# Homework 4

Alexandre Olivé Pellicer
aolivepe@purdue.edu

## 1 Theory Question

**What is the theoretical reason for why the LoG of an image can be computed as a DoG?**

For a given image $f(x, y)$, the LoG operator first smoothies the image with a Gaussian function

$$ff(x, y, \sigma) = f(x, y) * g(x, y) = \iint_{-\infty}^{\infty} f(x', y')g(x - x', y - y')dx'dy' \tag{1}$$

where $ff(x, y, \sigma)$ represent the $\sigma$-smoothed version of $f(x, y)$ and $g(x, y)$ is the Gaussian function:

$$g(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2}$$

Then, the Laplacian

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{3}$$

is applied to the smoothed image:

$$\nabla^2 ff(x, y, \sigma) = f(x, y) * h(x, y, \sigma) \tag{4}$$

where

$$h(x, y, \sigma) = -\frac{1}{2\pi\sigma^4}(2 - \frac{x^2 + y^2}{\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}) \tag{5}$$

The DoG refers to the difference of two Gaussian-smoothed versions of f(x, y) for two different values of $\sigma$. We now compute the DoG of an image and show that it equals to LoG.

$$\begin{aligned} \tfrac{\partial}{\partial\sigma} ff(x, y, \sigma) &= \iint f(x', y') \tfrac{\partial}{\partial\sigma} \{\tfrac{1}{2\pi\sigma^2} e^{-\frac{(x-x')^2+(y-y')^2}{2\sigma^2}}\} dx'dy' \\ &= \iint f(x', y') [\tfrac{-1}{\pi\sigma^3} + \tfrac{1}{2\pi\sigma^2}(-[(x - x')^2 + (y - y')^2]) \tfrac{-2}{2\sigma^3}] e^{-\frac{(x-x')^2+(y-y')^2}{2\sigma^2}} dx'dy' \\ &= -\tfrac{\sigma}{2\pi\sigma^4} \iint f(x', y') [2 - \tfrac{(x-x')^2+(y-y')^2}{\sigma^2}] e^{-\frac{(x-x')^2+(y-y')^2}{2\sigma^2}} dx'dy' \\ &= \sigma f(x, y) * h(x, y) \end{aligned} \tag{6}$$

Therefore:

$$\frac{\partial}{\partial\sigma} ff(x, y, \sigma) = \sigma\nabla^2 ff(x, y, \sigma) \tag{7}$$

**Explain in your own words why computing the LoG of an image as a DoG is computationally much more efficient for the same value of $\sigma$**

It is much faster to calculate the Difference of Gaussian (DoG) with $\sigma_1$ and $\sigma_2$ for the Gaussian smoothing than to calculate LoG for any given $\sigma$. This is because the Gaussian function

$$g(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{8}$$

is separable in $x$ and $y$. Therefore, each of the 2-D smoothings for the DoG can be carried out by two applications of 1-D smoothings, first along $x$ and then the result along $y$. The separability enables the DoG to be computed more efficiently by performing the 1-D smoothings twice, first along the $x$ direction and then along the $y$ direction. This cannot be done for the LoG operator h(x, y) since it is not separable and therefore its computation is more computationally expensive.

# 2 Description of Implementations

## 2.1 Harris Corner Detector

A corner is defined by any pixel in the vicinity of which the gray levels show significant variations in at least two different directions. Therefore, we first convert the images to gray level and normalize them.

In a $5\sigma \times 5\sigma$ neighborhood of the pixel where we wish to determine the presence or absence of a corner we construct the following matrix:

$$C = \begin{bmatrix} \sum d_x^2 & \sum d_x d_y \\ \sum d_x d_y & \sum d_y^2 \end{bmatrix} \tag{9}$$

where the summations are over all the pixels in the $5\sigma \times 5\sigma$ neighborhood.

The first-order derivatives ($d_x$ and $d_y$) of the image shown in Equation 9 are computed using the Haar wavelet filter over the gray scale image. As explained in *Lecture 9* notes, the basic form of the Haar wavelet is $\begin{bmatrix} -1 & 1 \end{bmatrix}$ along $x$ and $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$ along $y$. These forms are scaled up to an $M \times M$ operator where $M$ is the smallest integer greater than $4\sigma$. For example, Equation 10 is the $4 \times 4$ operator along $x$ and Equation 11 is the $4 \times 4$ operator along $y$.

$$\begin{bmatrix} -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{bmatrix} \tag{11}$$

$\sigma$ is the scale at which derivatives are taken. Therefore, $\sigma$ determines the size of the neighborhood used to compute the first-order derivatives. A large $\sigma$ means we are analyzing a broader region, which may result in missing significant changes. Conversely, a smaller $\sigma$ means we are analyzing a narrower region and could result in detecting too many corners. A pixel is considered a corner if both $d_x$ and $d_y$ are high.

Let $\lambda_1$ and $\lambda_2$ ($\lambda_1 \geq \lambda_2$) be the two originally eigenvalues of $C$. The pixel at the center of the window will be called corner if the ratio $r = \frac{\lambda_2}{\lambda_1}$ is higher than a specific threshold.

As seen in the theory lectures, the ratio $r$ can be computed by the following operations:

$$Tr(C) = \sum d_x^2 + \sum d_y^2 = \lambda_1 + \lambda_2 \tag{12}$$

$$det(C) = \sum d_x^2 \sum d_y^2 - (\sum d_x d_y)^2 = \lambda_1 \lambda_2 \tag{13}$$

This implies:

$$\frac{det(C)}{[Tr(C)]^2} = \frac{\lambda_1 \lambda_2}{(\lambda_1 + \lambda_2)^2} = \frac{r}{(1+r)^2} \tag{14}$$

Therefore, the ratio $\frac{det(C)}{[Tr(C)]^2}$ can be directly thresholded for corner detection.

Nevertheless, as seen in previous year's solutions, instead of computing the ratio $r$, we calculate the Harris response:

$$H = det(C) - k[Tr(C)]^2 \tag{15}$$

where we empirically set $k$=0.05.

To prevent identifying too many points near a corner as corners, we select only the point with the highest Harris response within a window of size $k \times k$ where $k$ is the double of the window size used to compute $C$. The best results are achieved when plotting the first 100 detected corners.

We have used 4 different values of sigma: 0.8, 1.2, 1.6 and 2.

## 2.2 Metrics to Establish Correspondences

The goal is matching the detected corners of the two images. In order to do it, we compare the gray levels in a $(b+1) \times (b+1)$ window around each of the detected corners of one image with the gray levels in a $(b+1) \times (b+1)$ window around each of the detected corners of the other image. The pair of corners with smallest distance are matched. Two different distances/metrics are tested in this homework to establish the correspondences: Sum of Squared Distances (SSD) and Normalized Cross Correlation (NCC).

### 2.2.1 Sum of Squared Distances (SSD)

$$SSD = \sum_i \sum_j |f_1(i,j) - f_2(i,j)|^2 \tag{16}$$

### 2.2.2 Normalized Cross Correlation (NCC)

$$NCC = \frac{\sum\sum(f_1(i,j) - m_1)(f_2(i,j) - m_2)}{\sqrt{(\sum\sum(f_1(i,j) - m_1)^2)\,(\sum\sum(f_2(i,j) - m_2)^2)}} \tag{17}$$

where

- f1 is image 1
- f2 is image 2
- m1 mean value of the gray levels in the window around the interest point in image 1
- m2 mean value of the gray levels in the window around the interest point in image 2

The window used for both cases SSD nad NCC is of size 10. For the NCC the closer to 1, the better. We set a threshold to 0.3 to eliminate worse matchings.

## 2.3 SIFT Overview

We use the Scale Invariant Feature Transform (SIFT) for extracting interest points. We do a quick overview of the SIFT algorithm following the steps described in *Lecture 9* notes.

- First, the DoG pyramid is constructed. An image of size $N \times N$ is smoothed at scale $\sigma_0$. The smoothed image is smoothed again using a scale of $\sigma_0 + \Delta\sigma_0$ where $\Delta\sigma_0 = \frac{\sigma_0}{2^i}$. This is done at least 3 times $i = 1, 2, 3, ...$ to obtain 4 different smoothed images. The difference between the adjacent smoothed images is computed. In case we repeated the process 3 times, 3 images will result from computing the differences. Next, $\sigma_1 = 2\sigma_0$ is used and the original image of size $N \times N$ is downsampled by 2. The same process described using $\sigma_0$ and the $N \times N$ image is done with $\sigma_1$ and the $N/2 \times N/2$ image and so on. Every scale $\sigma_i$ correspond to an octave.
- Find all the local extrema (maximums and minimums) in the DoG pyramid. This is done comparing every pixel in each ocatave with the 8 in the immediate $3 \times 3$ neighborhood at the same scale, the 9 points in the $3 \times 3$ neighborhood in the DoG that is at the next level up in the scale space and the 9 points in the $3 \times 3$ neighborhood in the DoG that is in the level just below.
- The location of the extremum in the DoG pyramid can be improved by estimating the second-order derivatives of $D$ at the sampling points in the DoG pyramid. By doing this, we localize the extremum with "sub-pixel" accuracy in the vicinity of where the extremum was found. After some calculations we will know that the extremum is given by:

  $$\overrightarrow{x} = -H^{-1}(\overrightarrow{x_0})J(\overrightarrow{x_0}) \tag{18}$$

  where $H$ is the Hessian and $J(\overrightarrow{x_0})$ is the gradient vector estimated at $\overrightarrow{x_0}$.
- We weed out the weak extrema by thresholding $|D(\overrightarrow{x})|$ at the locations $(x, y, \sigma)^T = \overrightarrow{x}$ of the extremums. Typically and extremum is rejected if $|D(x)| < 0.03$. We also reject those extrema in the scale space that draw their support from an edge in the image.

- We associate a "dominant local orientation" with each extremum that survives the previous step. To find the local orientation we calculate the gradient vector of the Gaussian-smoothed image $ff(x, y, \sigma)$ at the scale $\sigma$ of the extremum. At each extremum, in a $k \times k$ neighborhood we compute the gradient magnitude $m(x, y)$ and the gradient orientation $\vartheta(x, y)$. $\vartheta(x, y)$ is weighted with $m(x, y)$. We construct a histogram of $\vartheta(x, y)$ using 36 bins spanning the full $360^{\text{o}}$ range. The bin where the histogram peak occurs gives us the "dominant local orientation".

- Create the SIFT descriptor (128 dimensional vector) for each retained extremum in the DoG pyramid. At the scale of each extremum, the extremum point is surrounded by a $16 \times 16$ neighborhood of points that is divided into $4 \times 4$ cells, each consisting of a $4 \times 4$ block of points. The magnitude of the gradients in the $16 \times 16$ neighborhood are weighted by a Gaussian. For each 16 cells, an 8-bin orientation histogram is calculated from the weighted values of $\vartheta(x, y)$ at the 16 pixels in the cell. The 16 8-bin histograms are stringed together resulting in the 128 dimensional vector for each extremum. The length of the vector is normalized to unity to make it invariant to changes in illumination These SIFT descriptors are used ti establish the Correspondences between the points of the two different images.

Figure 1: Harris Corner detector performance for $\sigma$=0.8 in the hovde image

Figure 2: Correspondence between points using SSD for $\sigma$=0.8 in the hovde image

# 3 Obtained Results Using the Harris Corner Detector and SSD and NCC

Figures 1–48 show the obtained results when using the SIFT

**Observations:** Looking at the results we observe the following behaviors:

- As $\sigma$ increases, the Harris Corner Detector detects less corners. This behavior is aligned with the expected results since when we use a higher $\sigma$, we are looking for corners in a smoother image which results in less edges and therefore less corners are detected. We could also consider that the larger the $\sigma$ the smaller the number of false matches since most of the mismatches that occur when using a small value of $\sigma$ disappear when using a larger value.

- In the case of the TV image we see that the overall performance decreases. The main reason can be the reflection of the light in the TV screen. We see that some corners are detected from the reflected objects in the TV screen and do not have their correspondent corner in the second image because the reflection is different since the angle at which the image has been taken is different. This leads to more false matches. The extreme and particular case is when using $\sigma = 2$ and NCC where just one corner gets matched.

- It is difficult to get a complete conclusion of the obtained results without a quantitative metric. Doing a qualitative evaluation, comparing the SSD and the NCC method, we can say that the SDD method is more robust when using a small value of $\sigma$ or when there are image distortions. This behavior matches with the theory since SSD directly measures the pixel differences while the NCC also uses the relative distributions of the pixels surrounding the corners.

Figure 3: Correspondence between points using NCC for $\sigma$=0.8 in the hovde image

Figure 4: Harris Corner detector performance for $\sigma$=1.2 in the hovde image

Figure 5: Correspondence between points using SSD for $\sigma$=1.2 in the hovde image

Figure 6: Correspondence between points using NCC for $\sigma$=1.2 in the hovde image

Figure 7: Harris Corner detector performance for $\sigma$=1.6 in the hovde image

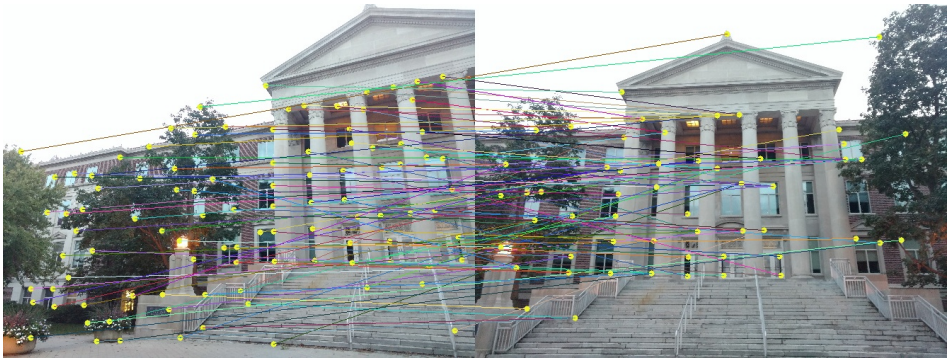


Figure 8: Correspondence between points using SSD for $\sigma$=1.6 in the hovde image

Figure 9: Correspondence between points using NCC for $\sigma$=1.6 in the hovde image

Figure 10: Harris Corner detector performance for $\sigma$=2.0 in the hovde image

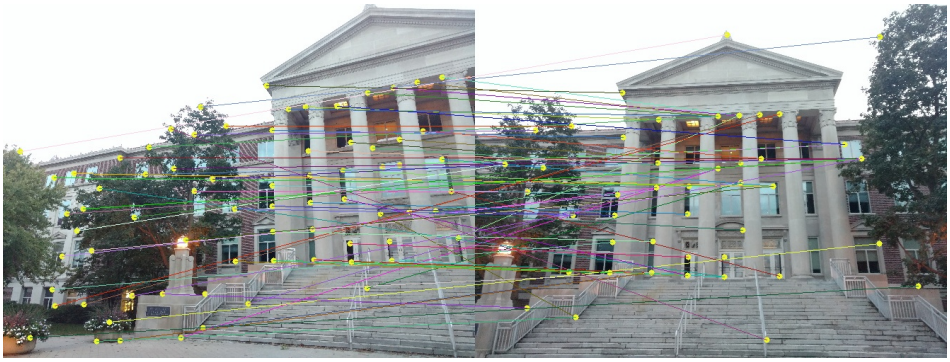


Figure 11: Correspondence between points using SSD for $\sigma$=2.0 in the hovde image

Figure 12: Correspondence between points using NCC for $\sigma$=2.0 in the hovde image

Figure 13: Harris Corner detector performance for $\sigma$=0.8 in the temple image

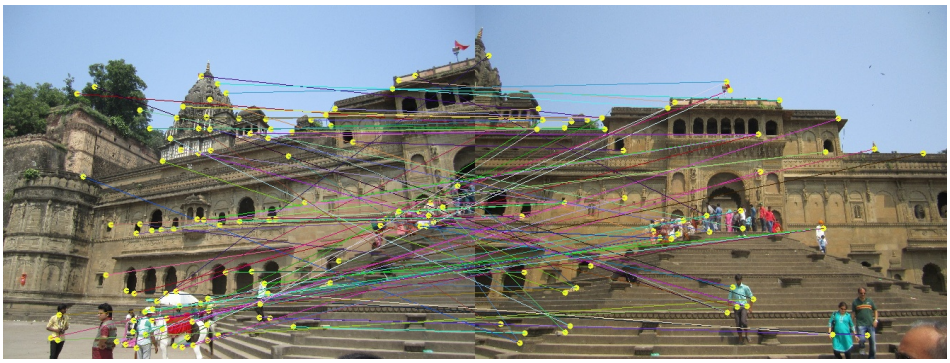


Figure 14: Correspondence between points using SSD for $\sigma$=0.8 in the temple image

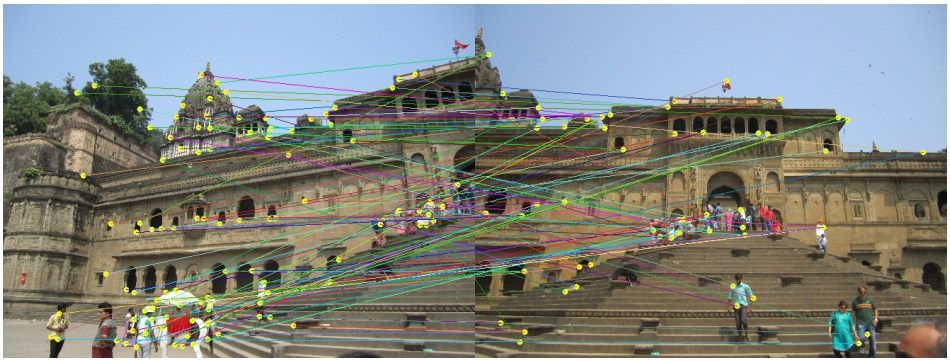


Figure 15: Correspondence between points using NCC for $\sigma$=0.8 in the temple image

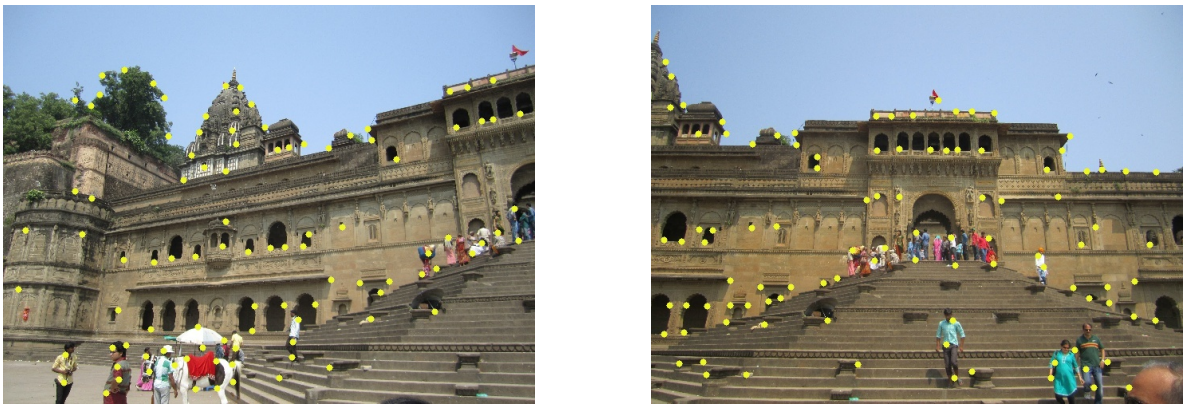


Figure 16: Harris Corner detector performance for $\sigma$=1.2 in the temple image

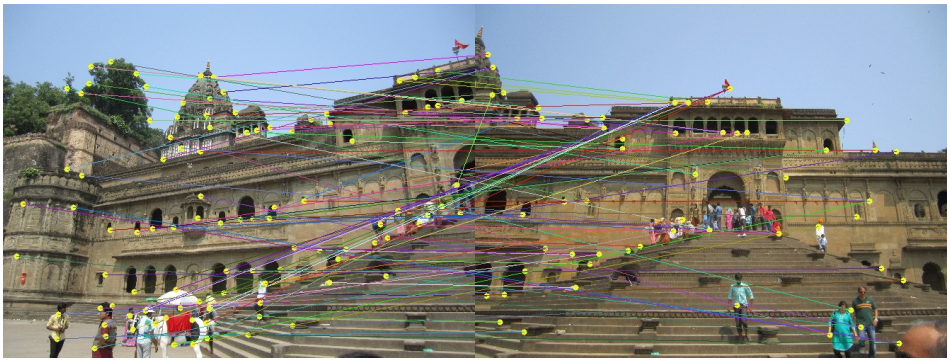


Figure 17: Correspondence between points using SSD for $\sigma$=1.2 in the temple image

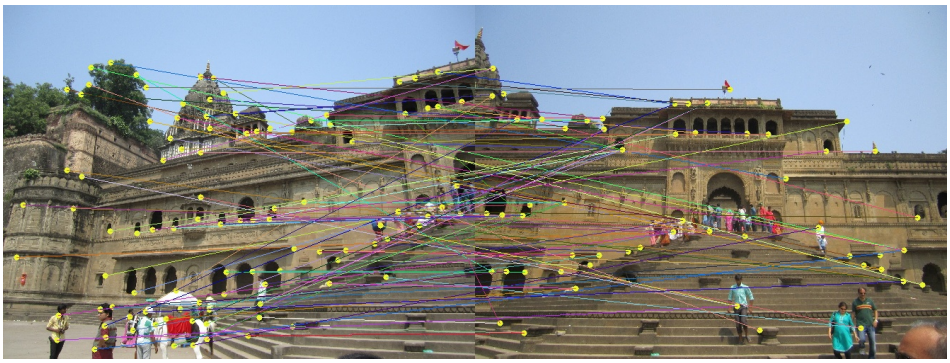


Figure 18: Correspondence between points using NCC for $\sigma$=1.2 in the temple image

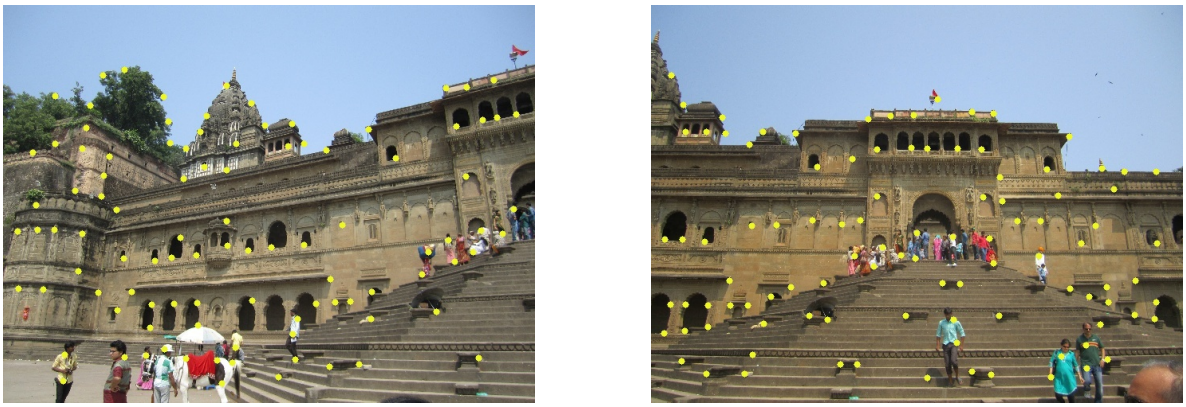


Figure 19: Harris Corner detector performance for $\sigma$=1.6 in the temple image

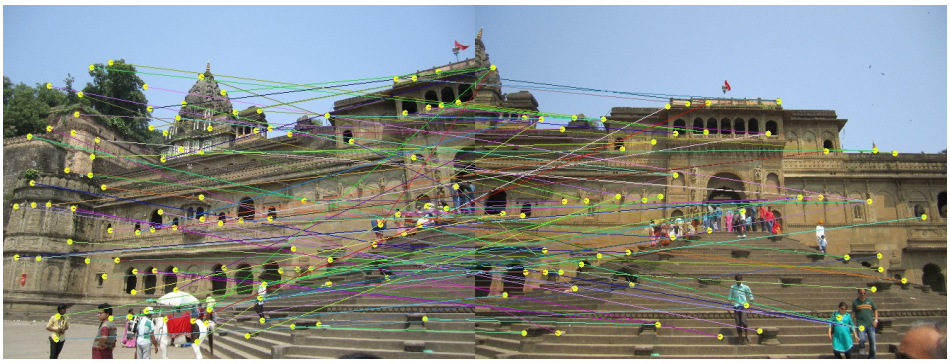


Figure 20: Correspondence between points using SSD for $\sigma$=1.6 in the temple image

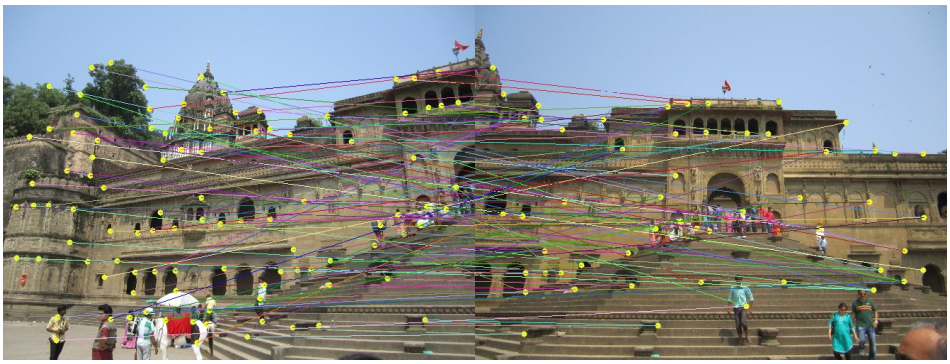


Figure 21: Correspondence between points using NCC for $\sigma$=1.6 in the temple image

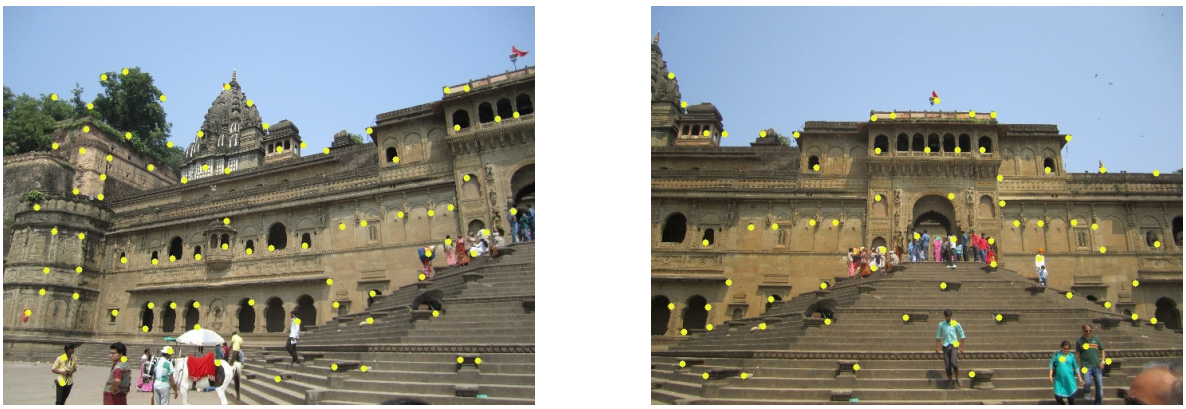


Figure 22: Harris Corner detector performance for $\sigma$=2.0 in the temple image

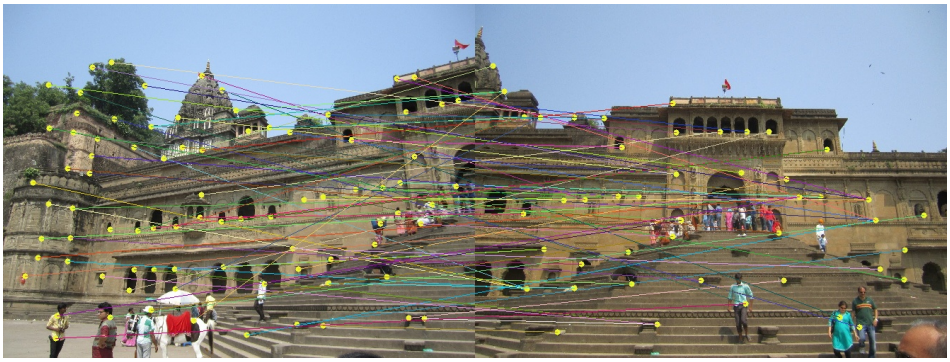


Figure 23: Correspondence between points using SSD for $\sigma$=2.0 in the temple image

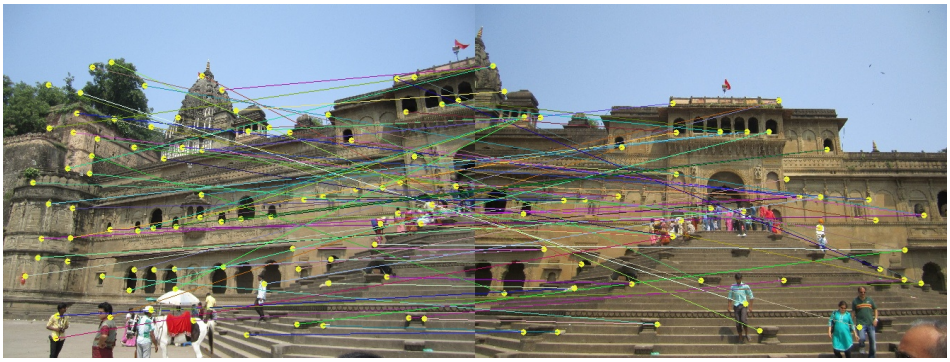


Figure 24: Correspondence between points using NCC for $\sigma$=2.0 in the temple image

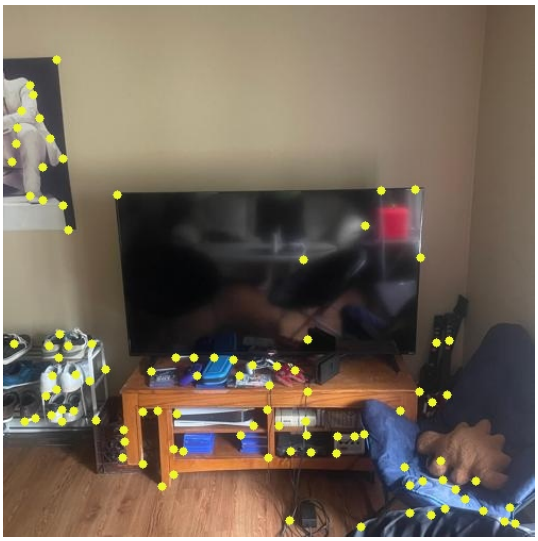


Figure 25: Harris Corner detector performance for $\sigma$=0.8 in the tv image

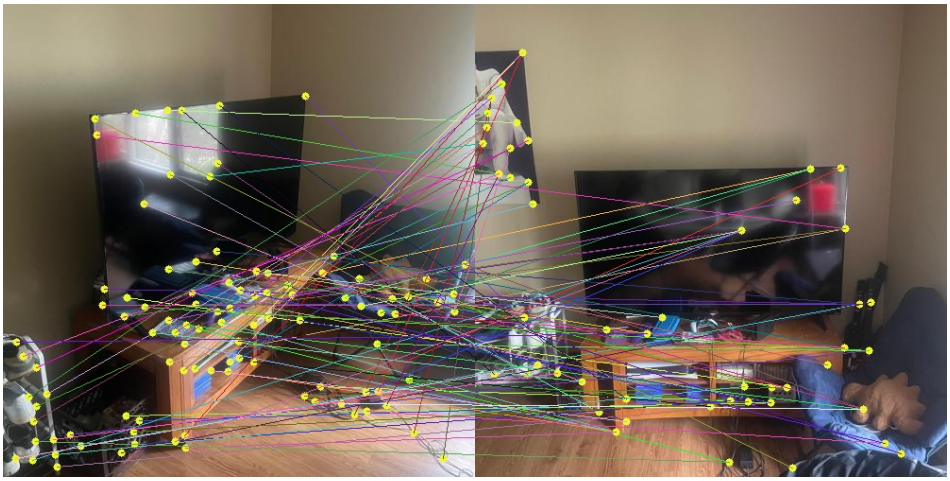


Figure 26: Correspondence between points using SSD for $\sigma$=0.8 in the tv image

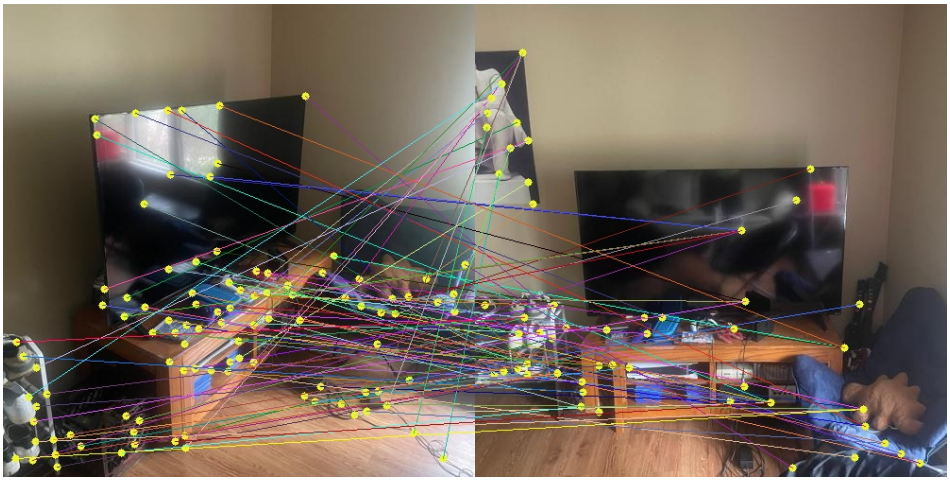
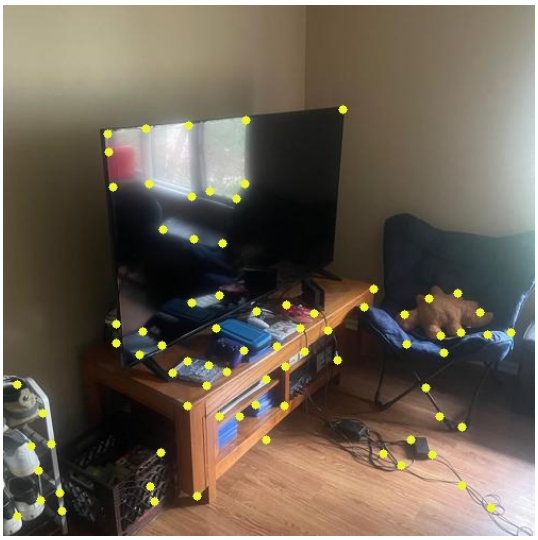


Figure 27: Correspondence between points using NCC for $\sigma$=0.8 in the tv image



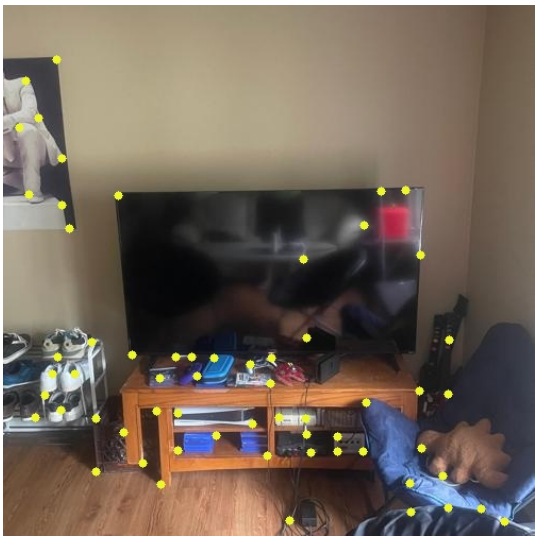


Figure 28: Harris Corner detector performance for $\sigma$=1.2 in the tv image

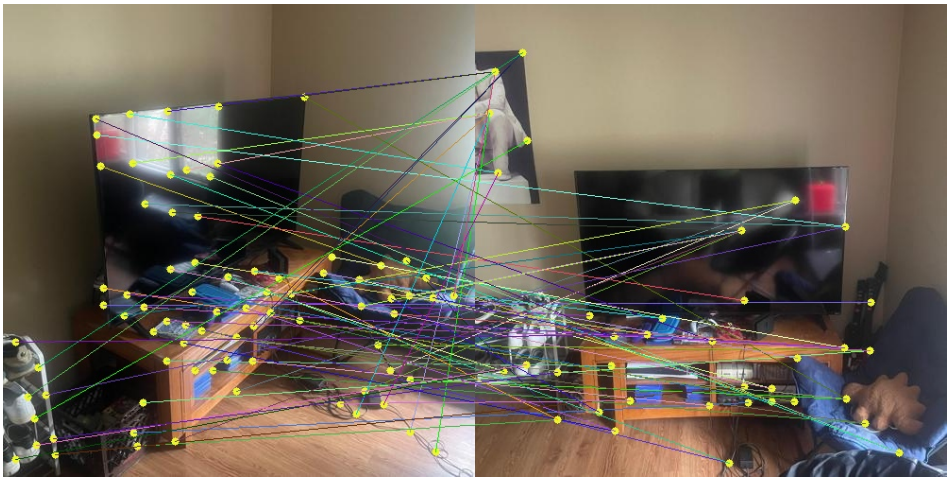


Figure 29: Correspondence between points using SSD for $\sigma$=1.2 in the tv image

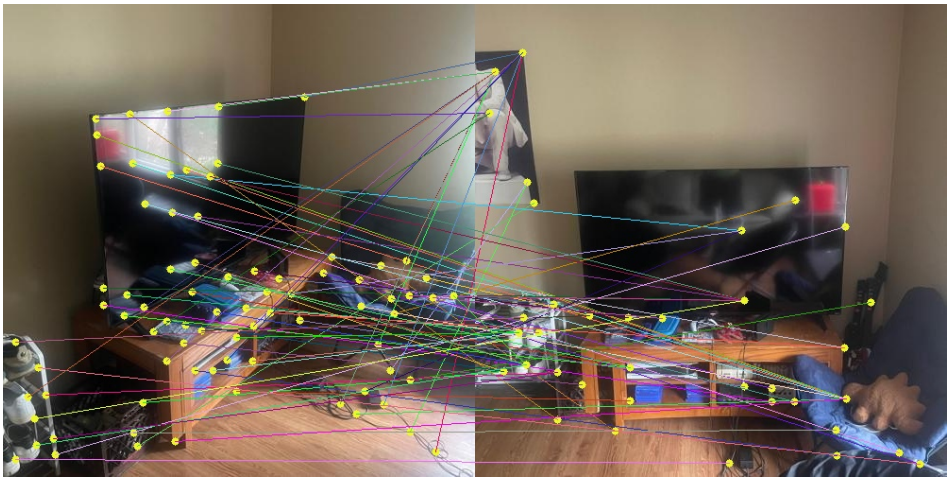


Figure 30: Correspondence between points using NCC for $\sigma$=1.2 in the tv image

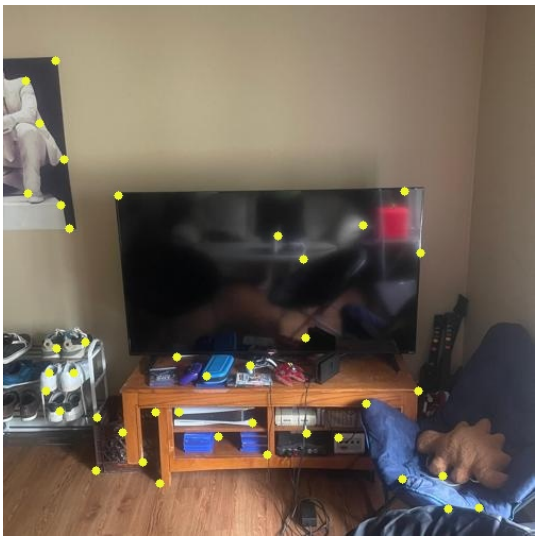


Figure 31: Harris Corner detector performance for $\sigma$=1.6 in the tv image

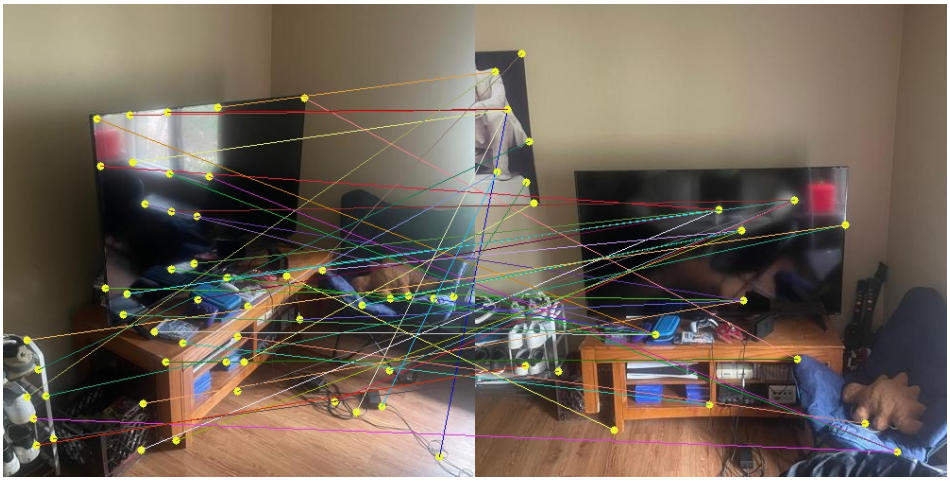


Figure 32: Correspondence between points using SSD for $\sigma$=1.6 in the tv image

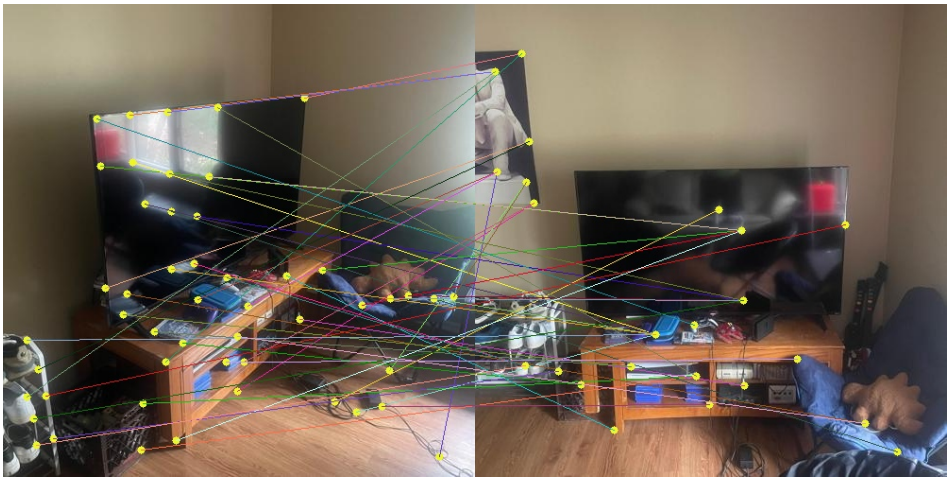
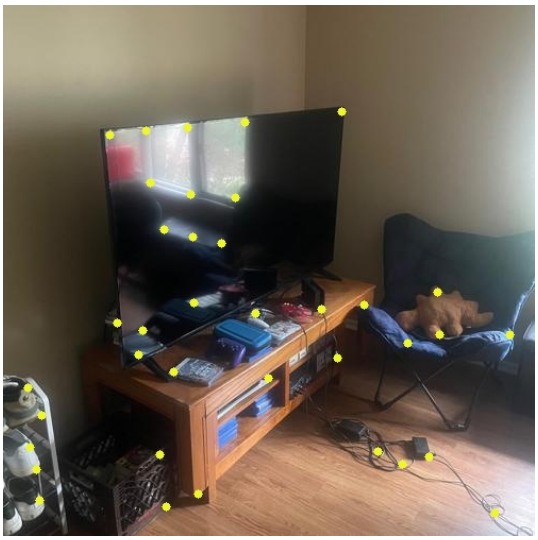


Figure 33: Correspondence between points using NCC for $\sigma$=1.6 in the tv image



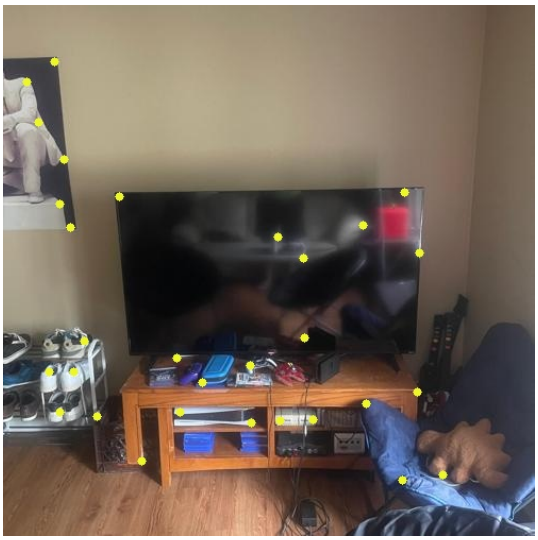


Figure 34: Harris Corner detector performance for $\sigma$=2.0 in the tv image

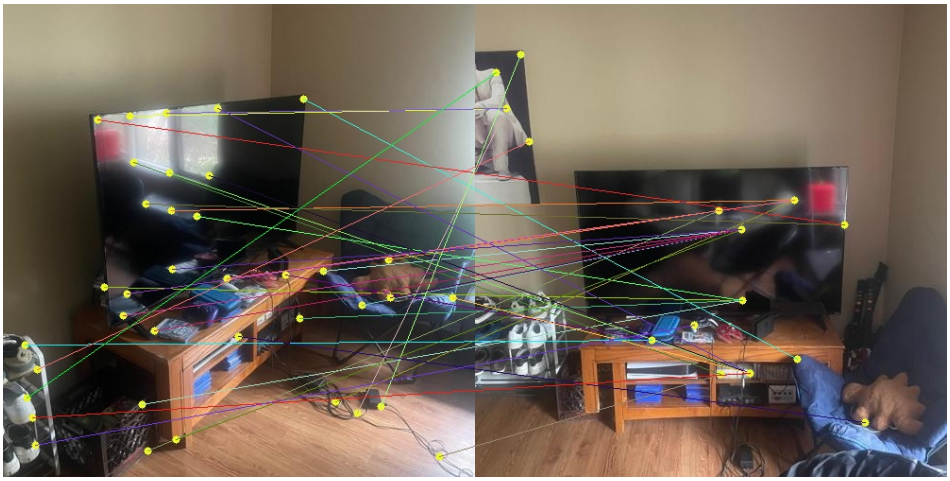


Figure 35: Correspondence between points using SSD for $\sigma$=2.0 in the tv image

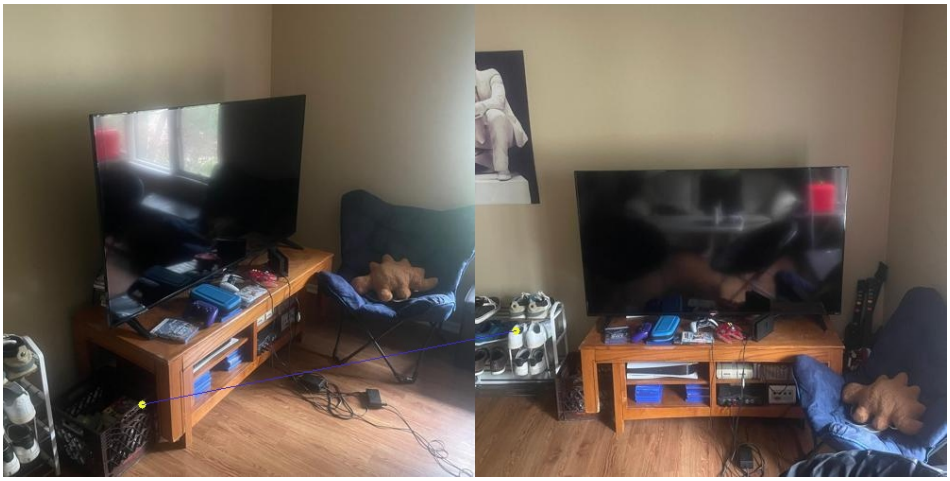


Figure 36: Correspondence between points using NCC for $\sigma$=2.0 in the tv image


Figure 37: Harris Corner detector performance for $\sigma$=0.8 in the shelf image

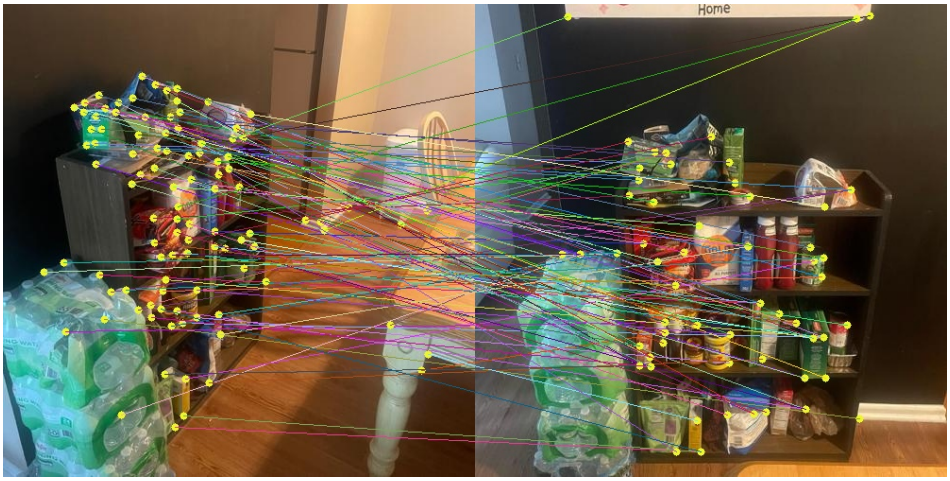



Figure 38: Correspondence between points using SSD for $\sigma$=0.8 in the shelf image

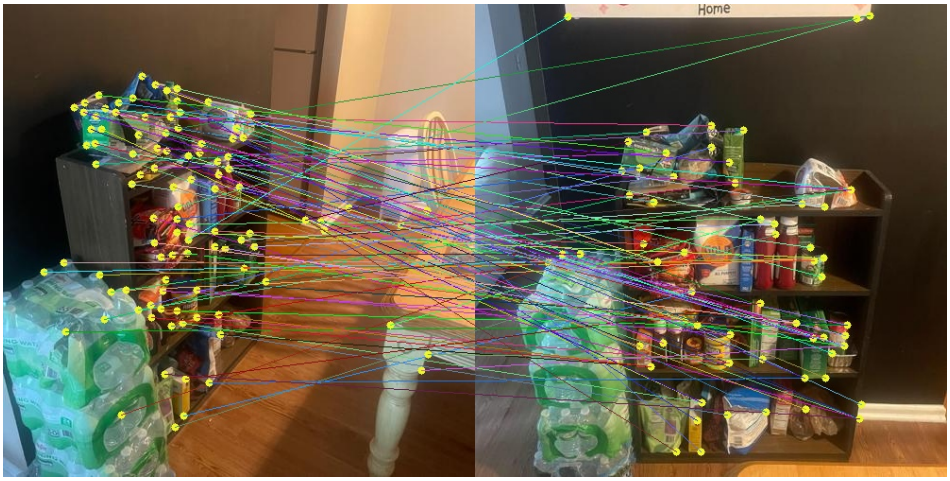
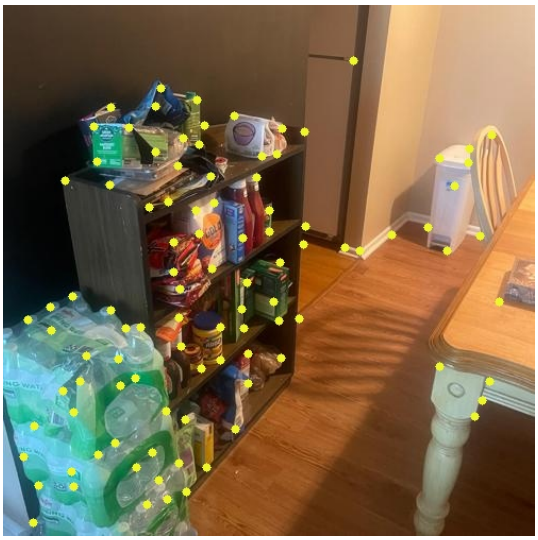



Figure 39: Correspondence between points using NCC for $\sigma$=0.8 in the shelf image



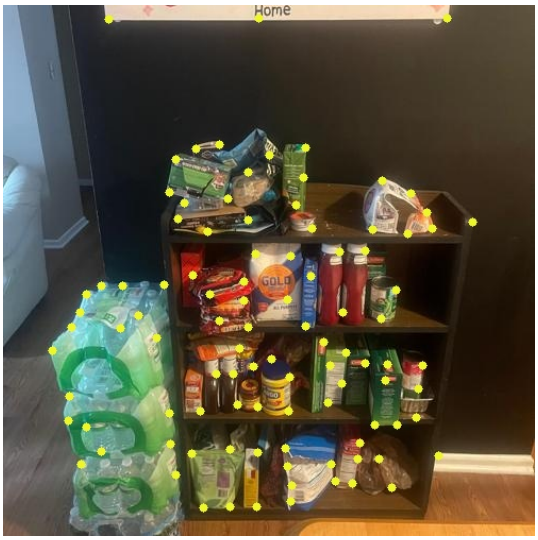



Figure 40: Harris Corner detector performance for $\sigma$=1.2 in the shelf image

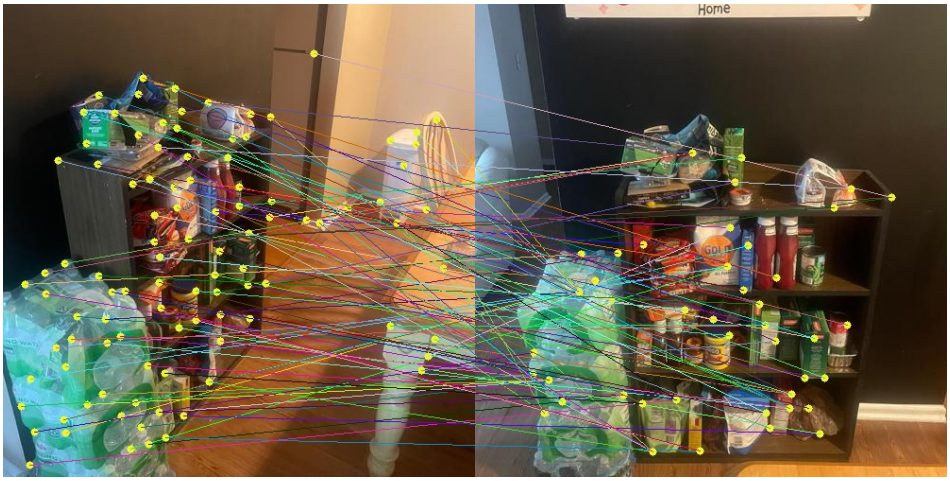



Figure 41: Correspondence between points using SSD for $\sigma$=1.2 in the shelf image

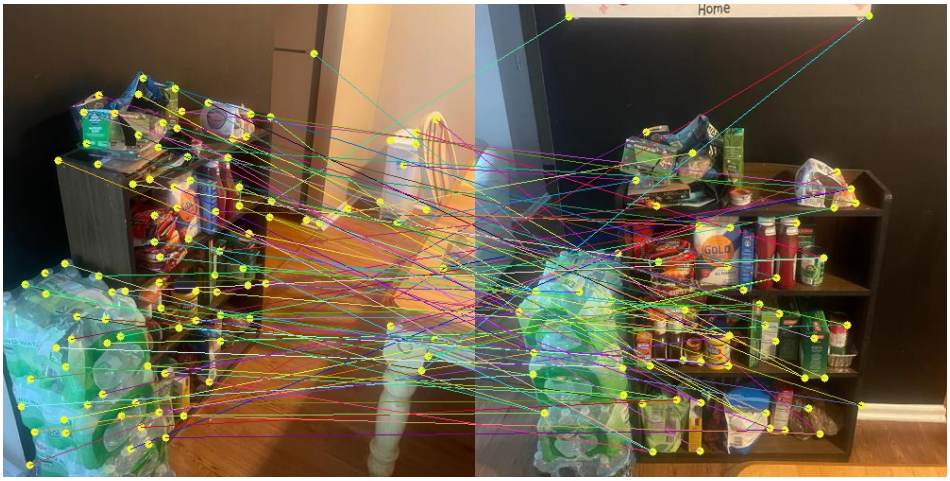



Figure 42: Correspondence between points using NCC for $\sigma$=1.2 in the shelf image


Figure 43: Harris Corner detector performance for $\sigma$=1.6 in the shelf image

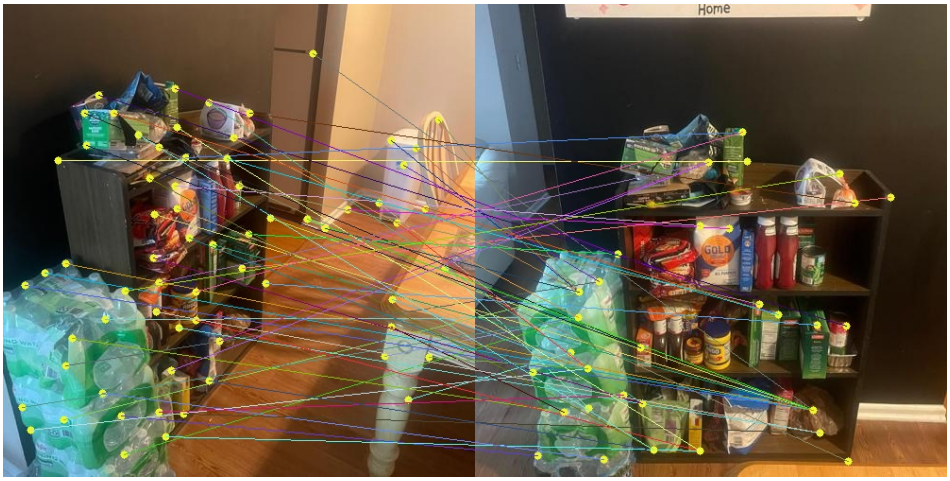



Figure 44: Correspondence between points using SSD for $\sigma$=1.6 in the shelf image


Figure 45: Correspondence between points using NCC for $\sigma$=1.6 in the shelf image


Figure 46: Harris Corner detector performance for $\sigma$=2.0 in the shelf image


Figure 47: Correspondence between points using SSD for $\sigma$=2.0 in the shelf image

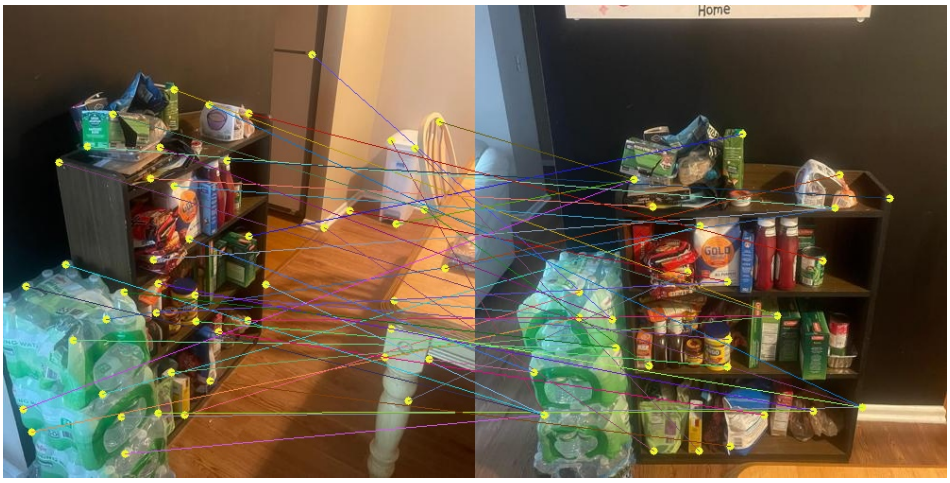



Figure 48: Correspondence between points using NCC for $\sigma$=2.0 in the shelf image

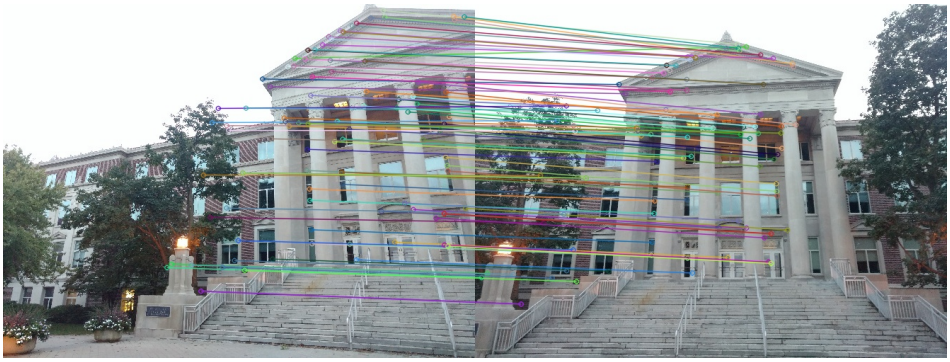


Figure 49: Correspondence between points using SIFT in the hovde image

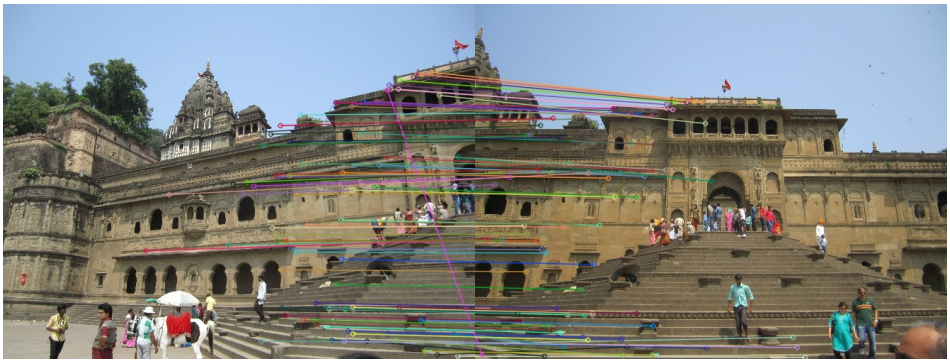


Figure 50: Correspondence between points using SIFT in the temple image

# 4 Obtained Results Using SIFT

Figures 49–52 show the obtained results when using the SIFT.

**Observations (Harris vs SIFT):**

- The results indicate that SIFT outperforms the Harris Corner Detection algorithm. This was the expected behavior since SIFT is invariant to scale and rotation. The mentioned improvement can be clearly seen looking at the images provided in the instructions: the hovde image and the template image. For example, in both images we see that many of the corners obtained from the trees when using the Harris Corner Detector where miss-matched with other corners. The main reason for that behavior was that the trees did not appear in both images. Therefore, there was not a correct correspondence for those corners. When using SIFT, these miss-matches disappear.

- Harris Corner Detection is sensitive to changes in scale, which reduces its reliability compared to SIFT, as observed in the temple image, for example. Furthermore, SIFT's ability to remain rotation invariant ensures consistent corner detection, as seen in the hovde image, while Harris tends to miss corners when objects are either rotated or scaled, highlighting its limitations in such scenarios.

- For the 2 images that I provided we do not see such a clear improvement in the performance as we could do for the 2 images provided in the instructions. For example, for the TV image, we see that SIFT is more robust to the reflections of the TV screen since less corners are detected on it and as a consequence, a few less miss-matches occur.

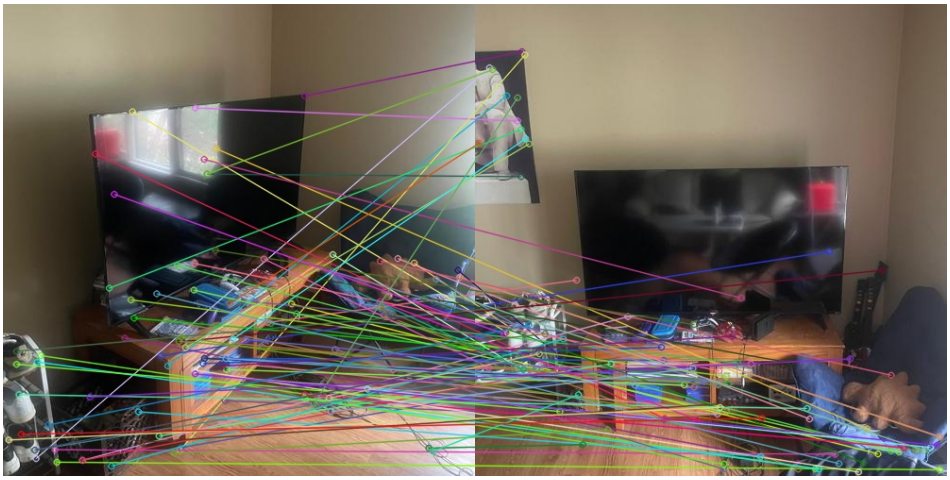


Figure 51: Correspondence between points using SIFT in the tv image

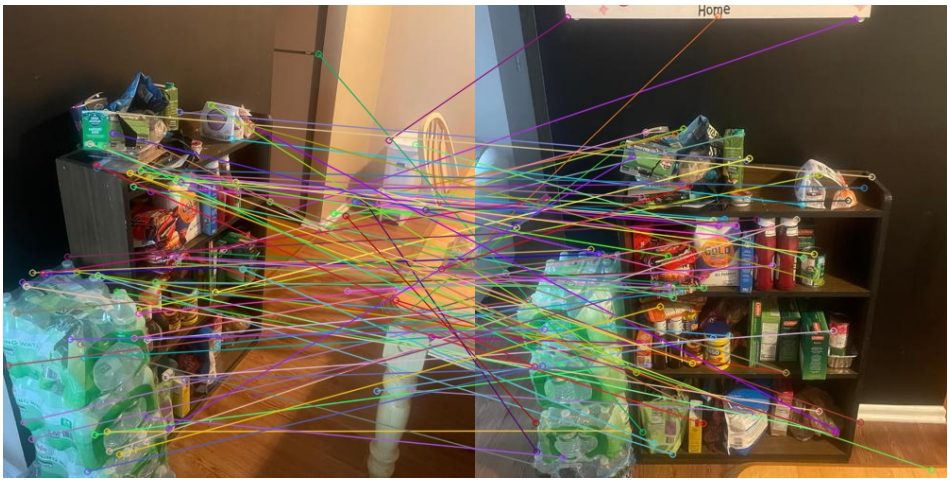



Figure 52: Correspondence between points using SIFT in the shelf image

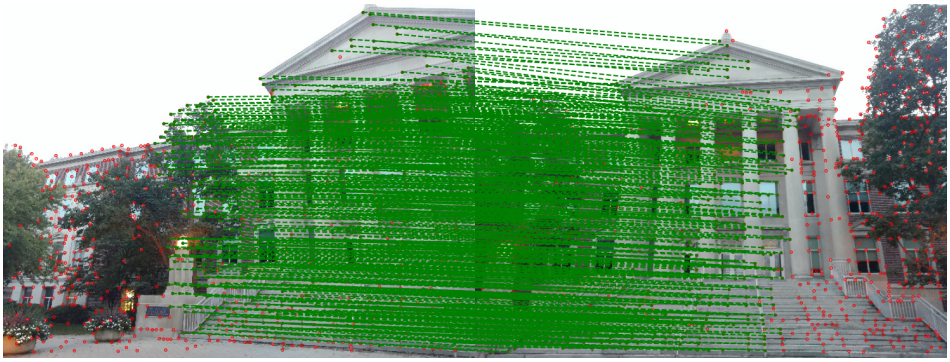


Figure 53: Correspondence between points using SuperPoint in the hovde image

# 5 Obtained Results Using SuperPoint and SuperGlue pipeline

Figures 53–56 show the obtained results when using the SuperPoint and SuperGlue pipeline

**Observations (Comparing SuperGlue with the other approaches):**

- SuperGlue outperforms by far the Harris Corner Detector approach and the SIFT approach.
- The performance of the SIFT approach with the hovde image and the temple image was consistent and showed a clear improvement with respect to the Harris Corner Detector approach although some miss-matches still occurred. These miss-matches are slightly reduced when using the SuperGlue approach. For example, we see that with the SuperGlue approach the points in the triangle in the top of the hovde are correctly matched. Looking at the results from the 2 images provided in the instructions, we can say that the SuperGlue slightly outperforms the SIFT approach.
- With the 2 images that I provided we can see a clear improvement with respect to the 2 methods tested before. The number of miss-matches when using the Harris Corner Detector approach and the SIFT approach for the TV image and the shelf image was considerably high. This number of miss-matches is clearly reduced when using the SuperGlue approach. For example, looking at the TV image, as done in the previous comparisons, we see that SuperGlue is robust to the reflections of the screen of the TV. We also see that most of the points are correctly mactched except of some of them corresponding to the wall. The points corresponding to the objects in the scene are mainly correctly matched. A similar behavior occurs with the shelf image where we can see how all the points from the packages of bottles of water are correctly matched and the same for the different items in the shelf.
- SuperGlue approach is based on a Graph Neural Network (GNN) used to model the relationships between keypoints. SuperGlue is able to match keypoints more effectively by taking into account the context of neighbor and global keypoints. This can result in an improvement of performance in challenging situations such as the TV image and the shelf image.

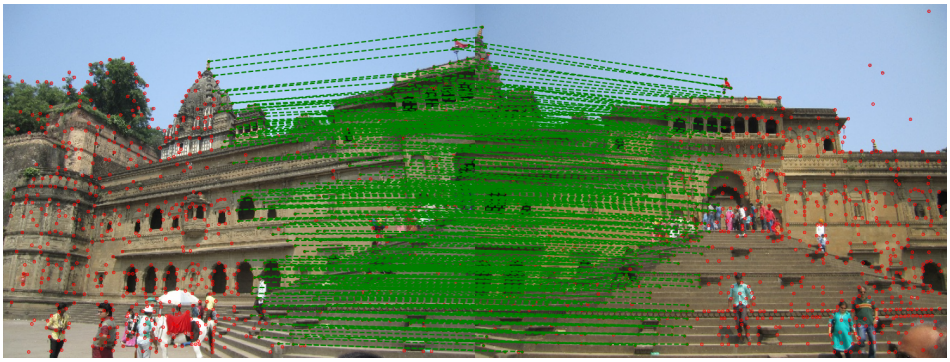


Figure 54: Correspondence between points using SuperPoint in the temple image

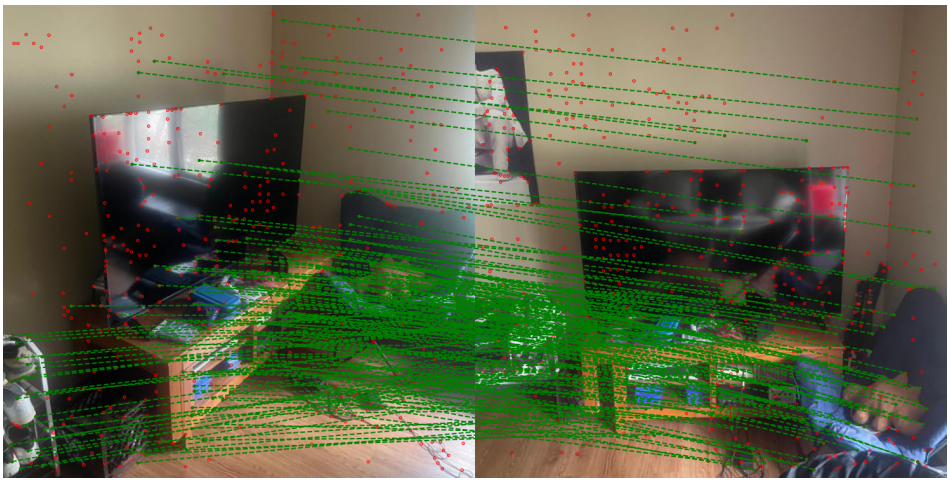


Figure 55: Correspondence between points using SuperPoint in the tv image

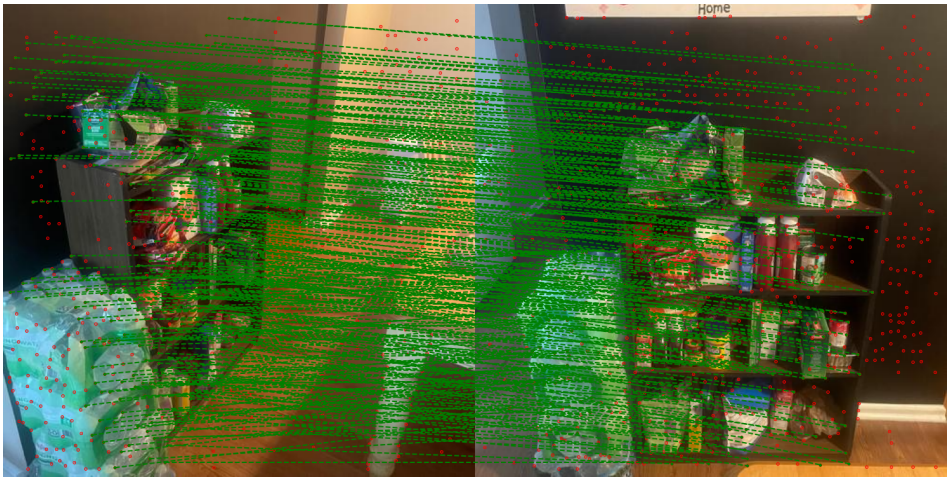



Figure 56: Correspondence between points using SuperPoint in the shelf image

CODE:

```
import matplotlib.pyplot as plt
import random
import numpy as np
import cv2
from scipy.ndimage import maximum_filter

## HARRIS APPROACH-----------------------------------------------------
# Function to draw points
def putPointsOnImage(corners, image):
    for corner in corners:
        cv2.circle(image, tuple(corner), radius=5, color=(42, 247, 237),
thickness =-1)
    return image

# Function to draw points and lines
def putPointsAndLinesOnImage(final_image, w, selected, p1, corners2):
    points_1=(p1)
    points_2 = (corners2[selected] + np.array([w, 0]))
    cv2.circle(final_image, points_1, radius=5, color = (42, 247, 237),
thickness=-1)
    cv2.circle(final_image, points_2, radius = 5, color=(42, 247, 237),
thickness =-1)
    cv2.line(final_image,points_1,points_2,(random.randint(0, 255),
random.randint(0, 255), random.randint(0, 255)), 1)
    return final_image

# Function to get the Harris response
def getHarrisResponse(dx, dy, N):
    s_dxdx = cv2.filter2D(dx*dx, ddepth=-1, kernel=np.ones((N,N)))
    s_dydy = cv2.filter2D(dy*dy, ddepth=-1, kernel=np.ones((N,N)))
    s_dxdy = cv2.filter2D(dx*dy, ddepth=-1, kernel=np.ones((N,N)))
    #Compute the trace and determinant
    tr_c = s_dxdx + s_dydy
    det_c = (s_dxdx*s_dydy)-(s_dxdy*s_dxdy)
    return det_c-0.05*(tr_c**2)

def harris_detect_corner(image, sig):
    #Get gray scale image
    gray_scale_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)/255

    # Build Haar filter
    M = int(np.ceil(4*sig))
    if M%2 == 1:
        M = M + 1
    haar_x = np.zeros((M, M))
    haar_x[:, :M // 2] = -1
    haar_x[:, M // 2:] = 1
```

```
    haar_y = np.zeros((M, M))
    haar_y[:M // 2, :] = 1
    haar_y[M // 2:, :] = -1

    # Get dx and dy by applying Haar filter in both directions
    dx = cv2.filter2D(gray_scale_image, ddepth = -1, kernel = haar_x)
    dy = cv2.filter2D(gray_scale_image, ddepth = -1, kernel = haar_y)

    #Get Harris response and set threshold
    N = int(np.ceil(5*sig))
    if N%2 == 1:
        N = N + 1
    H = getHarrisResponse(dx, dy, N)
    R_abs = np.abs(H)
    thr = np.mean(R_abs)
    K = 2*N

    #Apply maximum filter to get local max values in (2K+1, 2K+1) windows
    R_max_window = maximum_filter(H, size=(2*K+1, 2*K+1))
    #Create a boolean mask where the current pixel is the max in its
window and exceeds threshold
    max_mask = (H == R_max_window) & (H > thr)
    #Find the coordinates of the corners
    y_coords, x_coords = np.nonzero(max_mask)
    #Extract the corresponding R values at those positions
    R_values = H[y_coords, x_coords]
    #Stack coordinates with their corresponding R values
    corners = np.vstack((x_coords, y_coords, R_values)).T
    #Sort corners based on the R value (descending) and take the top 100
    corners_100 = corners[np.argsort(corners[:, 2])][-100:,
:2].astype(int)

    img_with_corners = putPointsOnImage(corners_100, image)

    return [corners_100, img_with_corners]

def ssd(image1, image2, fake_corners_1, fake_corners_2, window):
    #Get gray scale image
    gray_scale_image_1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)/255
    gray_scale_image_2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)/255
    final_image = np.concatenate((image1, image2), axis=1)

    # We remove the pixels that do not have enough margin to get the
window of pixels around them
    corners_1 = fake_corners_1[
        (fake_corners_1[:, 1] - window // 2 > 0) &
        (fake_corners_1[:, 0] - window // 2 > 0) &
        (fake_corners_1[:, 1] + window // 2 <
gray_scale_image_1.shape[0]) &
```

```
        (fake_corners_1[:, 0] + window // 2 <
gray_scale_image_1.shape[1])
    ]
    corners_2 = fake_corners_2[
        (fake_corners_2[:, 1] - window // 2 > 0) &
        (fake_corners_2[:, 0] - window // 2 > 0) &
        (fake_corners_2[:, 1] + window // 2 <
gray_scale_image_2.shape[0]) &
        (fake_corners_2[:, 0] + window // 2 <
gray_scale_image_2.shape[1])
    ]

    #Match corners similar to previous solution
    for p1 in corners_1:
        ssd = np.zeros((len(corners_2),2))
        for j , p2 in enumerate ( corners_2 ):
            image1_window = gray_scale_image_1[p1[1]-
window//2:p1[1]+window//2, p1[0]-window//2:p1[0]+window//2]
            image2_window = gray_scale_image_2[p2[1]-
window//2:p2[1]+window//2, p2[0]-window//2:p2[0]+window//2]
            ssd[j] = np.array([np.sum((image1_window-
image2_window)**2),j])
        final_image = putPointsAndLinesOnImage(final_image,
image1.shape[1], np.argmin(ssd[:,0], axis=0), p1, corners_2)
    return final_image

def ncc(image1, image2, fake_corners_1, fake_corners_2, window):
    #Get gray scale image
    gray_scale_image_1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)/255
    gray_scale_image_2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)/255
    final_image = np.concatenate((image1, image2), axis=1)

    # We remove the pixels that do not have enough margin to get the
window of pixels around them
    corners_1 = fake_corners_1[
        (fake_corners_1[:, 1] - window // 2 > 0) &
        (fake_corners_1[:, 0] - window // 2 > 0) &
        (fake_corners_1[:, 1] + window // 2 <
gray_scale_image_1.shape[0]) &
        (fake_corners_1[:, 0] + window // 2 <
gray_scale_image_1.shape[1])
    ]
    corners_2 = fake_corners_2[
        (fake_corners_2[:, 1] - window // 2 > 0) &
        (fake_corners_2[:, 0] - window // 2 > 0) &
        (fake_corners_2[:, 1] + window // 2 <
gray_scale_image_2.shape[0]) &
        (fake_corners_2[:, 0] + window // 2 <
gray_scale_image_2.shape[1])
```

```
    ]

    #Match corners similar to previous solution
    for p1 in corners_1:
        ncc = np.zeros((len(corners_2),2))
        for j, p2 in enumerate(corners_2):
            image1_window = gray_scale_image_1[p1[1]-
window//2:p1[1]+window//2, p1[0]-window//2:p1[0]+window//2]
            image2_window = gray_scale_image_2[p2[1]-
window//2:p2[1]+window//2, p2[0]-window//2:p2[0]+window//2]
            ncc[j] = np.array([np.sum((image1_window-
np.mean(image1_window))*(image2_window-
np.mean(image2_window)))/np.sqrt((np.sum((image1_window-
np.mean(image1_window))**2))*(np.sum((image2_window-
np.mean(image2_window))** 2))),j])
        if np.max(ncc[:,0])>0.3:
            selected_ncc = np.argmax(ncc[:,0], axis=0)
        else: break
        final_image = putPointsAndLinesOnImage(final_image,
image1.shape[1], selected_ncc, p1, corners_2)
    return final_image

#Example for the hovde image. We would just change the path for the other
images
image1 = cv2.imread("/home/aolivepe/Computer-
Vision/HW4/HW4_images/hovde_1.jpg")
image2 = cv2.imread("/home/aolivepe/Computer-
Vision/HW4/HW4_images/hovde_2.jpg")

for sig in [0.8, 1.2, 1.6, 2.0]:
    #Get corners from first image
    corners1, h_image1 = harris_detect_corner(image1, sig)
    cv2.imwrite(f'./output/corners_1_{str(sig)}.jpeg', h_image1)

    #Get corners from second image
    corners2, h_image2 = harris_detect_corner(image2, sig)
    cv2.imwrite(f'./output/corners_2_{str(sig)}.jpeg', h_image2)

    #Match corners using ssd
    ssd_final = ssd(image1, image2, corners1, corners2, 10)
    cv2.imwrite(f'./output/ssd_{str(sig)}.jpeg', ssd_final)

    #Match corners using ncc
    ncc_final = ncc(image1, image2, corners1, corners2, 10)
    cv2.imwrite(f'./output/ncc_{str(sig)}.jpeg', ncc_final)

## SIFT APPROACH ------------------------------------------------------
--------------------------------
```

```
#Example for the hovde image. We would just change the path for the other
images
image1 = cv2.imread("/home/aolivepe/Computer-
Vision/HW4/HW4_images/hovde_1.jpg")
image2 = cv2.imread("/home/aolivepe/Computer-
Vision/HW4/HW4_images/hovde_2.jpg")

# Create a SIFT detector instance
sift_detector = cv2.SIFT_create()

# Extract keypoints and vectors from both images
kp1, vector1 = sift_detector.detectAndCompute(cv2.cvtColor(image1,
cv2.COLOR_BGR2GRAY), None)
kp2, vector2 = sift_detector.detectAndCompute(cv2.cvtColor(image2,
cv2.COLOR_BGR2GRAY), None)

# Setup the Brute Force Matcher
raw_matches = cv2.BFMatcher(cv2.NORM_L1, crossCheck=True).match(vector1,
vector2)

# Sort matches by distance
sorted_matches = []
for match in raw_matches:
    sorted_matches.append(match)
sorted_matches.sort(key=lambda m: m.distance)

# Draw the top 100 matches on the images
final_image = cv2.drawMatches(image1, kp1, image2, kp2,
sorted_matches[:100], None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Construct the output file name and save the image
cv2.imwrite("./output/sift.jpeg", final_image)
```