ECE 661 Homework 4

Michael Goldberg

September 23 2024

1 Theoretical Question

Let's first start by deriving what the LoG looks like for any given image:

$$LoG(I) = \nabla^2(G * I) \tag{1}$$

In this equation, G is the Gaussian function, and I is the image. However, this is able to be applied as a single filter rather than as two disparate operations:

$$\nabla^2 G(x, y) = \frac{\partial^2 G}{\partial x^2} + \frac{\partial^2 G}{\partial y^2} \tag{2}$$

Now let's expand out the Gaussian function and apply the Laplacian:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp -\frac{x^2 + y^2}{2\pi\sigma^2} \tag{3}$$

Taking the double derivative with respect to x and y and summing, we are left with:

$$LoG(I) = -\frac{(2\sigma^2 - x^2 - y^2) \exp -\frac{x^2+y^2}{2\pi\sigma^2}}{2\pi\sigma^6} \tag{4}$$

Now, we should evaluate what the DoG operator does to an image. For small variations in σ, we can write the Gaussian function as:

$$G_{\sigma 2}(x, y) \approx G_{\sigma 1}(x, y) + \frac{\partial G_\sigma}{\partial \sigma} \tag{5}$$

Therefore, it can be seen that the difference of Gaussians follows the relationship: $G_{\sigma 2} - G_{\sigma 1} = -\frac{\partial G_\sigma}{\partial \sigma}$ Let's compute this derivative and see the result:

$$DoG(I) = \frac{(2\sigma^2 - x^2 - y^2) \exp -\frac{x^2+y^2}{2\pi\sigma^2}}{2\pi\sigma^5} \tag{6}$$

It can be clearly seen then that:

$$LoG(I) = \frac{-1}{\sigma} DoG(I) \tag{7}$$

Notice how LoG and DoG are proportional to one another by a constant factor of $\frac{-1}{\sigma}$, which when applied over an entire kernel, does not change the contrast differences between elements along the kernel (as all pixels within the kernel are scaled by the same factor). Therefore, since DoG is more computationally efficient to find as it only requires a single derivative over a single variable, it can be used in place of the LoG operator.

2 Programming Tasks

2.1 Harris Corner Detection

The following task was to write a Harris Corner Detection algorithm that can extract corner points from multiple images. These corners will then be used to establish correspondences between image pairs of taken of the same scene using the SSD and the NCC metrics. Below are the 2 image pairs used:

(a) Hovde Image 1

(b) Hovde Image 2

(c) Temple Image 1

(d) Temple Image 2

My Harris Corner detection algorithm follows the following process. First, I convolved a Haar Wavelet Filter in the x- and y-directions with the image to produce the gradients dx along x and dy along y. The Haar Wavelet Filter is of the form:

$$\begin{pmatrix} 1 & -1 \\ 1 & -1 \end{pmatrix} \tag{8}$$

for the x-direction and

$$\begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix} \tag{9}$$

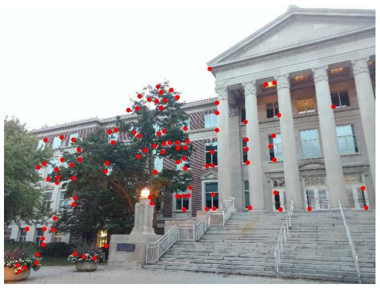
for the y-direction, scaled up to an $M \times M$ matrix, where M is the smallest even integer greater than 4σ.

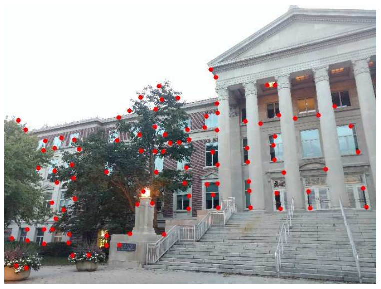
We now want to produce the C matrix, which is of the form:

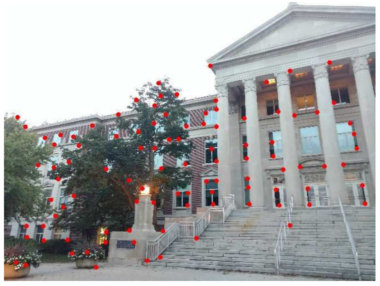
$$C = \begin{bmatrix} \sum d_x^2 & \sum d_x d_y \\ \sum d_x d_y & \sum d_y^2 \end{bmatrix} \tag{10}$$

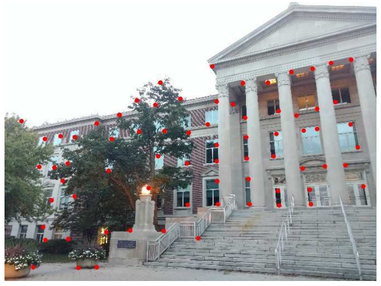
The Harris response can then be expressed as $\det(C) - k\,\mathrm{Tr}(C)^2$, where k is a sensitivity factor whose value is usually between 0.04 and 0.06. My implementation used $k = 0.05$. A pixel is said to be a detected corner if the harris response is greater than the mean of the absolute value of the collection of harris responses across all pixels.

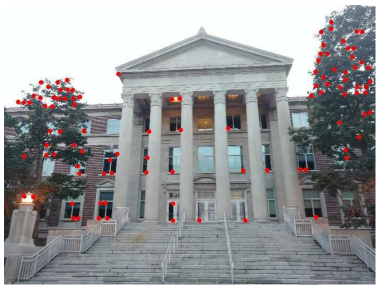
The results of this algorithm can be seen below, with varying levels of sigma (0.8, 1.2, 1.4, and 2) and additionally using 2 new pairs of images that I took of a keyboard and an oscilloscope.


(a) Hovde Image 1, $\sigma = 0.8$

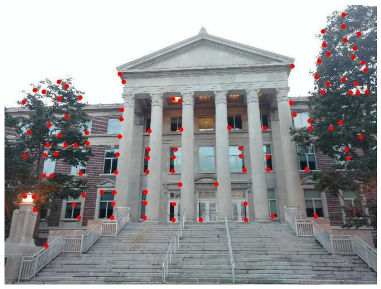

(b) Hovde Image 1, $\sigma = 0.1.2$


(c) Hovde Image 1, $\sigma = 1.4$

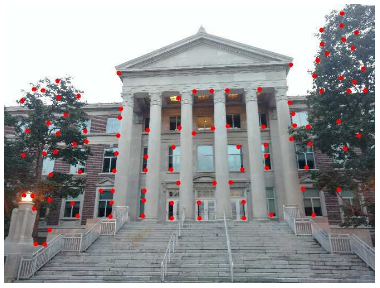

(d) Hovde Image 1, $\sigma = 2$



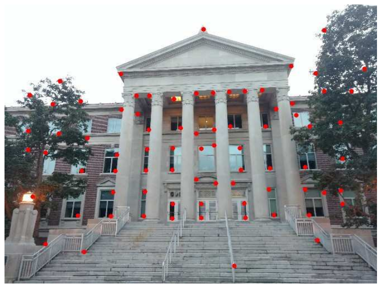
(a) Hovde Image 2, $\sigma = 0.8$



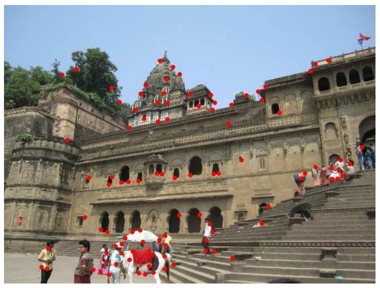
(b) Hovde Image 2, $\sigma = 0.1.2$



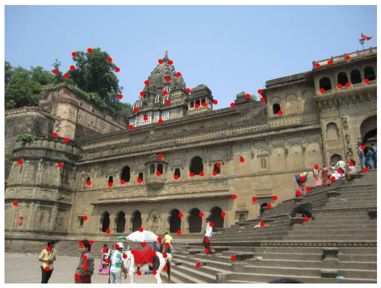
(c) Hovde Image 2, $\sigma = 1.4$



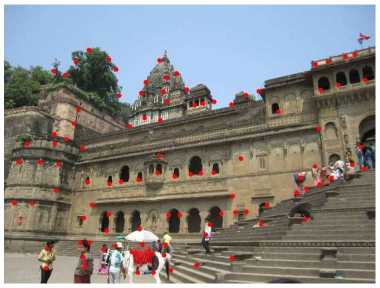
(d) Hovde Image 2, $\sigma = 2$



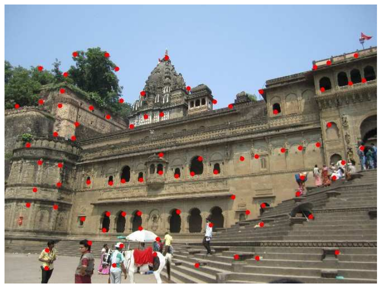
(a) Temple Image 1, $\sigma = 0.8$



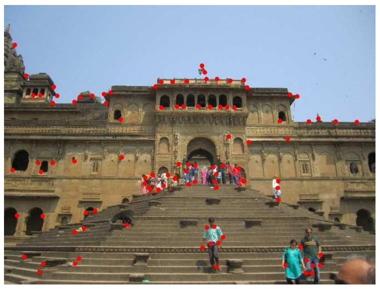
(b) Temple Image 1, $\sigma = 0.1.2$

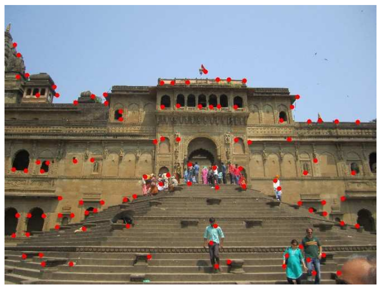


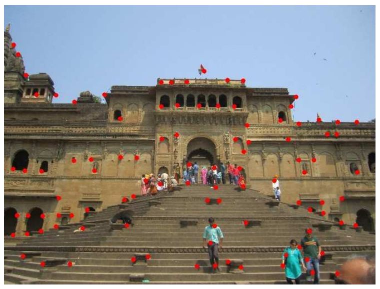
(c) Temple Image 1, $\sigma = 1.4$

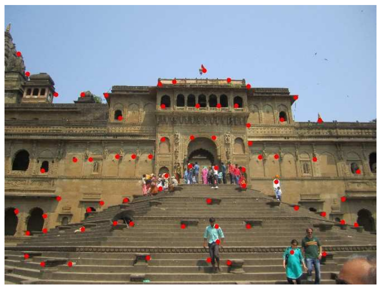


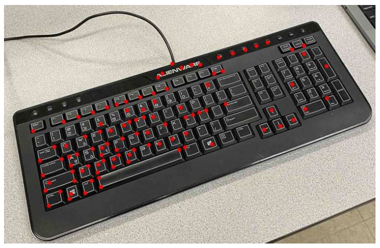
(d) Temple Image 1, $\sigma = 2$

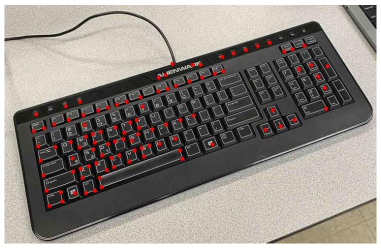

(a) Temple Image 2, $\sigma = 0.8$

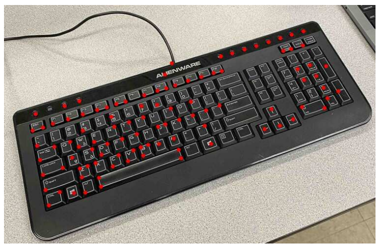

(b) Temple Image 2, $\sigma = 0.1.2$

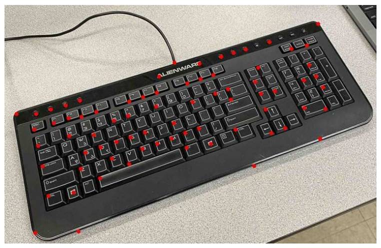

(c) Temple Image 2, $\sigma = 1.4$

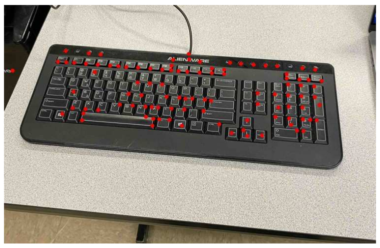

(d) Temple Image 2, $\sigma = 2$

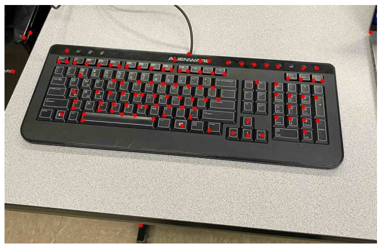

(a) Keyboard Image 1, $\sigma = 0.8$

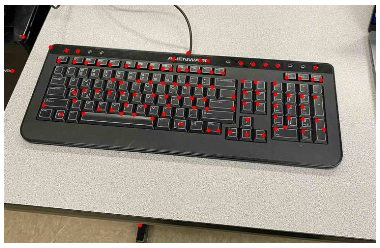

(b) Keyboard Image 1, $\sigma = 0.1.2$

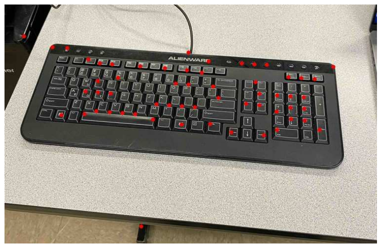

(c) Keyboard Image 1, $\sigma = 1.4$

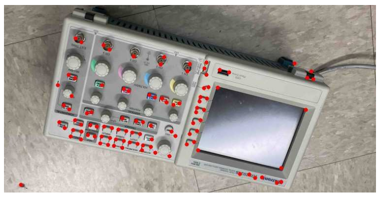

(d) Keyboard Image 1, $\sigma = 2$

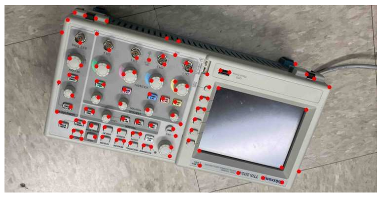

(a) Keyboard Image 2, $\sigma = 0.8$

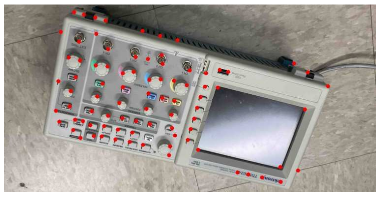

(b) Keyboard Image 2, $\sigma = 0.1.2$

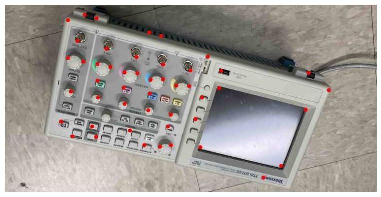

(c) Keyboard Image 2, $\sigma = 1.4$


(d) Keyboard Image 2, $\sigma = 2$


(a) Oscilloscope Image 1, $\sigma = 0.8$


(b) Oscilloscope Image 1, $\sigma = 0.1.2$


(c) Oscilloscope Image 1, $\sigma = 1.4$


(d) Oscilloscope Image 1, $\sigma = 2$

(a) Oscilloscope Image 2, $\sigma = 0.8$

(b) Oscilloscope Image 2, $\sigma = 0.1.2$

(c) Oscilloscope Image 2, $\sigma = 1.4$

(d) Oscilloscope Image 2, $\sigma = 2$

It can be seen that $\sigma = 2$ gave some pretty nice results with clear corners identified across all of the images. Lower values of σ seem to pick up noise from other things happening within the image, yet are not strictly corners, for example the people in the temple image. Therefore, $\sigma = 2$ was used for the feature matching processes listed below.

Now that corner points were identified for all images, we now were tasked with using the Sum of Squared Differences (SSD) and Normalized Cross Correlations (NCC) metrics to find correspondences between the keypoints within the image pairs. Let's start with the process for SSD.

First, a kernel of size (39×39) was chosen. This kernel needs to be sufficiently large with dimensions that are odd, and I found that this size worked the best for this task.

The SSD metric is as follows:

$$SSD = \sum_i \sum_j |f_1(i,j) - f_2(i,j)|^2 \tag{11}$$

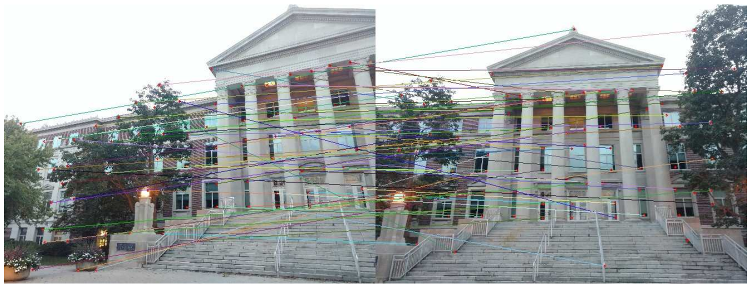
where f_1 is image 1 and f_2 is image 2 in the pair. i and j are indices within the kernel.

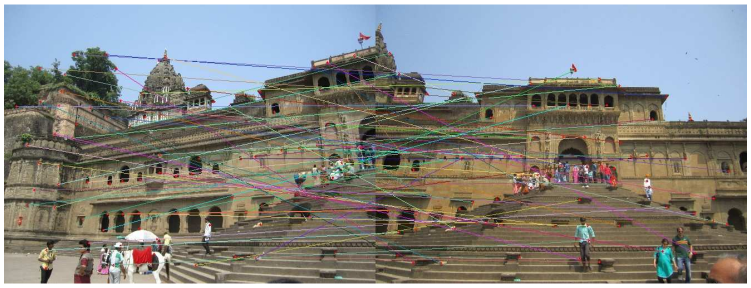
We can apply this kernel to each interest point in image 1 and each potential matching interest point in image 2. The minimization of this metric signifies a match between the image 1 point and the found image 2 point. This can be repeated across all image 1 interest points, resulting in a match being found in image 2 for each one.

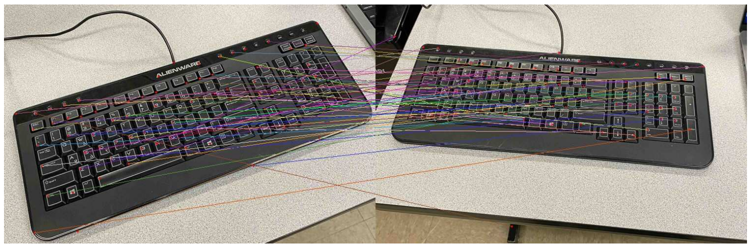
A similar process can be done with NCC, however a different metric is used, and the maximization signifies a match. Here is the metric used for NCC:

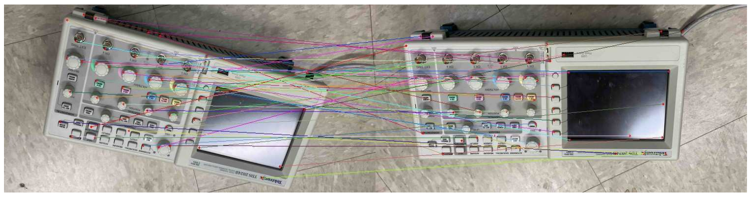
$$NCC = \frac{\sum_i \sum_j (f_1(i,j) - m_1)(f_2(i,j) - m_2)}{\sqrt{(\sum_i \sum_j (f_1(i,j) - m_1))^2 (\sum_i \sum_j (f_1(i,j) - m_1))^2}} \tag{12}$$

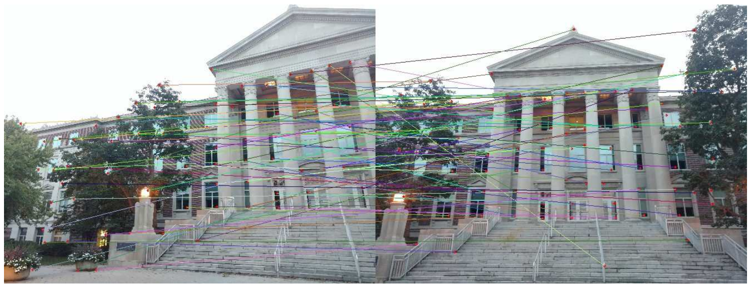
The results of both of these approaches can be seen below for all four image pairs:


(a) Hovde Images SSD Matching

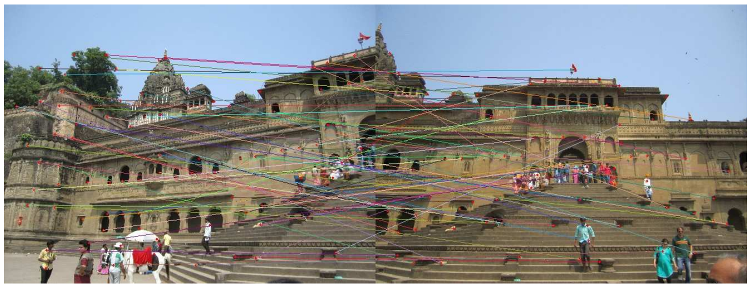

(b) Temple Images SSD Matching


(c) Keyboard Images SSD Matching

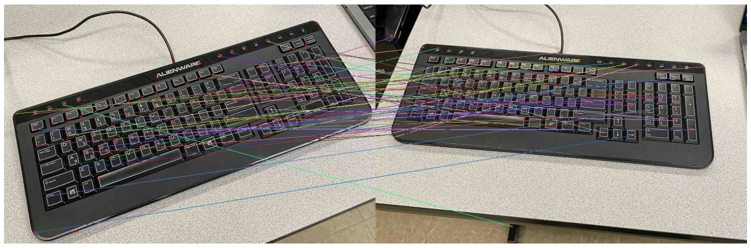

(d) Oscilloscope Images SSD Matching



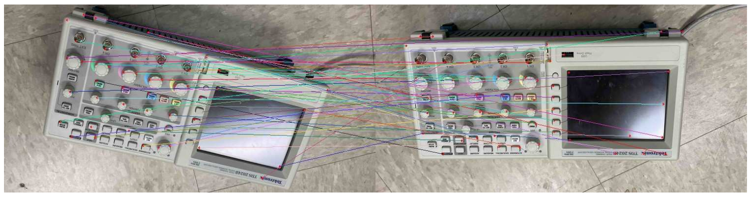
(a) Hovde Images NCC Matching



(b) Temple Images NCC Matching



(c) Keyboard Images NCC Matching



(d) Oscilloscope Images NCC Matching

It seems that SSD works better for certain types of interest points while NCC works better for others. For example, SSD works great for points lying near a high contrast area within the Hovde image to the bottom right of the lamp on the stairs, yet NCC matches tree points better. A better approach would possibly be to perform the SSD and NCC correspondence matching bidirectionally (from image 1 to 2 and then from image 2 to 1), and taking only the correspondences that both directions agree upon.

2.2 Scale Invariant Feature Transform (SIFT) Feature Matching

The first step in SIFT is the scale-space construction. This allows use the detect features even when the image is scaled. This is achieved by blurring the images using different gaussian filters and finding points that are invariant across this gaussian blur pyramid. If a gaussian blurred image is denoted by $G(x, y, \sigma)$, then the pyramid is constructed using the DoG:

$$D(x, y, \sigma) = G(x, y, k\sigma) - G(x, y, \sigma) \tag{13}$$

, where k is a constant factor for scaling the gaussian.

Next, SIFT finds keypoints by searching for extrema within the DoG pyramid space. Each pixel in the DoG pyramid is compared to its 26 neighbors (8 in the same level, 9 in the level above, and 9 in the level below). If the pixel is a maximum or minimum, it is a candidate for being a keypoint.

SIFT then performs rejection of points that are unstable or not well-defined. This involves a second-order Taylor expansion of the DoG to find the keypoint in sub-pixel space. It also involves edge and low-contrast rejection, with keypoints along the edges or in low-contrast settings being rejected, as these are less stable than those in higher contrast settings and those more inset within the image. This is checked using the Hessian $H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$, whose eigenvalues indicate curvature. If a point has high curvature along one direction but low along another, this indicates an edge, and that candidate is rejected.

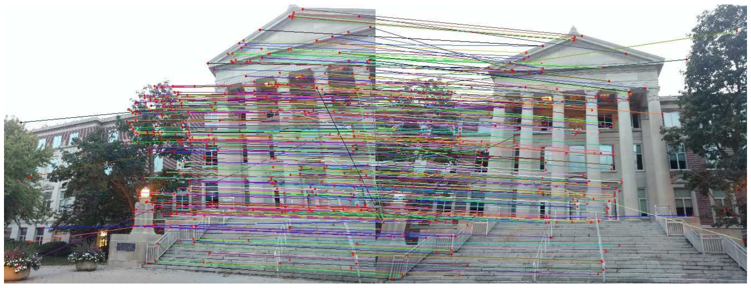
The next step is to assign an orientation to each keypoint using local pixel differences. These gradients can be collected together to form a histogram, whose dominant orientation is assigned to the keypoint. The orientation equation is:

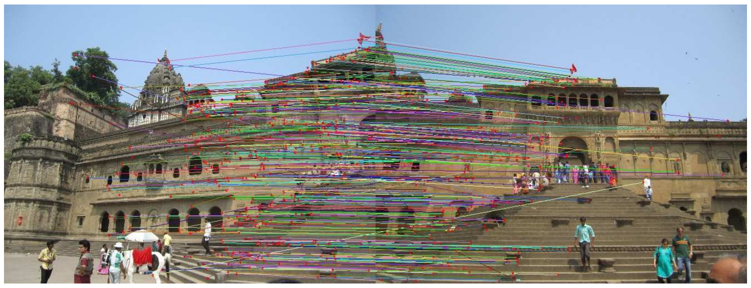
$$\theta(x, y) = \tan^{-1}\left(\frac{G(x, y+1) - G(x, y-1}{G(x+1, y) - G(x-1, y)}\right) \tag{14}$$

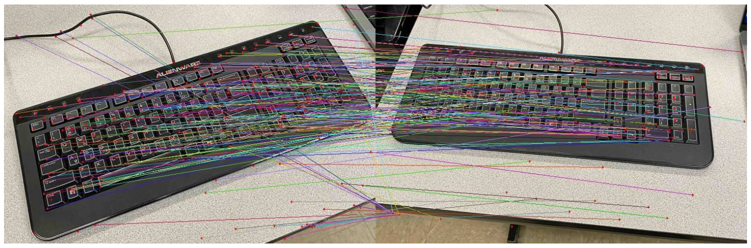
Once assigned an orientation, the keypoints are then labelled with a feature descriptor, which indicates the local image gradients around the keypoint and is highly sensitive to small changes within the image. To create this, a 16×16 region around each keypoint is subdivided into smaller 4×4 regions. Each of these regions get assigned a histogram with the local gradient directions and magnitudes of the pixels in that region. This leaves us with a 128-dimensional vector that describes each keypoint.

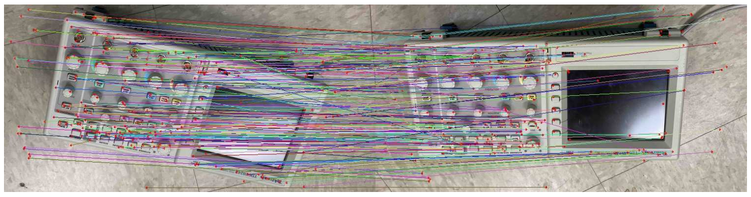
Finally, keypoints between two images can be matched by finding the smallest Euclidean distance between descriptor vectors. The Lowe Ratio test is used to ensure that a match is strong - this is simply if the ratio of the distances between the closest match and the second-closest match is less than 0.75, then the match is reliable.

Below are the results from SIFT. It can be seen that the interest point detection outperforms the Harris implementation from before, and the feature matching far outperforms both SSD and NCC from before as well. However, with the feature point detection, it is important to note that Harris is designed solely to identify corners whereas SIFT is designed to find points of high contrast or high curvature, so there are some points SIFT finds that are not corners but are still important to the image context for matching.


(a) Hovde Images SIFT Feature Point Identification and Matching

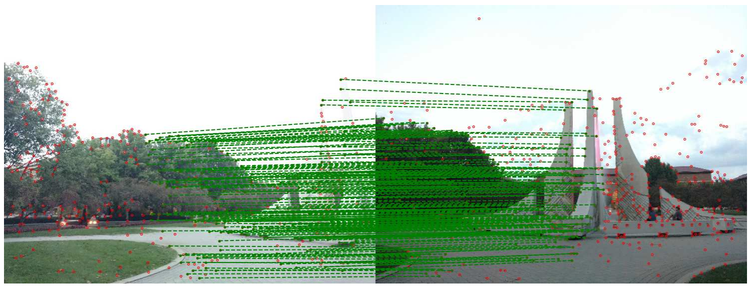

(b) Temple Images SIFT Feature Point Identification and Matching

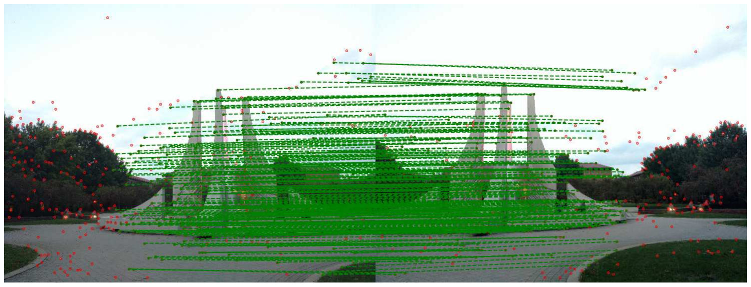

(c) Keyboard Images SIFT Feature Point Identification and Matching

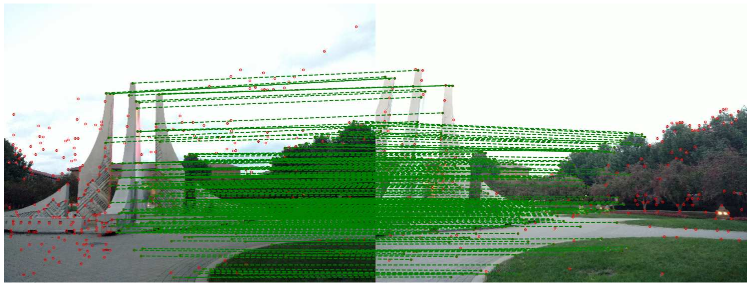

(d) Oscilloscope Images SIFT Feature Point Identification and Matching

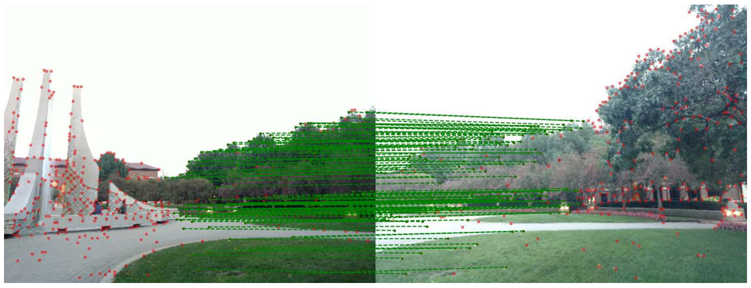
2.3 SuperPoint and SuperGlue Feature Matching

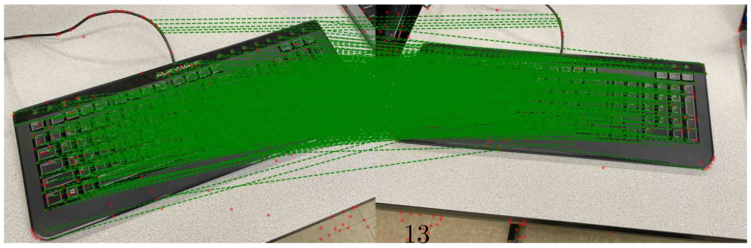
The best performing algorithm was SuperPoint and SuperGlue. It can be seen that even small details like the clouds and trees around the engineering fountain can be detected and matched properly. These results are much better, however, the amount of feature points extracted is quite a lot, as seen in the Keyboard and Oscilloscope images. With SIFT or other classical approaches, there are fewer keypoints in these images, which makes it easier to see if the mappings are proper or not.


(a) SuperPoint + SuperGlue Engineering Fountain Pair 1

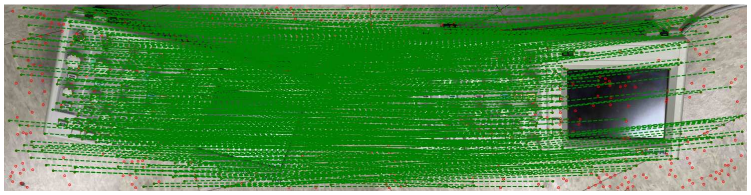

(b) SuperPoint + SuperGlue Engineering Fountain Pair 2


(c) SuperPoint + SuperGlue Engineering Fountain Pair 3


(d) SuperPoint + SuperGlue Engineering Fountain Pair 4



(e) SuperPoint + SuperGlue Keyboard Pair



3 Code

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import random

def harris_corner_detection(image, sigma, k, namestring):
 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) / 255

 # Finding M (lowest even integer > 4*sigma)
 if(int(sigma * 4) % 2 == 0):
  kernel_size = int(sigma * 4) + 2
 else:
  kernel_size = int(sigma * 4) + 1

 # Creating and applying haar wavelet kernels
 kernel_x = np.ones((kernel_size, kernel_size), dtype=np.float32)
 kernel_y = np.ones((kernel_size, kernel_size), dtype=np.float32)
 kernel_x[:, :int(kernel_size / 2)] = -1
 kernel_y[int(kernel_size / 2):, :] = -1
 dx = cv2.filter2D(gray, -1, kernel_x)
 dy = cv2.filter2D(gray, -1, kernel_y)


 # Computing C matrix components
 dx2 = dx ** 2
 dy2 = dy ** 2
 dxdy = dx * dy

 # Kernel of 1s to perform a sum across the kernel
 summation_kernel = np.ones((int(5 * sigma), int(5 * sigma)), dtype=np.float32)
 sum_dx2 = cv2.filter2D(dx2, -1, summation_kernel)
 sum_dy2 = cv2.filter2D(dy2, -1, summation_kernel)
 sum_dxdy = cv2.filter2D(dxdy, -1, summation_kernel)

 # Finding harris response and thresholding
 detC = (sum_dx2 * sum_dy2) - (sum_dxdy ** 2)
 traceC = sum_dx2 + sum_dy2

 response = detC - k * (traceC ** 2)
 response_threshold = np.mean(np.abs(response))


 corner_image = np.copy(image)
 # Reducing number of points to most prominent 100
 corners = np.empty((0, 3))
 corner_image = np.copy(image)
 N = int(10 * sigma)

 for x in range(N, gray.shape[1] - N):
  for y in range(N, gray.shape[0] - N):
   response_window = response[y - N:y + N + 1, x - N:x + N + 1]
   response_max = np.max(response_window)
   if response[y, x] == response_max and response_max > response_threshold:
    corners = np.vstack((corners, np.array([x, y, response[y, x]])))

 corners_filtered = np.array(corners[np.argsort(corners[:, 2])])[-100:,:2].astype(int)

 for corner in corners_filtered:
   cv2.circle(corner_image, (corner[0], corner[1]), radius = 4, color = (0, 0, 255), thickne

 # plt.imshow(corner_image)
 # plt.show()

 cv2.imwrite('HW4_images/harris_' + namestring + '_' + str(sigma) + '.jpg', corner_image, [c


 return corners_filtered


def SSD(img1, img2, img1_corners, img2_corners, kernel_size, namestring):
 gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY) / 255
 gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY) / 255

 # Drawing interest points first
 for corner in img1_corners:
  cv2.circle(img1, (corner[0], corner[1]), radius = 2, color = (0, 0, 255), thickness = -1)
 for corner in img2_corners:
  cv2.circle(img2, (corner[0], corner[1]), radius = 2, color = (0, 0, 255), thickness = -1)

 # Creating combined image for visualization
 combined_image = np.concatenate((img1, img2), axis = 1)

 half_k = kernel_size // 2
 matches = []

 # Converting img2 corners to list to allow for point removal to ensure no dual matches
 img2_corners = [tuple(corner) for corner in img2_corners]
 for (x1, y1) in img1_corners:
  best_match = None
  lowest_ssd = float('inf')

  # Skipping corners in img1 too close to edge
  if(x1 - half_k < 0 or x1 + half_k >= img1.shape[1] or y1 - half_k < 0 or y1 + half_k >= in
   continue

  # Creaitng kernel at each image1 feature point
  window1 = img1[y1 - half_k : y1 + half_k + 1, x1 - half_k : x1 + half_k + 1]

  for (x2, y2) in img2_corners:
   # Skipping corners in img2 too close to edge
   if(x2 - half_k < 0 or x2 + half_k >= img2.shape[1] or y2 - half_k < 0 or y2 + half_k >= i
    continue

   # Creating kernel at each image2 feature point
   window2 = img2[y2 - half_k : y2 + half_k + 1, x2 - half_k : x2 + half_k + 1]

   # Calculating SSD metric across both kernels
   ssd = np.sum((window1 - window2) ** 2)

   # Updating lowest ssd value
   if(ssd < lowest_ssd):
    lowest_ssd = ssd
    best_match = (x2, y2)

  # If a match is found, draw a line and remove img2 point from potential candidates for fut
  if(best_match):
   cv2.line(combined_image, (x1, y1), (best_match[0] + img1.shape[1], best_match[1]), color
   img2_corners.remove(best_match)

 # plt.imshow(cv2.cvtColor(combined_image, cv2.COLOR_BGR2RGB))
 # plt.show()

 cv2.imwrite('HW4_images/SSD_' + namestring + '.jpg', combined_image, [cv2.IMWRITE_JPEG_QUAL


def NCC(img1, img2, img1_corners, img2_corners, kernel_size, namestring):
 gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY) / 255
 gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY) / 255

 for corner in img1_corners:
  cv2.circle(img1, (corner[0], corner[1]), radius = 2, color = (0, 0, 255), thickness = -1)
 for corner in img2_corners:
```

```
 cv2.circle(img2, (corner[0], corner[1]), radius = 2, color = (0, 0, 255), thickness = -1)

combined_image = np.concatenate((img1, img2), axis = 1)

half_k = kernel_size // 2
matches = []

img2_corners = [tuple(corner) for corner in img2_corners]

for (x1, y1) in img1_corners:
 best_match = None
 highest_ncc = -1

 # Skipping corners in img1 too close to edge
 if(x1 - half_k < 0 or x1 + half_k >= img1.shape[1] or y1 - half_k < 0 or y1 + half_k >= im
  continue

 window1 = img1[y1 - half_k : y1 + half_k + 1, x1 - half_k : x1 + half_k + 1]

 window1_mean = np.mean(window1)

 for (x2, y2) in img2_corners:
  # Skipping corners in img2 too close to edge
  if(x2 - half_k < 0 or x2 + half_k >= img2.shape[1] or y2 - half_k < 0 or y2 + half_k >= i
   continue

  window2 = img2[y2 - half_k : y2 + half_k + 1, x2 - half_k : x2 + half_k + 1]

  window2_mean = np.mean(window2)

  # Calculating NCC metric piecewise
  numerator = np.sum((window1 - window1_mean) * (window2 - window2_mean))
  denominator = np.sqrt((np.sum(window1 - window1_mean)) ** 2 * np.sum((window2 - window2_m

  # Ensuring no division by zero
  if denominator != 0:
   ncc = numerator / denominator
  else:
   ncc = 0

  if(ncc > highest_ncc):
   highest_ncc = ncc
   best_match = (x2, y2)

 # If a match is found, draw a line and remove img2 point from potential candidates for fut
 if(best_match):
    cv2.line(combined_image, (x1, y1), (best_match[0] + img1.shape[1], best_match[1]), color
    img2_corners.remove(best_match)

  # plt.imshow(cv2.cvtColor(combined_image, cv2.COLOR_BGR2RGB))
  # plt.show()

  cv2.imwrite('HW4_images/NCC_' + namestring + '.jpg', combined_image, [cv2.IMWRITE_JPEG_QUAL

def sift(img1, img2, namestring):
  gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
  gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

  combined_image = np.concatenate((img1, img2), axis = 1)

  # Instantiating sift object, creating keypoints and descriptors
  sift = cv2.SIFT_create()
  keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
  keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

  # Using k-nearest neighbors for matching, and applying Lowe Ratio test to determine if matc
  matcher = cv2.DescriptorMatcher_create(cv2.DescriptorMatcher_FLANNBASED)

  matches = matcher.knnMatch(descriptors1, descriptors2, k=2)

  good_matches = []
  for m, n in matches:
   if m.distance < 0.75 * n.distance:
    good_matches.append(m)

  # Draw lines between matches
  for match in good_matches:
   img1_idx = match.queryIdx
   img2_idx = match.trainIdx

   (x1, y1) = keypoints1[img1_idx].pt
   (x2, y2) = keypoints2[img2_idx].pt

   x2 += img1.shape[1]

   cv2.circle(combined_image, (int(x1), int(y1)), 2, color=(0, 0, 255), thickness=-1)
   cv2.circle(combined_image, (int(x2), int(y2)), 2, color=(0, 0, 255), thickness=-1)

   cv2.line(combined_image, (int(x1), int(y1)), (int(x2), int(y2)), color=(random.randint(0,

  cv2.imwrite('HW4_images/SIFT_' + namestring + '.jpg', combined_image, [cv2.IMWRITE_JPEG_QUA
 hovde1 = cv2.imread('HW4_images/hovde_2.jpg')
 hovde2 = cv2.imread('HW4_images/hovde_3.jpg')
 temple1 = cv2.imread('HW4_images/temple_1.jpg')
 temple2 = cv2.imread('HW4_images/temple_2.jpg')

 keyboard1 = cv2.imread('HW4_images/keyboard1.jpg')
 keyboard2 = cv2.imread('HW4_images/keyboard2.jpg')
 oscilloscope1 = cv2.imread('HW4_images/oscilloscope1.jpg')
 oscilloscope2 = cv2.imread('HW4_images/oscilloscope2.jpg')


 sigmas = [0.8, 1.2, 1.4, 2] # replace with custom values

 for sigma in sigmas:
  hovde1_corners = harris_corner_detection(hovde1, sigma, 0.05, 'hovde1')
  hovde2_corners = harris_corner_detection(hovde2, sigma, 0.05, 'hovde2')
  temple1_corners = harris_corner_detection(temple1, sigma, 0.05, 'temple1')
  temple2_corners = harris_corner_detection(temple2, sigma, 0.05, 'temple2')

  keyboard1_corners = harris_corner_detection(keyboard1, sigma, 0.05, 'keyboard1')
  keyboard2_corners = harris_corner_detection(keyboard2, sigma, 0.05, 'keyboard2')
  oscilloscope1_corners = harris_corner_detection(oscilloscope1, sigma, 0.05, 'oscilloscope1
  oscilloscope2_corners = harris_corner_detection(oscilloscope2, sigma, 0.05, 'oscilloscope2


 SSD(hovde1, hovde2, hovde1_corners, hovde2_corners, 39, 'hovde')
 SSD(temple1, temple2, temple1_corners, temple2_corners, 39, 'temple')
 SSD(keyboard1, keyboard2, keyboard1_corners, keyboard2_corners, 21, 'keyboard')
 SSD(oscilloscope1, oscilloscope2, oscilloscope1_corners, oscilloscope2_corners, 21, 'oscill

 NCC(hovde1, hovde2, hovde1_corners, hovde2_corners, 39, 'hovde')
 NCC(temple1, temple2, temple1_corners, temple2_corners, 39, 'temple')
 NCC(keyboard1, keyboard2, keyboard1_corners, keyboard2_corners, 21, 'keyboard')
 NCC(oscilloscope1, oscilloscope2, oscilloscope1_corners, oscilloscope2_corners, 21, 'oscill

 sift(hovde1, hovde2, 'hovde')
 sift(temple1, temple2, 'temple')
 sift(keyboard1, keyboard2, 'keyboard')
 sift(oscilloscope1, oscilloscope2, 'oscilloscope')
```