        # And not the other way round. So the first element of each of the tuples is
        # a keypoint of image 1 and the 2nd element is a keypoint of image 2.
        # This is done using the flag Ais1.
        if Ais1:
            matchedPairs1To2 = [ ( cornersA[i], matchedCornersB[i] ) for i in range( \
                                                                   len( cornersA ) ) ]
        else:
            matchedPairs1To2 = [ ( matchedCornersB[i], cornersA[i] ) for i in range( \
                                                                   len( cornersA ) ) ]

        # Sorting the list with descending order of NCC value, such that the best
        # highest NCC value is at the beginning.
        matchedPairs1To2 = sorted( matchedPairs1To2, \
                                   key=lambda x: distValue[ matchedPairs1To2.index(x) ],
                                   reverse=True )
        distValue = sorted( distValue, reverse=True )

        # If no matchThresh is specified then all points are returned.
        matchThresh = 0 if matchThresh == None else matchThresh

        goodMatches1to2 = [ matchedPairs1To2[i] for i in range( len(matchedPairs1To2) ) \
                                        if distValue[i] > matchThresh ]

#-------------------------------------------------------------------------------

    return matchedPairs1To2, goodMatches1to2, distValue

#===============================================================================

def distanceSift( kp1=None, des1=None, kp2=None, des2=None, matchThresh=None ):
    '''
    This calculates the euclidean distance between two keypoint descriptors for sift.
    keypoints are lists of keypoint object in opencv and descriptors are numpy arrays
    with 128 columns (for sift), where each of the rows represent the 128 element
    vector for the corresponding keypoint.
    '''

    # Taking the smaller set among the two listOfCorners. Such that the
    # no. of corners in A image is always less than that of B image.
    if len( kp1 ) < len( kp2 ):
        kpA, desA, kpB, desB = kp1, des1, kp2, des2
        Ais1 = True     # Shows if desA is des1 or not.
        # Used later to return matched pair of points.
    else:
        kpA, desA, kpB, desB = kp2, des2, kp1, des1
        Ais1 = False     # Shows if desA is des1 or not.
        # Used later to return matched pair of points.

    # Creating lists of same lenth as kpA
    matchedKpB = np.ones( len(kpA) ).tolist()
    distValue = np.ones( len(kpA) ).tolist()

#-------------------------------------------------------------------------------

    # Scanning all the descriptors of A and finding the euclidean distance with
    # each of B to get the match (i.e. the descriptor in B with which it has the
    # minimum ssd value).
    for idxa, a in enumerate( desA ):
        minDist = 100000
        for idxb, b in enumerate( desB ):
            dist = (a - b) * (a - b)
            dist = np.sqrt( np.sum( dist ) )

            if minDist > dist:
                minDist = dist    # Calculating the minimum euclidean distance value.
                pt = [ int(kpB[ idxb ].pt[0]), int(kpB[ idxb ].pt[1]) ]
                matchedKpB[ idxa ] = pt    # Storing best match.
                distValue[ idxa ] = minDist      # Storing the distance values.

#-------------------------------------------------------------------------------

    # The matchedPairs1To2 list stores the tuple of matched points of image 1 to 2.
    # And not the other way round. So the first element of each of the tuples is
    # a keypoint of image 1 and the 2nd element is a keypoint of image 2.
    # This is done using the flag Ais1.
    if Ais1:
        matchedPairs1To2 = [ ( [ int(kpA[i].pt[0]), int(kpA[i].pt[1]) ], matchedKpB[i] ) \
                                for i in range( len( kpA ) ) ]
    else:
        matchedPairs1To2 = [ ( matchedKpB[i], [ int(kpA[i].pt[0]), int(kpA[i].pt[1]) ] ) \
                                for i in range( len( kpA ) ) ]

    # Sorting the list with ascending order of distance value, such that the best
    # lowest distance value is at the beginning.
    matchedPairs1To2 = sorted( matchedPairs1To2, \
                               key=lambda x: distValue[ matchedPairs1To2.index(x) ] )
    distValue = sorted( distValue )

    # If no matchThresh is specified then all points are returned.
    matchThresh = 100000 if matchThresh == None else matchThresh
    #print(matchThresh)

    goodMatches1to2 = [ matchedPairs1To2[i] for i in range( len(matchedPairs1To2) ) \
                                    if distValue[i] < matchThresh ]

#-------------------------------------------------------------------------------

    return matchedPairs1To2, goodMatches1to2, distValue

#===============================================================================

if __name__ == '__main__':

    # TASK 2.1.1

    # Finding matching harris corners in image1 and image2.

    filepath = './PicsSelf'
    subfolder1 = 'pair2'
    filename1, filename2 = '1.jpg', '2.jpg'

    img1 = cv2.imread( os.path.join( filepath, subfolder1, filename1 ) )
    img2 = cv2.imread( os.path.join( filepath, subfolder1, filename2 ) )

#-------------------------------------------------------------------------------

    # Reshaping the images as some of them are too big for display.

    imgW, imgH = 640, 480

    h, w, c = img1.shape
    if h > imgH or w > imgW:    inp = cv2.INTER_AREA
    else:       inp=cv2.INTER_LINEAR
    img1 = cv2.resize( img1, (imgW, imgH), interpolation=inp )

    h, w, c = img2.shape
    if h > imgH or w > imgW:    inp = cv2.INTER_AREA
    else:       inp=cv2.INTER_LINEAR
    img2 = cv2.resize( img2, (imgW, imgH), interpolation=inp )

    #cv2.imshow( 'Image', img1 )
    #cv2.waitKey(0)

#-------------------------------------------------------------------------------

    sigma = 0.707     # Scale.
    listOfCorners1 = calculateHarrisCorners( img1, sigma=sigma )
    print( f'Number of corners found in image1: {len(listOfCorners1)}' )
    listOfCorners2 = calculateHarrisCorners( img2, sigma=sigma )
    print( f'Number of corners found in image2: {len(listOfCorners2)}' )

    mode = 'SSD'
    matchThresh = 1200
    matchedPairs1To2, goodMatches1to2, distValue = findGoodMatches( img1, listOfCorners1, \
                                                     img2, listOfCorners2, \
                                                     kernel=21, mode=mode, \
                                                     matchThresh=matchThresh )

    print( f'Number of good matches for {mode}: {len(goodMatches1to2)} out of ' \
           f'{len(matchedPairs1To2)} total matches (with threshold: {matchThresh}).' )

    #print(distValue)

    img = np.hstack( ( img1, img2 ) )
    for idx, i in enumerate( goodMatches1to2 ):
        pt1 = i[0]
        pt2 = [ i[1][0] + imgW, i[1][1] ]
        cv2.line( img, tuple(pt1), tuple(pt2), (0,255,255), 1 )

        cv2.circle( img, tuple(pt1), 2, (0,255,0), -1 )
        cv2.circle( img, tuple(pt1), 3, (0,0,255), 1 )

        cv2.circle( img, tuple(pt2), 2, (0,255,0), -1 )
        cv2.circle( img, tuple(pt2), 3, (0,0,255), 1 )

    #cv2.imshow( 'Matches', img )
    #cv2.waitKey(0)
    cv2.imwrite( './img4_ssd_sigma1.png', img )

#===============================================================================

    # TASK 2.2

    # Finding matching sift keypoints in image1 and image2.

    img1gray = cv2.cvtColor( img1, cv2.COLOR_BGR2GRAY )
    img2gray = cv2.cvtColor( img2, cv2.COLOR_BGR2GRAY )

    sift = cv2.xfeatures2d.SIFT_create()       # Initiate sift detector.
    kp1, des1 = sift.detectAndCompute( img1gray, None )
    kp2, des2 = sift.detectAndCompute( img2gray, None )

    matchThresh = 230

    matchedPairs1To2, goodMatches1to2, distValue = distanceSift( kp1, des1, kp2, des2, \
                                                           matchThresh=matchThresh)

    print( f'Number of good matches for Sift: {len(goodMatches1to2)} out of ' \
        f'{len(matchedPairs1To2)} total matches (with threshold: {matchThresh}).' )

    #print(distValue)

    img = np.hstack( ( img1, img2 ) )
    for idx, i in enumerate( goodMatches1to2 ):
        pt1 = i[0]
        pt2 = [ i[1][0] + imgW, i[1][1] ]
        cv2.line( img, tuple(pt1), tuple(pt2), (255,0,255), 1 )

        cv2.circle( img, tuple(pt1), 2, (0,255,0), -1 )
        cv2.circle( img, tuple(pt1), 3, (0,0,255), 1 )

        cv2.circle( img, tuple(pt2), 2, (0,255,0), -1 )
        cv2.circle( img, tuple(pt2), 3, (0,0,255), 1 )

    cv2.imshow( 'Matches', img )
    cv2.waitKey(0)
```