Purdue University

ECE 661 Computer Vision

Homework 4

Submission: Arjun Kramadhati Gopi

Email: akramadh@purdue.edu

Tasks for this homework

We have two broad tasks for this homework assignment. They are:

- Theoretical question
- Programming tasks

Solution 1 - Theoretical question

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

LoG stands for Laplacian of Gaussian. By now we know from the lectures that every edge operator must contain some sort of a smoothing operator that must be applied before estimating the gradients and the direction vectors. For LoG operator, we have the Gaussian smoothing operator give by:

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

Once we have the smoothed image, we apply the laplacian to get the double derivatives in the x and y directions at each pixel. The laplacian is given by:

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

Together we can write the LoG operator as:

$$h(x,y) = (\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}) g(x,y) \tag{3}$$

Or:

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

The task here is to prove that we can approximate LoG with DoG. DoG stands for difference of Gaussian. We need to prove that:

$$LoG(f(x,y)) = \frac{\partial}{\partial\sigma} ff(x,y,\sigma) \tag{5}$$

That is, we have to prove:

$$\frac{\partial}{\partial\sigma} ff(x,y,\sigma) = \sigma\nabla^2 ff(x,y,\sigma) \tag{6}$$

Considering the RHS in equation 6, we know from the lectures that:

$$\nabla^2 ff(x,y,\sigma) = f(x,y) * h(x,y,\sigma) \tag{7}$$

Where $h(x,y,\sigma)$ is as shown in equation 4. Now taking the LHS in equation 6, we have the relation:

$$\frac{\partial}{\partial\sigma} ff(x,y,\sigma) = \iint f(x',y') \frac{\partial}{\partial\sigma} (e^{\frac{-((x-x')^2+(y-y')^2)}{2\sigma^2}}) dx'dy' \tag{8}$$

This can then be written as:

$$\frac{\partial}{\partial\sigma}ff(x,y,\sigma) = \iint f(x',y')(\frac{-1}{2\pi^3}+\frac{1}{2\pi\sigma^2}(-((x-x')^2+(y-y')^2))\frac{1}{2\pi\sigma^3})(e^{\frac{-((x-x')^2+(y-y')^2)}{2\sigma^2}})dx'dy' \tag{9}$$

Simplifying this we get:

$$\frac{\partial}{\partial\sigma}ff(x,y,\sigma) = \frac{-1}{2\pi\sigma^4}\iint f(x,y)(2-\frac{(x-x')^2+(y-y')^2}{\sigma^2})(e^{\frac{-((x-x')^2+(y-y')^2)}{2\sigma^2}})dx'dy' \tag{10}$$

We can see that equation 10 is of the form:

$$\frac{\partial}{\partial\sigma}ff(x,y,\sigma) = \sigma f(x,y) * h(x,y) \tag{11}$$

The RHS in equation 11 can be substituted based on equation equation 7. Thus we have the final relation:

$$\frac{\partial}{\partial\sigma}ff(x,y,\sigma) = \sigma\nabla^2 ff(x,y,\sigma) \tag{12}$$

Hence we proved that the LoG can approximated by the DoG. This means the LoG can approximated by subtracting the $(\sigma+\delta\sigma)$ smoothed image from the σ smoothed image.

Why LoG is computationally heavier than DoG

To compute the LoG of an image, we have the digital expression that we make use of:

$$h(m,n) = A(1-k\frac{m^2+n^2}{\sigma^2})e^{\frac{-(m^2+n^2)}{2\sigma^2}} \tag{13}$$

This gives us a $(2N+1)X(2N+1)$ operator. Where $-N \geq m,n \leq N$. So, for a sigma value of $\sqrt{2}$ we get $N=6$. This results in a 13X13 operator to estimate the LoG.Comparing this with the DoG, we know that the Gaussian is given by:

$$g(x,y) = \frac{1}{2\pi\sigma^2}e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{14}$$

We can see that the terms x and y are separable. Therefore, the 2D smoothing can be arrived at by two 1D smoothing. One in the x-direction and one in the y-direction. This cannot be done for the LoG. From equation 3 we know how the LoG is represented in the equation form. We can see that the terms x and y are not separable. Therefore for the same σ value of $\sqrt{2}$ we just need a 7 element 1D operator. Even for a 1D LoG operator, we still need a higher sized operator as compared to the DoG. This is because of the criterion that the half width of the oveall operator be three times the half width of central lobe. So for a sigma value of $\sqrt{2}$ we still need a nine element 1D operator for the 1D LoG. Hence, LoG is computationally heavier than DoG. Therefore, computing the DoG and approximating it as the LoG is much more efficient than calculating the LoG.

Solution 2 - Programming Tasks

We have the following programming tasks to achieve in this assignment:

1. Use custom Harris corner detector code to extract interest points from a pair of images. Use SSD and NCC metrics to calculate the correspondence between the interest points in the given pair of images. Do the same task for at least 4 different σ (scale) values.

2. Use **either** SIFT **or** SURF implementations in the openCV library to achieve the same tasks listed in point one.

3. Repeat the tasks one and two for at least two distinct pairs of images clicked by me.

Programming Task 1: Custom Harris corner detector
First, it is essential define what a corner is. A corner is any point whose immediate surrounding region lies on two distinct edge regions. This means to say that a corner is literally the intersection of two distinct non parallel edges.

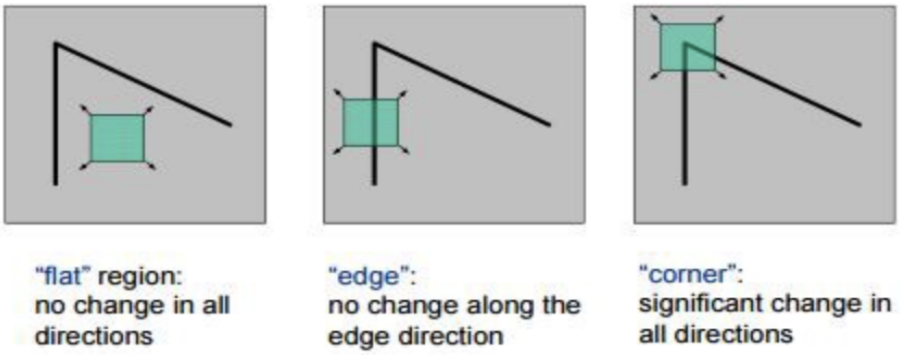

Figure 1: **Image credit: Introduction to Harris Corner Detectors, www.medium.com**

From the figure above, we notice a very interesting pattern which makes a corner point invaluable to many image processing algorithms. Suppose we take a 'interest-area' or a 'window-of-vision' which is shown by the green patch in the images. If we move this patch on an area in the image which has no features, we don't see any difference at all. That is, each new area covered, looks exactly like the previous areas. The same is not the case when we move the patch on a feature like an edge or a corner. In the case of an edge, if we move the patch along the edge direction, we still have the same problem i.e there is no change in the scene. But when it comes to a corner, the situation changes completely. When the patch is on a corner region, whichever direction you move the patch, the scene changes very significantly. This property makes a corner point very important in identifying interest points for correspondence between images.

Identifying corner points:
We use the above mentioned property of a corner to detect them. We move the patch around and check if the 'scene' varies by a significant degree (decided by a threshold). If it does, then we conclude that the scene is a corner region and then we identify the corner point. To do this, we need to achieve the following sub tasks:

- Build Haar filters (one for x-direction and one for y-direction) to obtain the gradients of the gray scale variation in both x and y directions. By convolving the filters on the pixels we obtain the said gradients. The haar filter is built based on the size of the scale value σ where the filter size is:

  $$ceil(4 * \sigma) X ceil(4 * \sigma)$$

  For the Harr filter in x-direction, the filter comprises of '-1' for the first half of the columns and '1' for the second half. For the Haar filter in the y-direction the filter comprises of '1' for the first half of the rows and '-1' for the second half of the rows.

- The next task would be find out which region shows the highest variation in 'scene'. This we do by maximising the intensity function C. C is the intensity matrix which has the summations of the gradient changes in the x and y directions. If HF_x and HF_y are the Haar filters coordinate (x,y) for the two cardinal directions then we calculate the d_x and d_y values. Since we are using a window around the pixel, let us take the window size as at least 5σ X 5σ. Thus we obtain the C matrix which is given by:

$$C = \begin{bmatrix} \sum (d_x)^2 & \sum d_x d_y \\ \sum d_x d_y & \sum (d_y)^2 \end{bmatrix}$$

- From our initial reasoning on how the scenes change on the features we know that at a corner, there are variations in both the x and y directions. There is variation in only one direction on an edge (depending on how it is oriented) and there are no variations on a region with no feature. Therefore, for a scene with a corner, the C matrix will have a rank of two as the value $\sum d_x d_y$ will be non-zero.

- Using the above facts, we then calculate the two eigen values at each pixel from the C matrix. If these values are:

$$\lambda_1, \lambda_2$$

then we calculate a score value **R** which is used to determine, finally, whether the point is a corner point or not. We calculate R using the following relation:

$$R = det(C) - k(trace(C))^2$$

we know that:

$$det(C) = \lambda_1 \lambda 2$$

and

$$trace(C) = \lambda_1 + \lambda_2$$

We calculate the k value by computing the following term:

$$k = \frac{\sum \frac{\lambda_1 \lambda 2}{(\lambda_1 + \lambda_2)^2}}{(h * w)}$$

Where h and w are the pixel height and pixel width of the image. So, k is basically the averaged value of the term:

$$\sum \frac{\lambda_1 \lambda 2}{(\lambda_1 + \lambda_2)^2}$$

at each pixel in the image.

- Finally, we sort the R values by descending order. The highest values of R represent the corner points. It is evident from our solution that there will be a lot of points with very high R values in a single region (containing only one corner). To remove this problem of 'overlapping pseudo corners" we perform non-maxima suppression. The idea of using a non-maxima suppression was borrowed from it's popular implementation to solve the problem of 'overlapping bounding boxes' in object detection challenges. This idea was also borrowed from the previous year's solution. However, the implementation is original and I have used list comprehensions and numpy built-in functions to speed up the process.

Programming Task 2: SSD and NCC feature matching

Once we have the interest points (the corners we detected in the previous step), we apply SSD or NCC algorithms to match the corresponding interest points in the given pair of images.

Sum of Squared Differences (SSD)

To find the correspondence or the feature matching between the images using SSD, we basically find the patches in the images which have the closest resemblance to each other. Suppose we have a corner point P_1 in **Image 1** and corner point P_2 in **Image 2**. Let us take a window around these points and name them $window_{P1}$ and $window_{P2}$. Then, we say that these two points are matching points if the gray levels in $window_{P1}$ and $window_{P2}$ show the **least** variation. To measure this difference, we measure the sum of the squared differences of the pixel values in the two patches. We use the following relation to calculate SSD:

$$SSD = \sum_x \sum_y (window1_{(x,y)} - window2_{(x,y)})^2$$

Normalized Cross Correlation (NCC)

We use similar reasoning we used for the SSD method above to calculate the feature matching using NCC. But for NCC we use a slightly different approach to calculate the variation in the gray levels between the two patches. We additionally calculate the mean gray level in each of the window/patch. For NCC, we use:

$$NCC = \frac{\sum_x \sum_y (window1_{(x,y)} - Mean_1)(window2_{(x,y)} - Mean_2)}{\sqrt{(\sum_x \sum_y (window1_{(x,y)} - Mean_1)^2)(\sum_x \sum_y (window2_{(x,y)} - Mean_2)^2)}}$$

By calculating either of the two scores (SSD/NCC) we establish feature matching between the interest points in the two images.

Programming notes

While implementing SSD/NCC algorithms in the code we need to make sure we follow these approaches:

- **SSD**: The sum of squared differences between the windows around each corner point gives us the comparison between the amplitude of the pixel values. We are directly comparing the variations in the pixels in one window with the variations in the other. So, the SSD value has to be the **least** of all the comparisons if two points are said to be matched. Therefore, in the code, For each corner point in Image 1 we calculate SSD values with all the corner points in Image 2. After this, we find out the corner point in Image 2 which has the least SSD value with the point in question in Image 1. Likewise, we do the same thing for all the points in Image 1. Then, we filter the matched points based on some cutoff value so that we retain only the worthwhile matches.

- **NCC**: The normalised cross correlation gives us the comparison between the variations based on the mean value in each of the windows. The formula we are using will give us the correlation score between the two things we are comparing. So, two points are said to be matched if their correlation scores are the highest. By normalising the scores, we make sure we keep the scores between -1 and 1. Therefore, we ensure we properly compare the amplitudes of the correlations

between even large signals (or windows). Therefore, in the code, we find the point with the **highest** NCC score to make a good match. Then, we filter the matched points based on a cutoff value to retain only worthwhile matches.

SIFT - Scale Invariant Feature Transform

In this homework assignment, we are required to only implement the openCV SIFT library to find the SIFT corner points. Therefore, we will not be writing functions to calculate and estimate the corner points. Hence, I will not be writing in detail about the SIFT algorithm here. Instead I will point out the highlights of what the SIFT library in openCV does and how we finally match the points.

- Using the values of difference of Gaussian across scale and space we select a maxima. A maxima value might be a potential keypoint in that particular scale and space.
- We eliminate low contrast keypoints and edge points from the list of potential keypoints. This is done using a 2 X 2 Hessian Matrix. What we have remaining will thus be the accurate corners in the image.
- We then find the prominent orientation of the keypoint. We do this by calculating the gradient in magnitude and orientation in the image. We use these values as weights to build a histogram. The place where the histogram peaks is the required orientation.
- We then create a 128-dimensional vector at each of the maxima points. By normalising we then get a unit vector of the same form. This vector acts as a descriptor for the keypoint or the corner.
- Lastly, for feature matching between the two images, we can use eucledian distances between the corresponding windows. We can also use the principle of K-Nearest Neighbor (KNN) where k=2.

Results

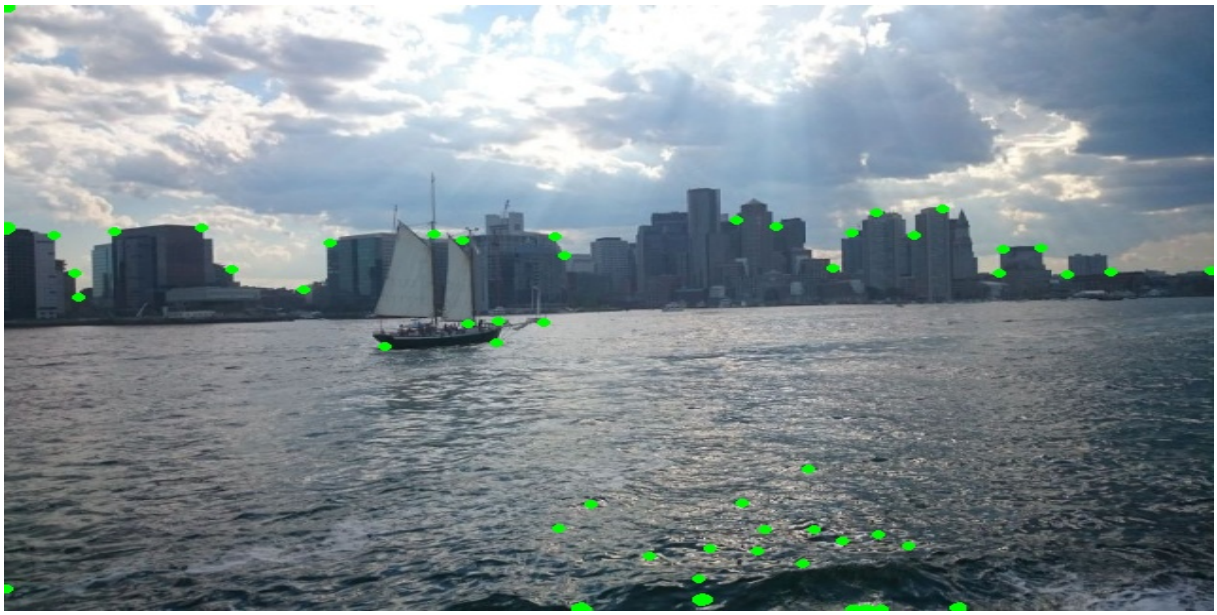

Figure 2: **Harris corner detection with sigma = 0.7**

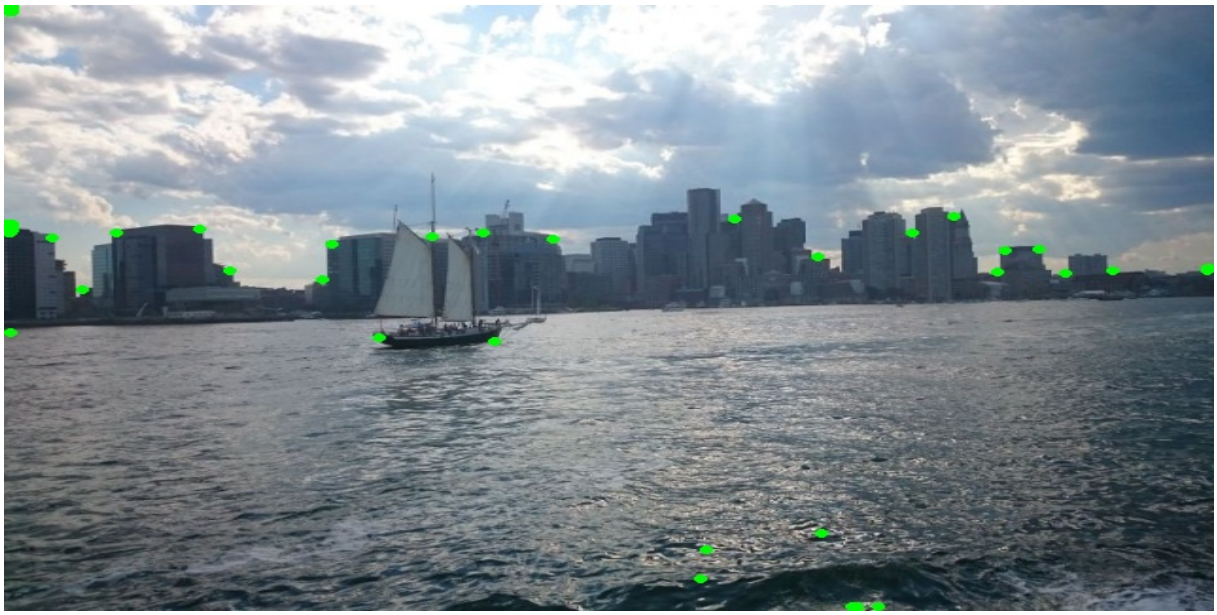

Figure 3: **Harris corner detection with sigma = 1.2**

Figure 4: **Harris corner detection with sigma = 1.8**

Figure 5: **Harris corner detection with sigma = 2.4**

Figure 6: **Harris corner detection with sigma = 0.7**

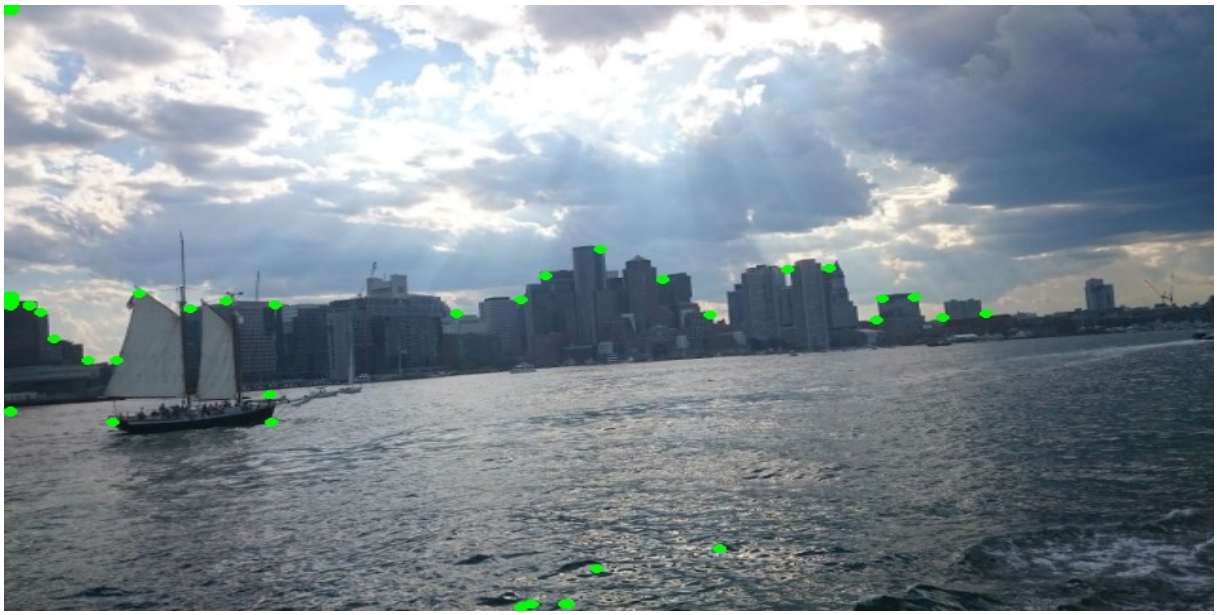

Figure 7: **Harris corner detection with sigma = 1.2**

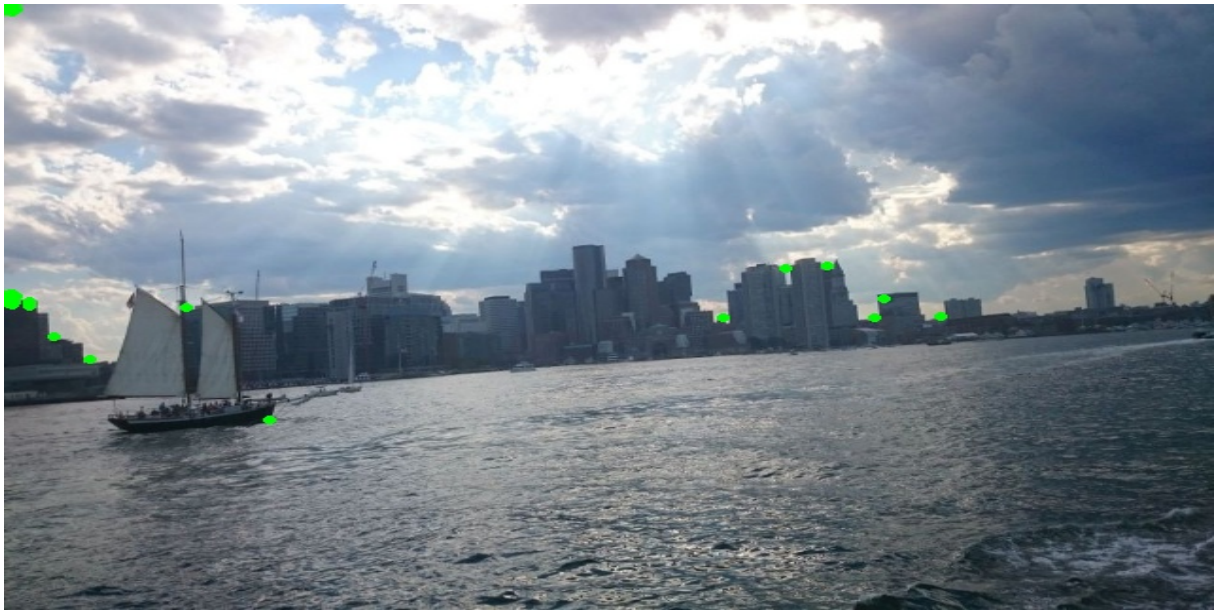

Figure 8: **Harris corner detection with sigma = 1.8**

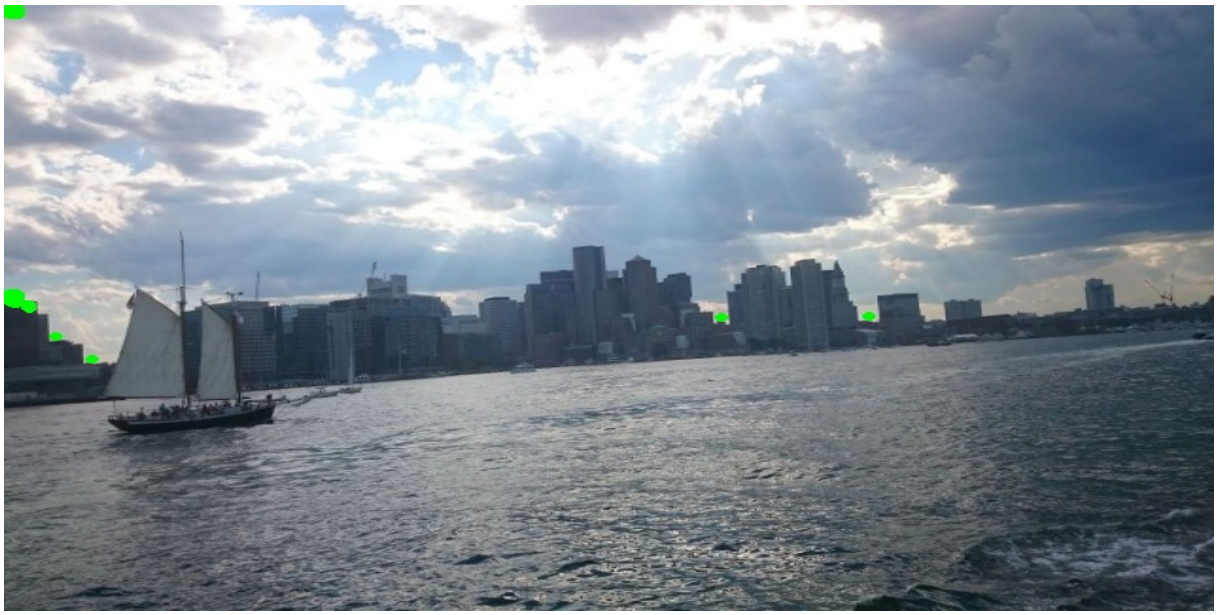

Figure 9: **Harris corner detection with sigma = 2.4**

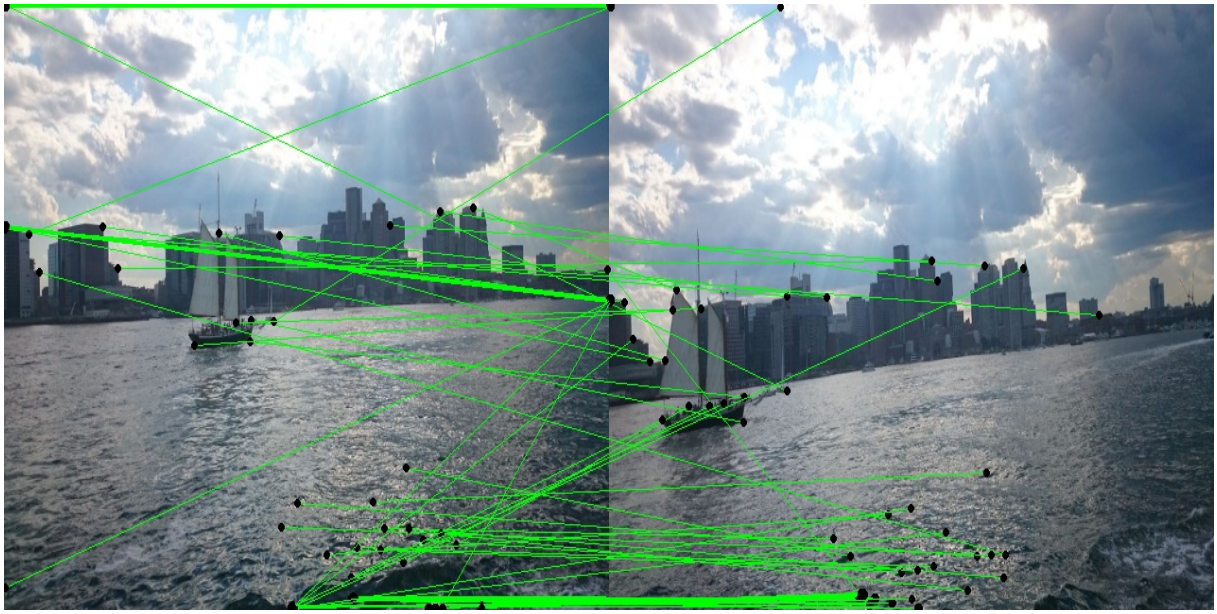

Figure 10: **SSD correspondence at sigma = 0.7**

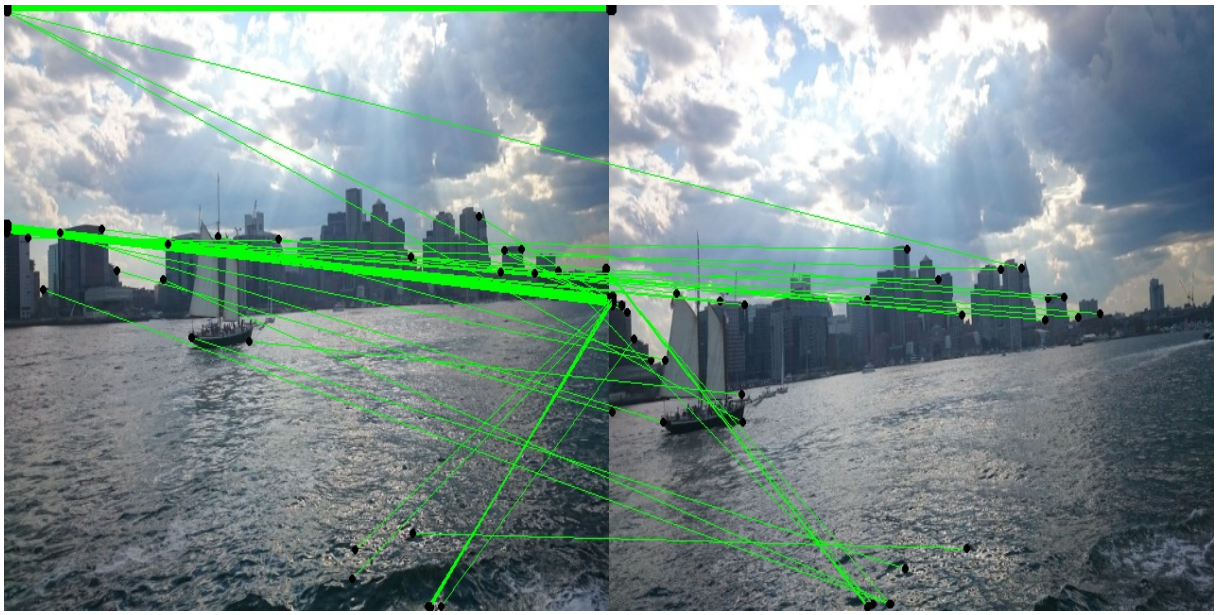

Figure 11: **SSD correspondence at sigma = 1.2**

Figure 12: **SSD correspondence at sigma = 1.8**

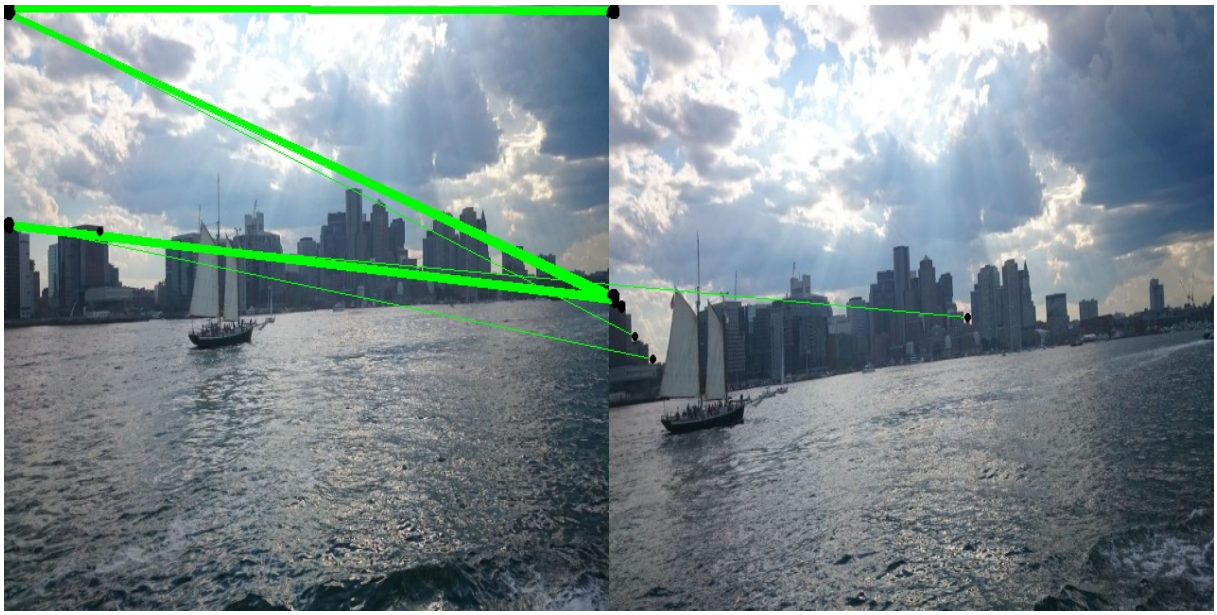

Figure 13: **SSD correspondence at sigma = 2.4**

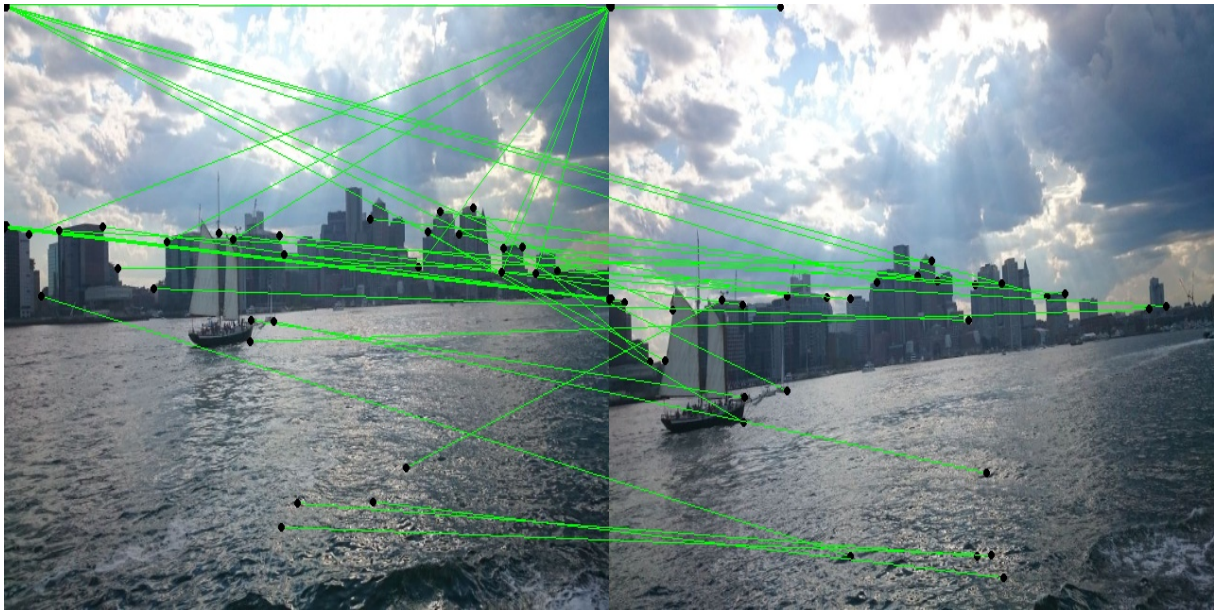

Figure 14: **NCC correspondence at sigma = 0.7**

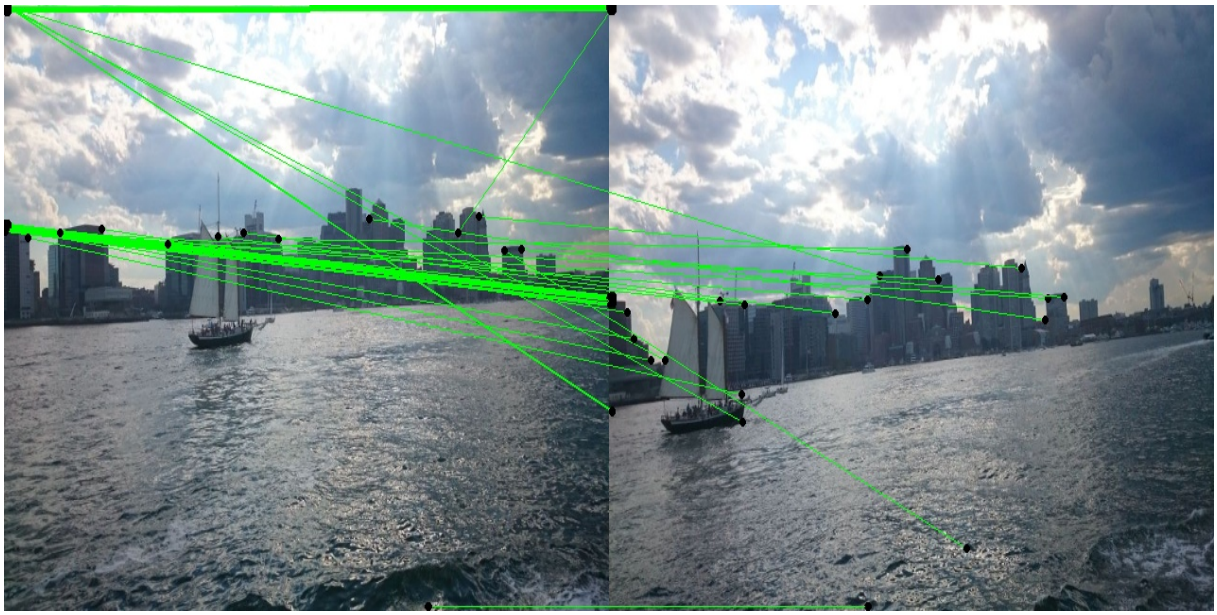

Figure 15: **NCC correspondence at sigma = 1.2**

Figure 16: **NCC correspondence at sigma = 1.8**

Figure 17: **NCC correspondence at sigma = 2.4**

Figure 18: **Sift corner detection**

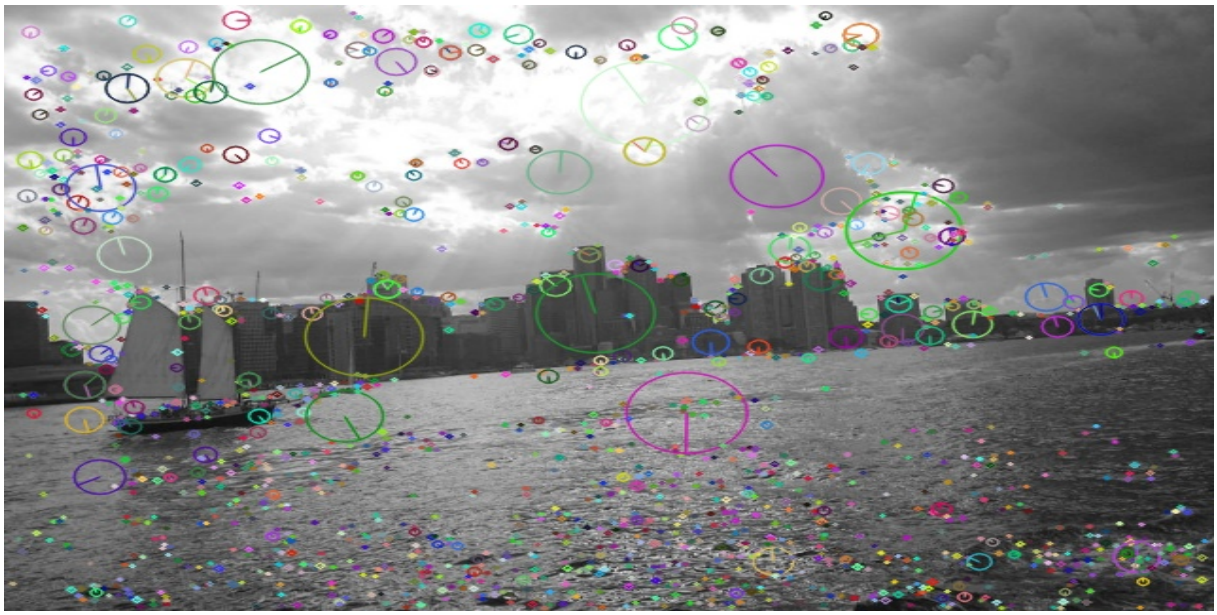

Figure 19: **Sift corner detection**

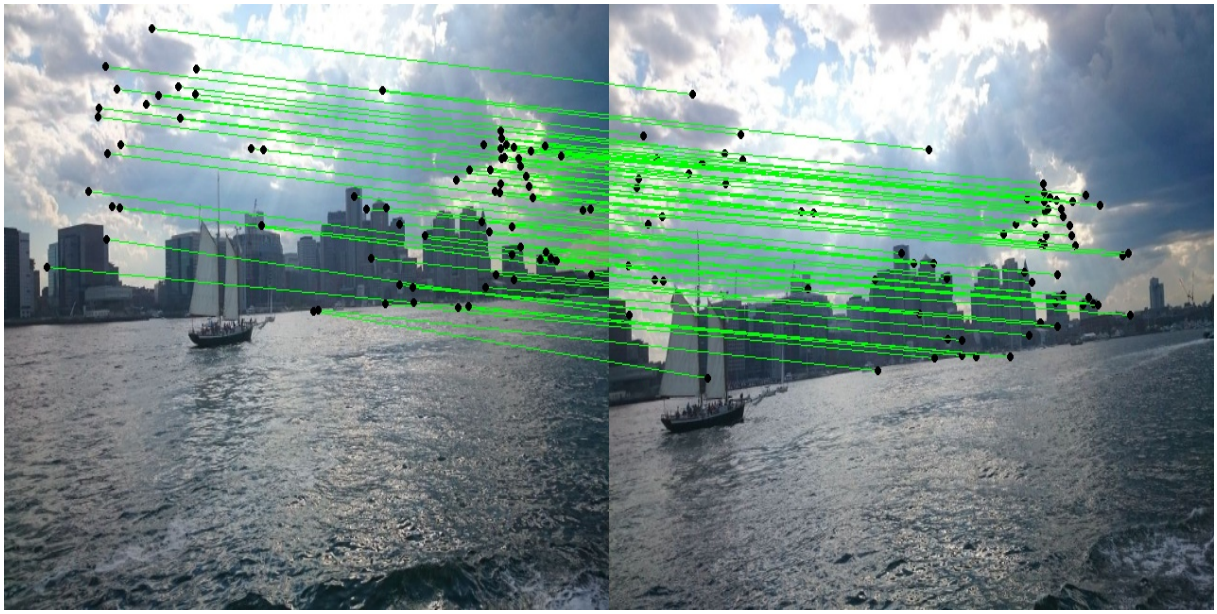

Figure 20: **Eucledian matching for SIFT**

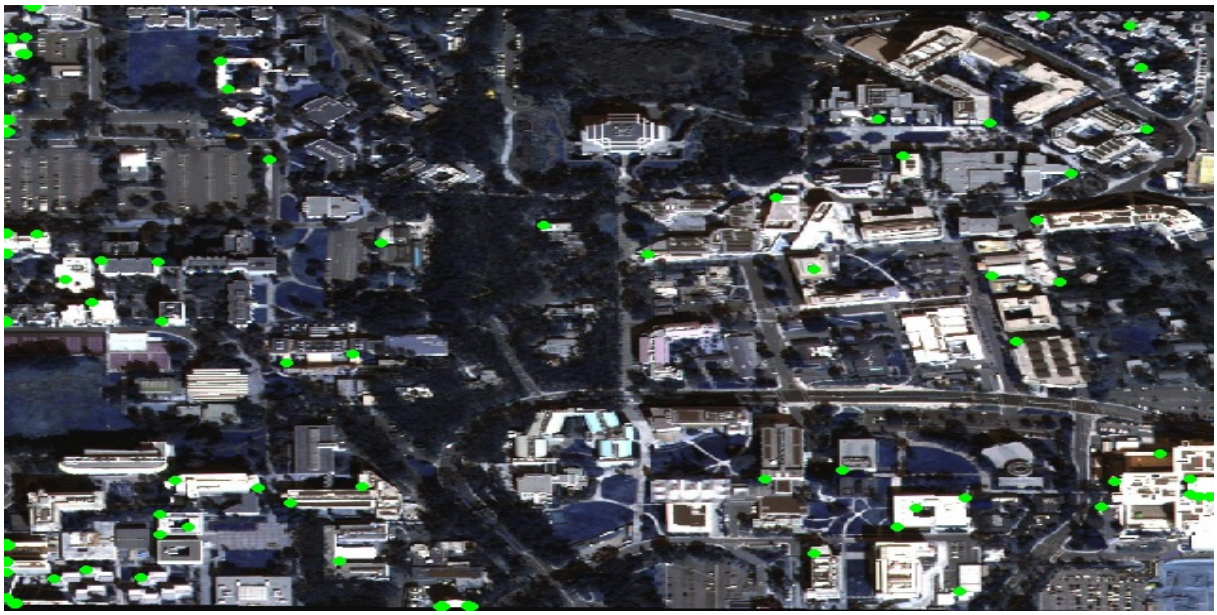

Figure 21: **Harris corner detection with sigma = 0.7**

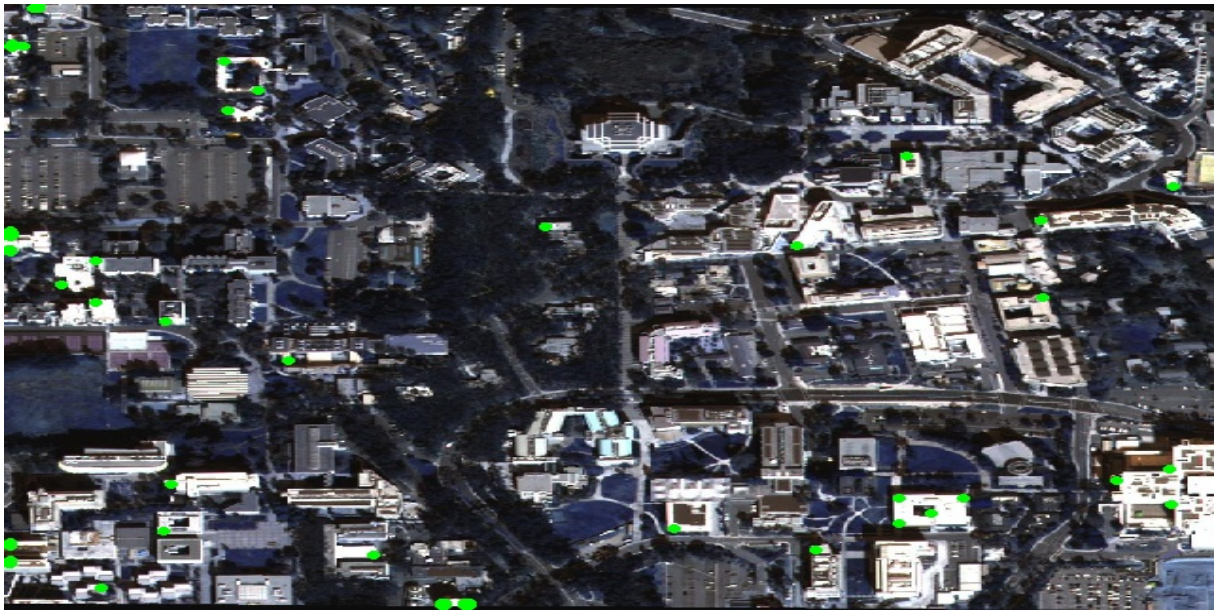

Figure 22: **Harris corner detection with sigma = 1.2**

Figure 23: **Harris corner detection with sigma = 1.8**

Figure 24: **Harris corner detection with sigma = 2.4**

Figure 25: **Harris corner detection with sigma = 0.7**

Figure 26: **Harris corner detection with sigma = 1.2**

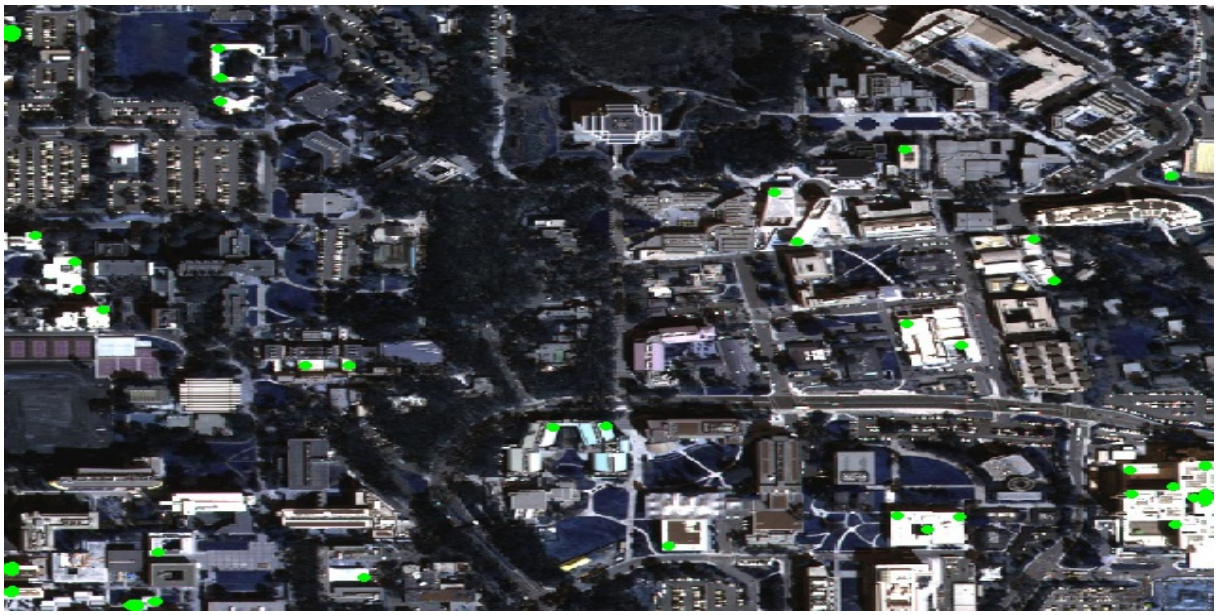

Figure 27: **Harris corner detection with sigma = 1.8**

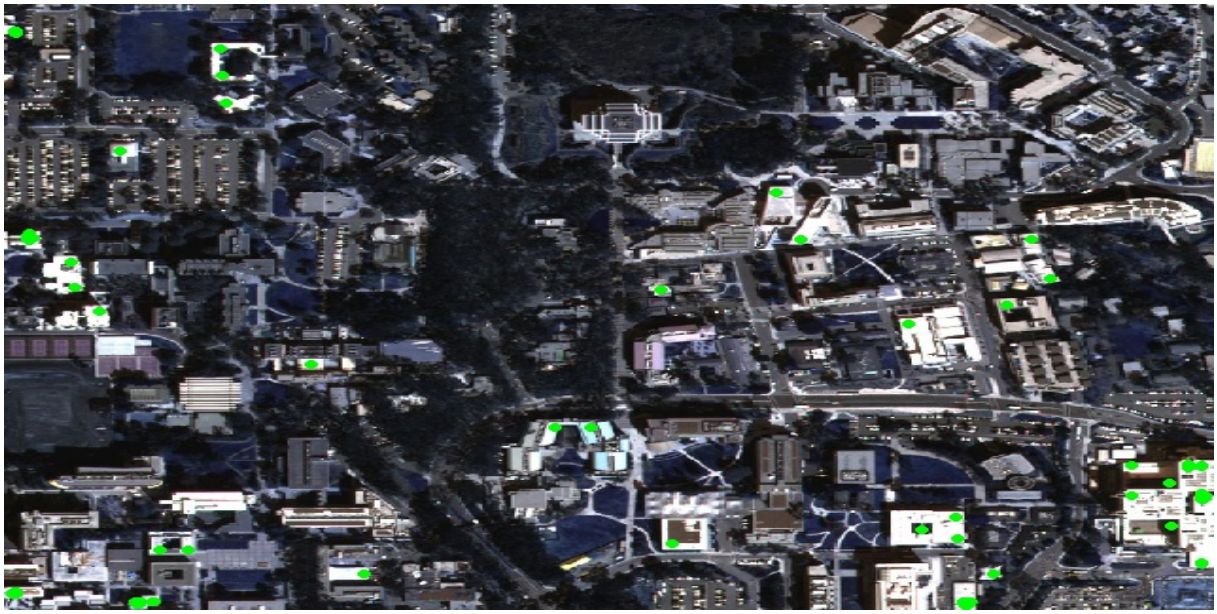

Figure 28: **Harris corner detection with sigma = 2.4**

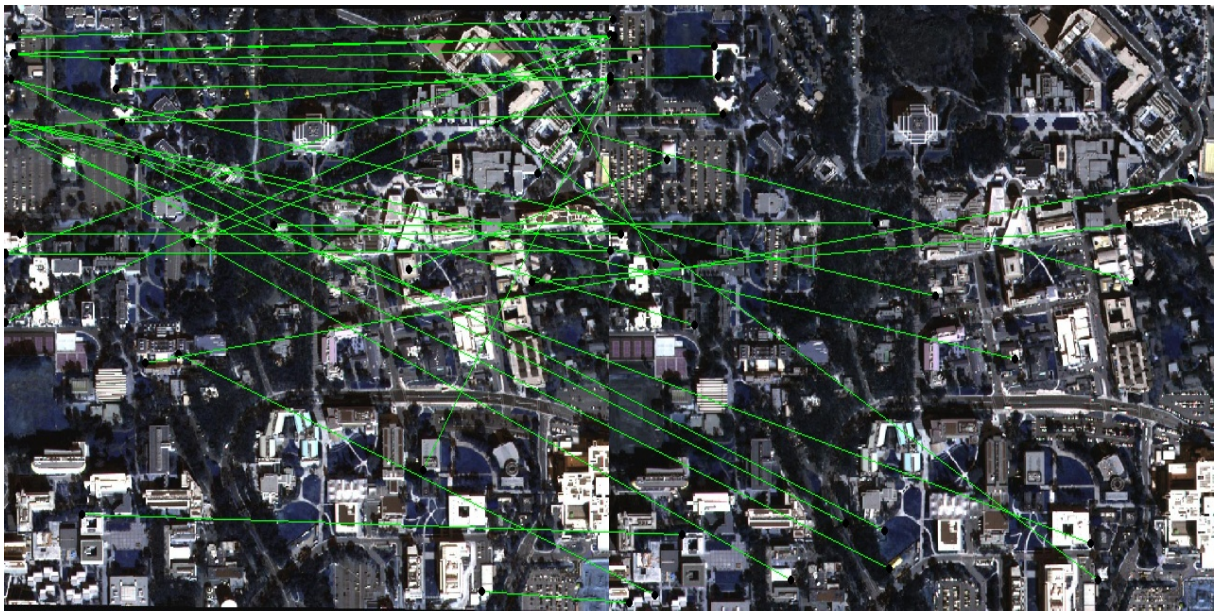

Figure 29: **SSD correspondence at sigma = 0.7**

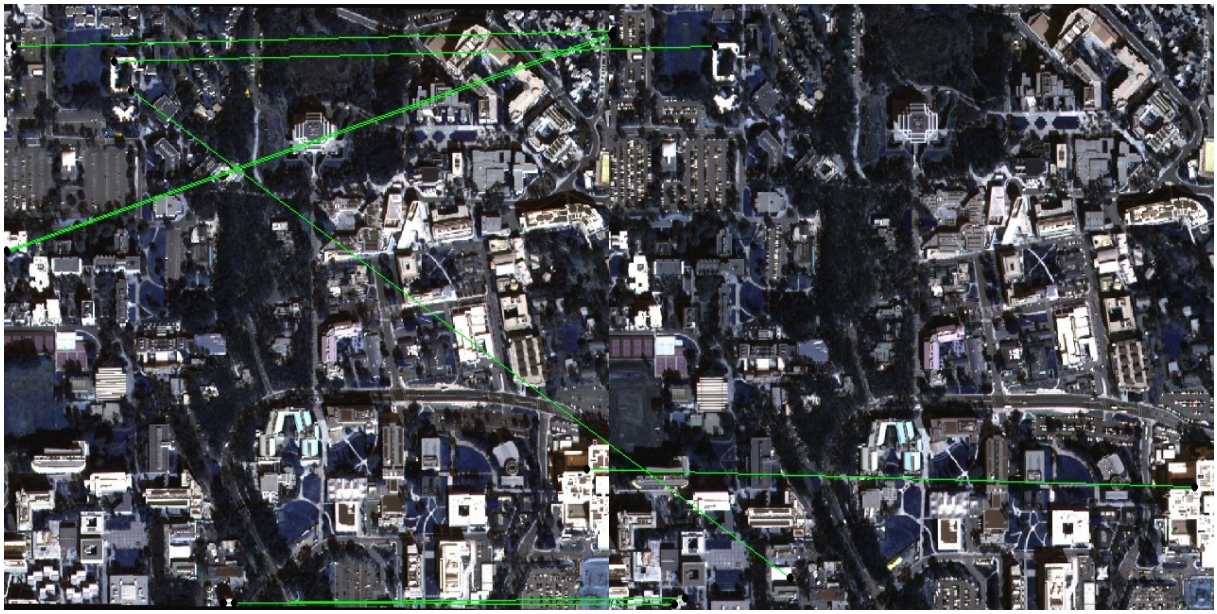

Figure 30: **SSD correspondence at sigma = 1.2**

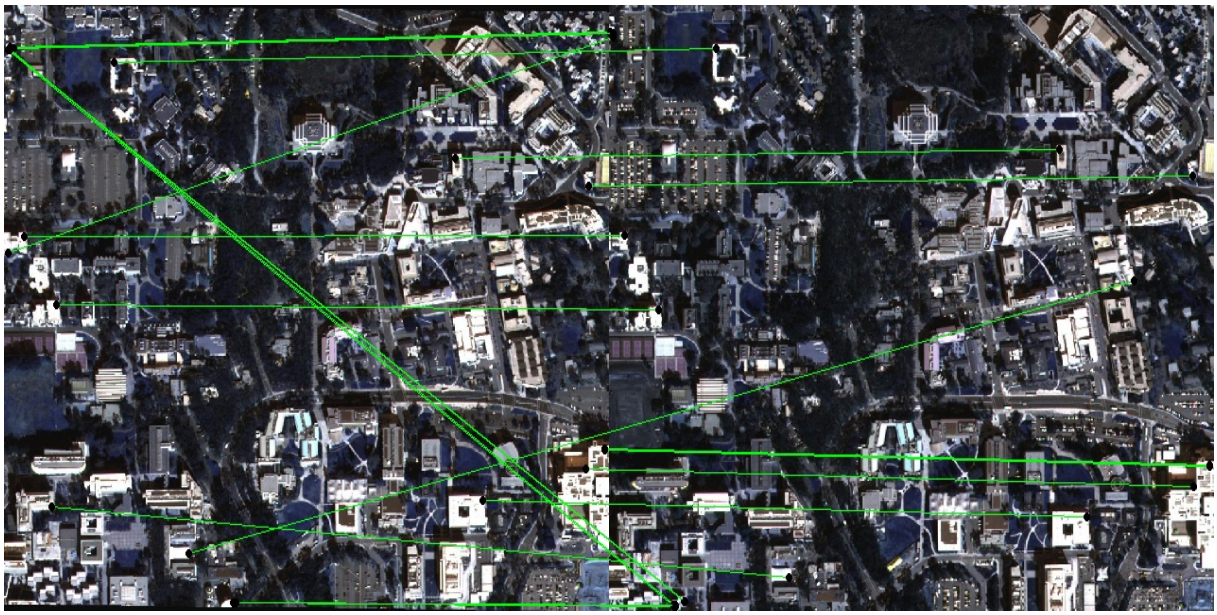

Figure 31: **SSD correspondence at sigma = 1.8**

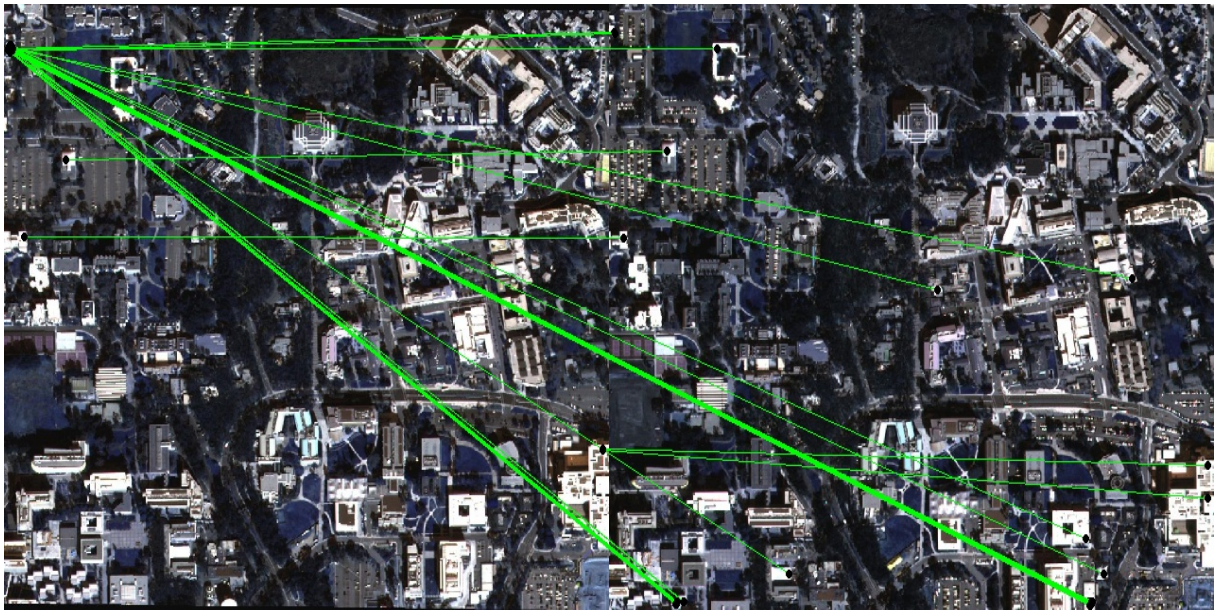

Figure 32: **SSD correspondence at sigma = 2.4**

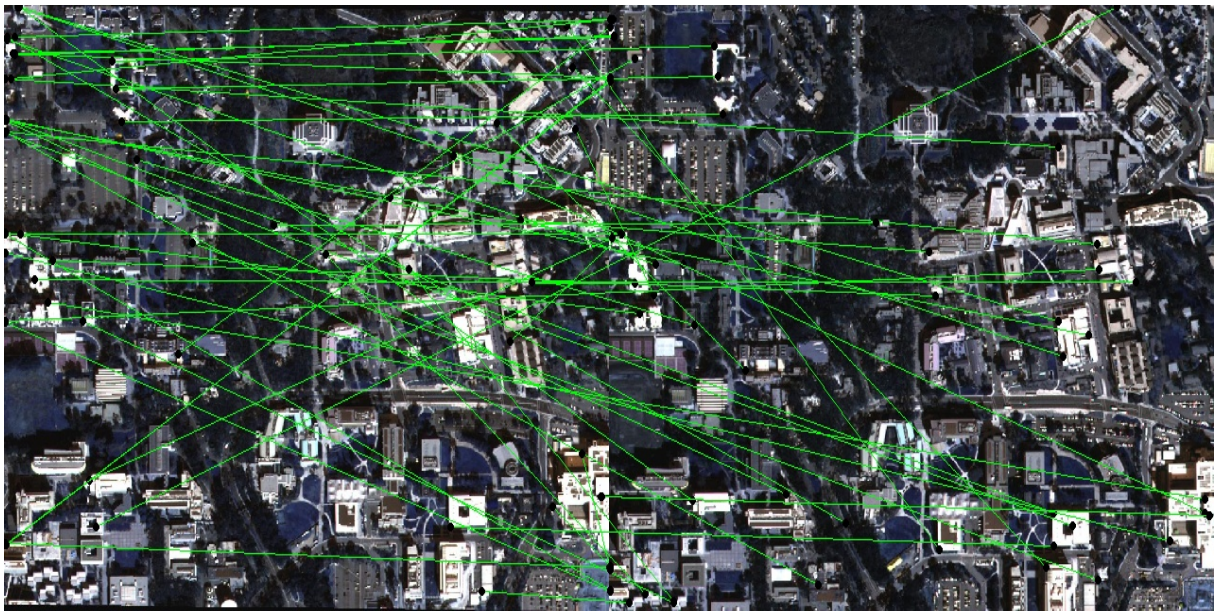

Figure 33: **NCC correspondence at sigma = 0.7**

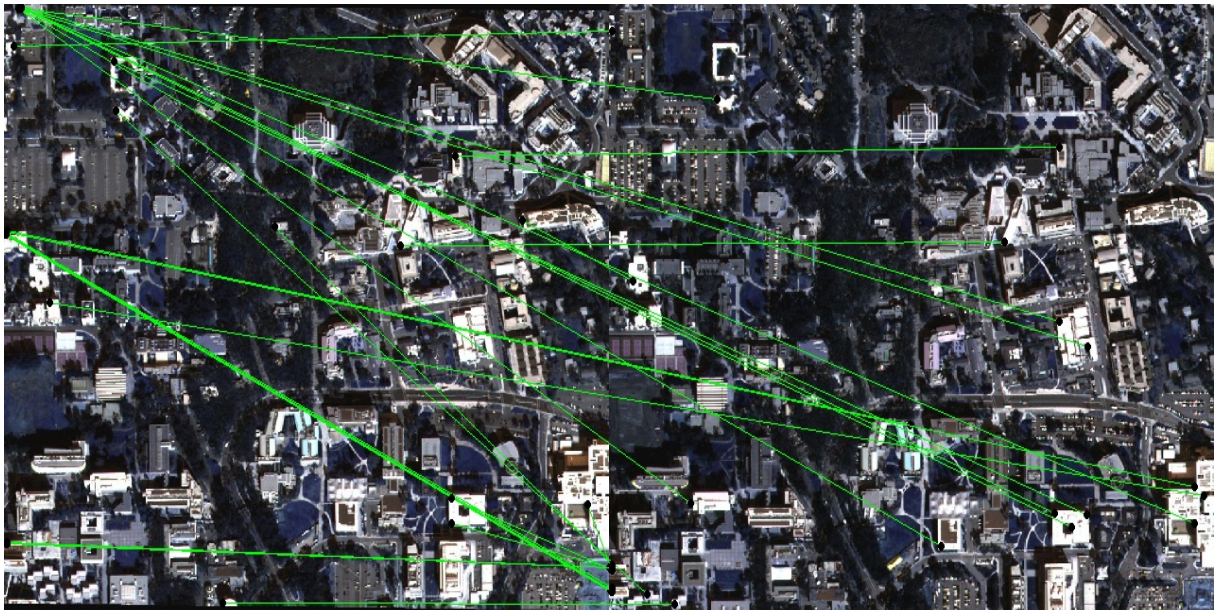

Figure 34: **NCC correspondence at sigma = 1.2**

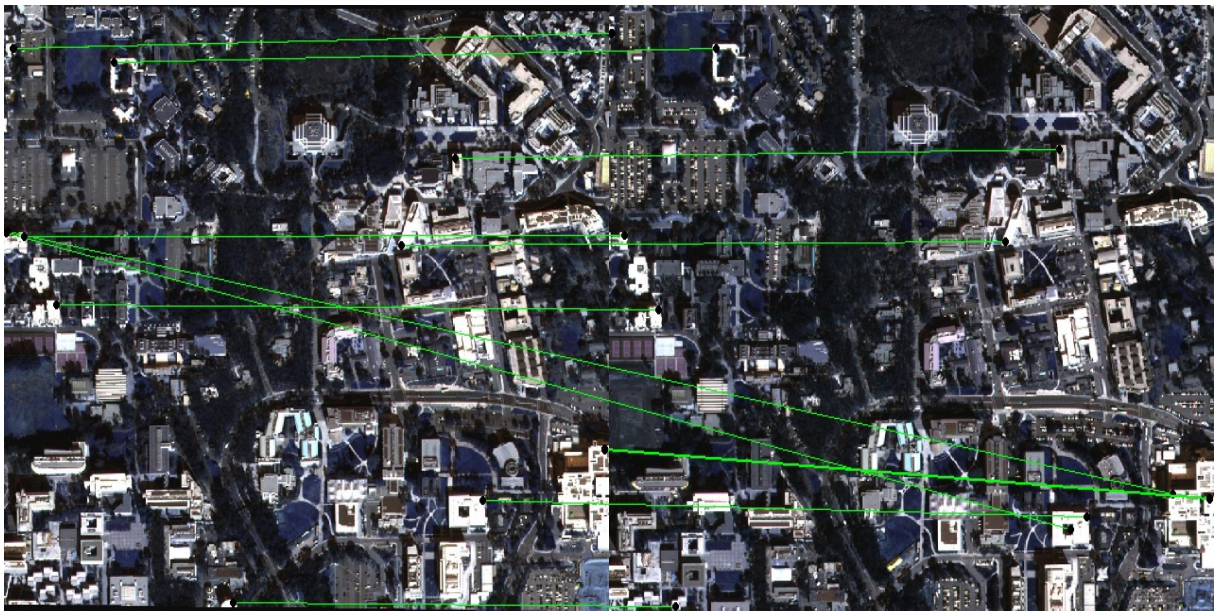

Figure 35: **NCC correspondence at sigma = 1.8**

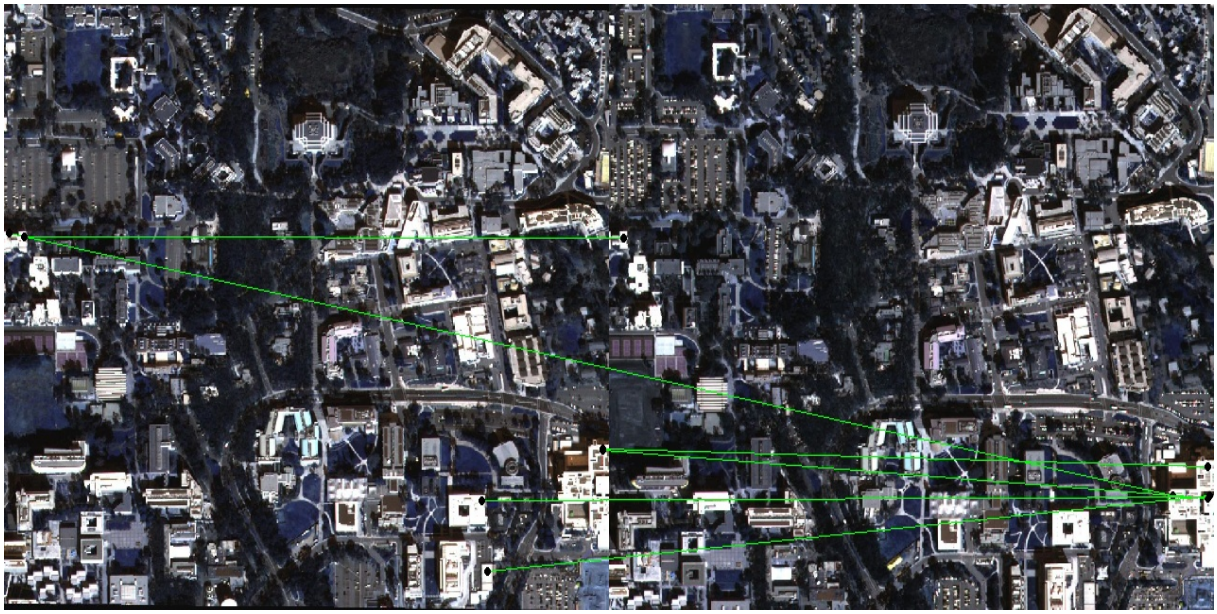

Figure 36: **NCC correspondence at sigma = 2.4**

Figure 37: **Sift corner detection**

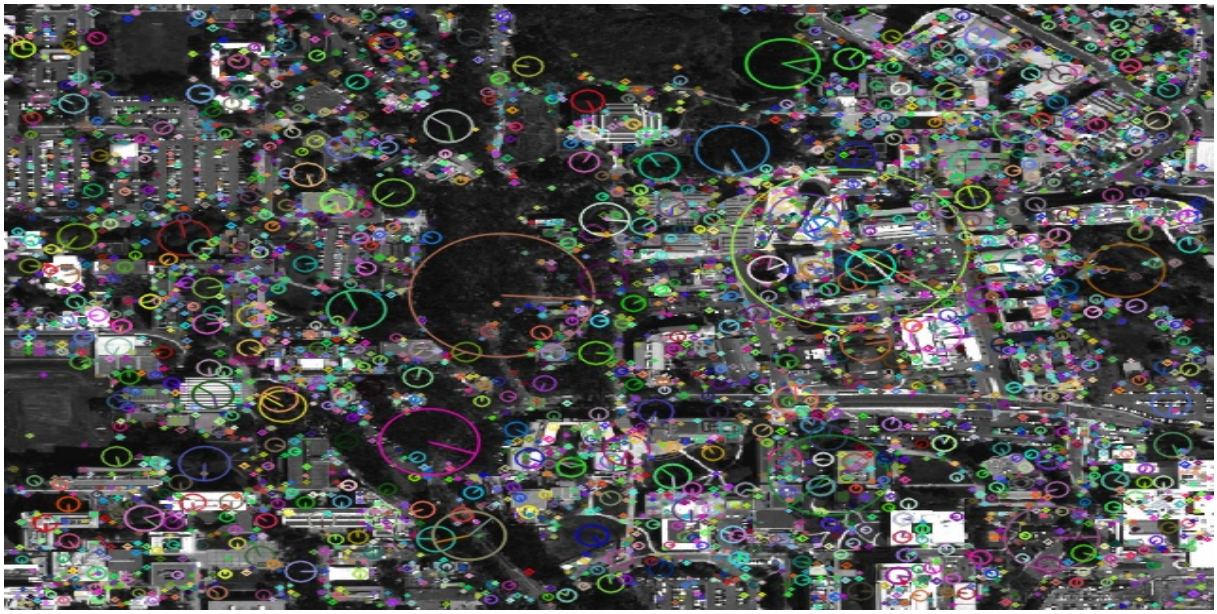

Figure 38: **Sift corner detection**

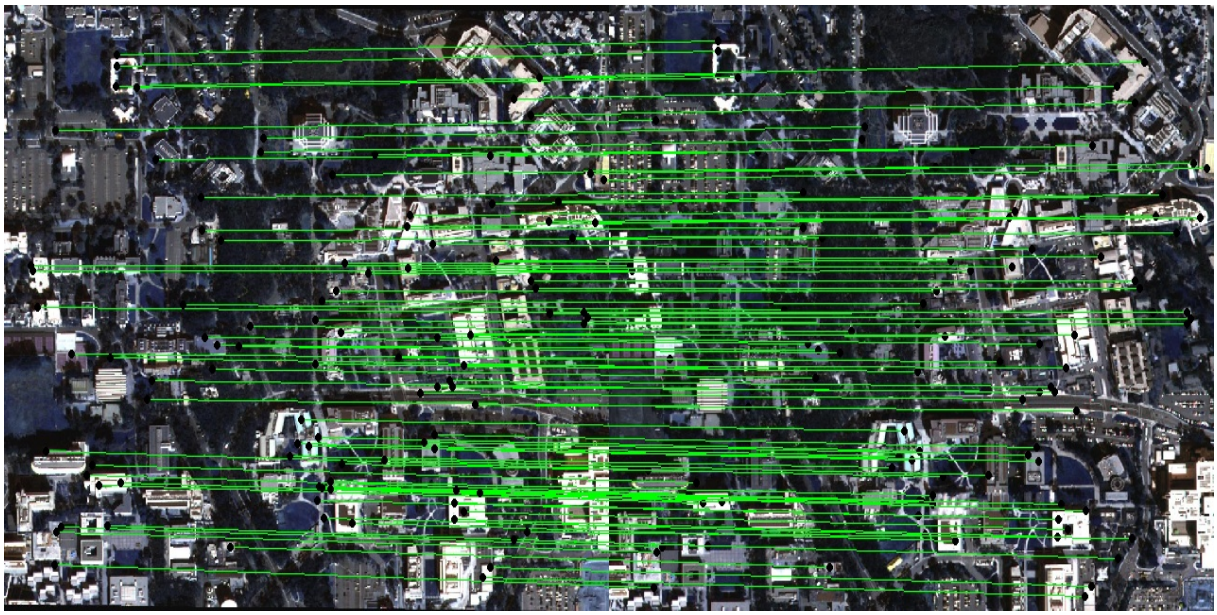

Figure 39: **Eucledian matching for SIFT**

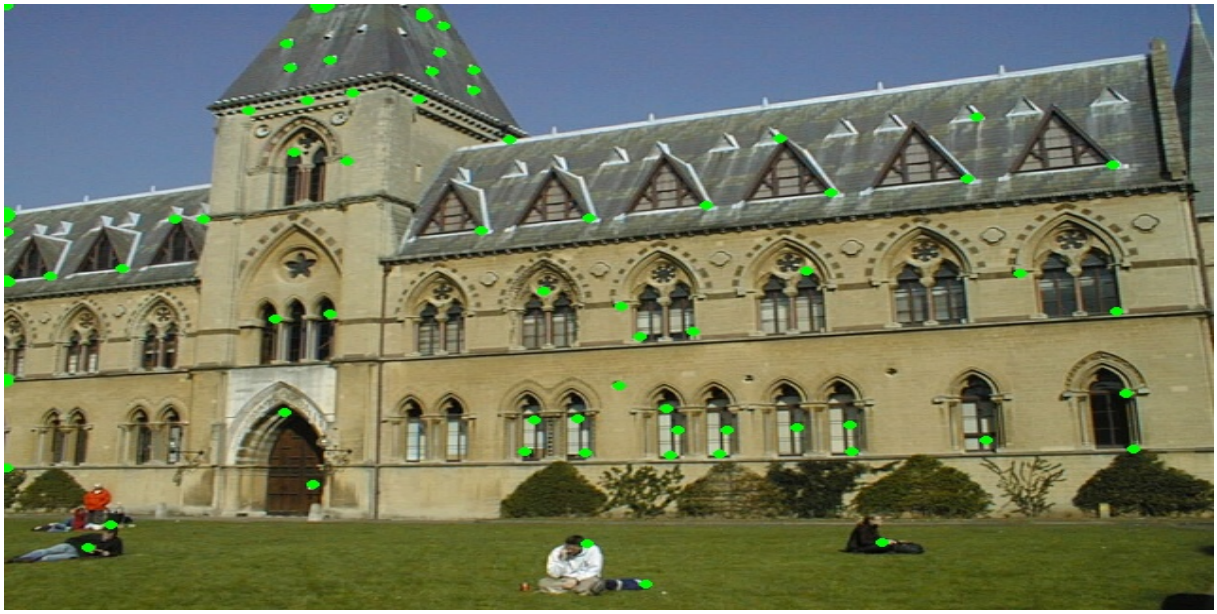

Figure 40: **Harris corner detection with sigma = 0.7**

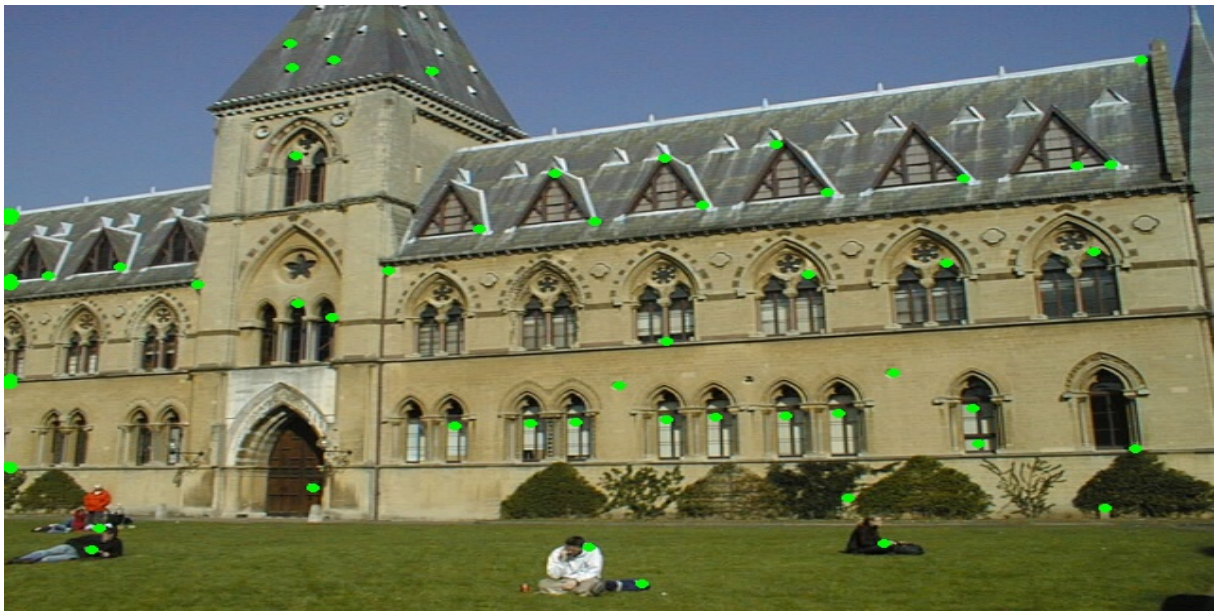

Figure 41: **Harris corner detection with sigma = 1.2**

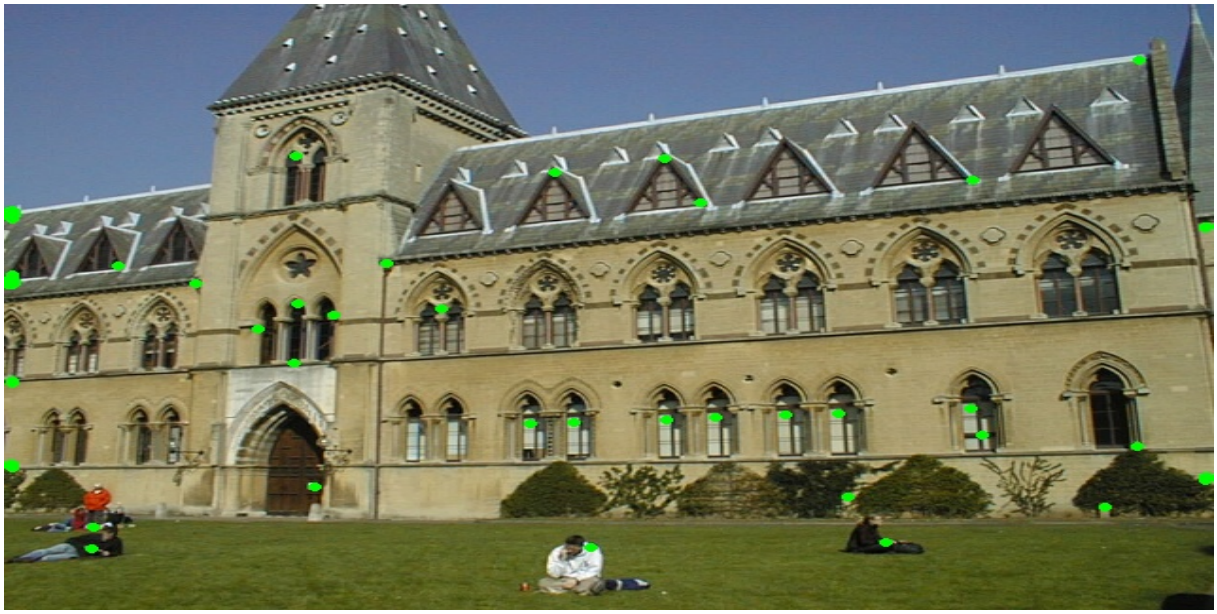

Figure 42: **Harris corner detection with sigma = 1.8**

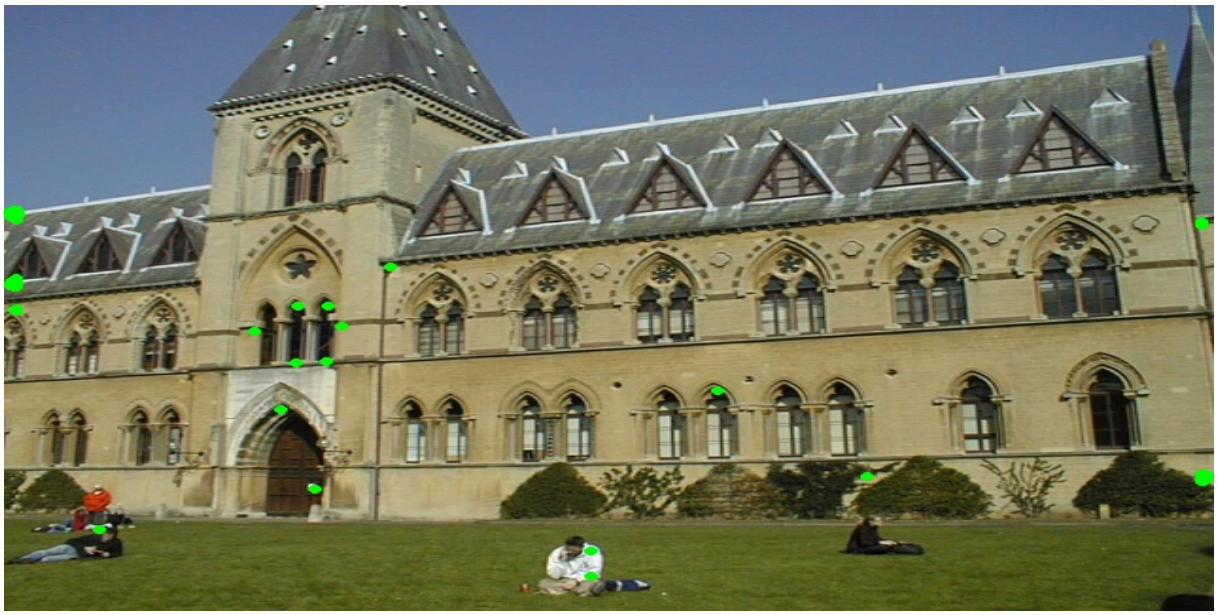

Figure 43: **Harris corner detection with sigma = 2.4**

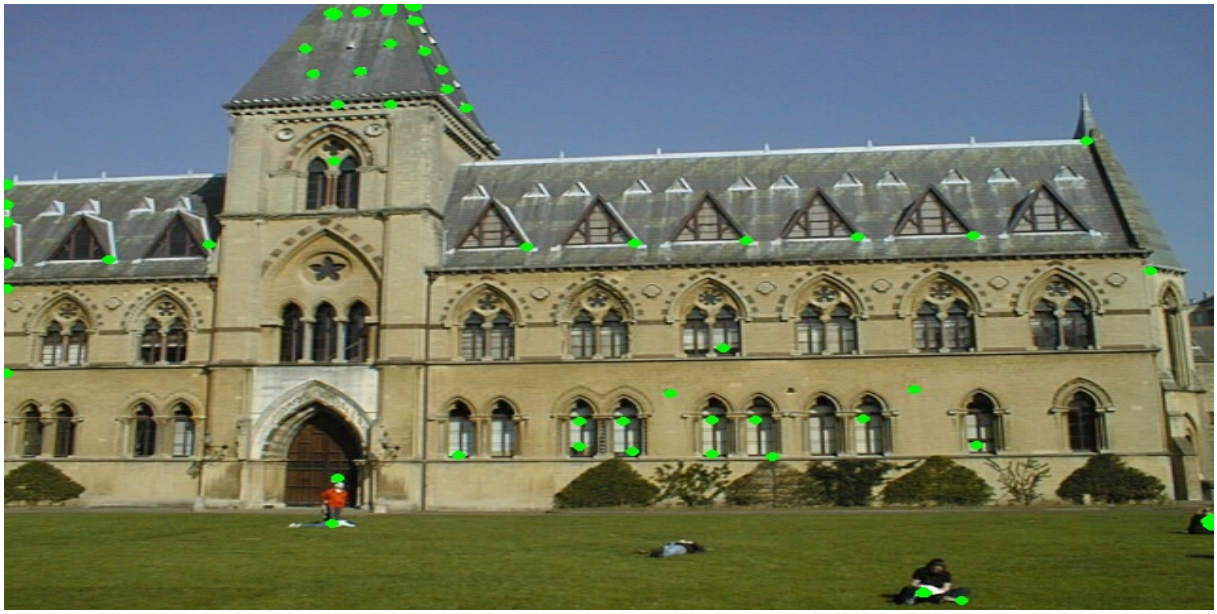

Figure 44: **Harris corner detection with sigma = 0.7**

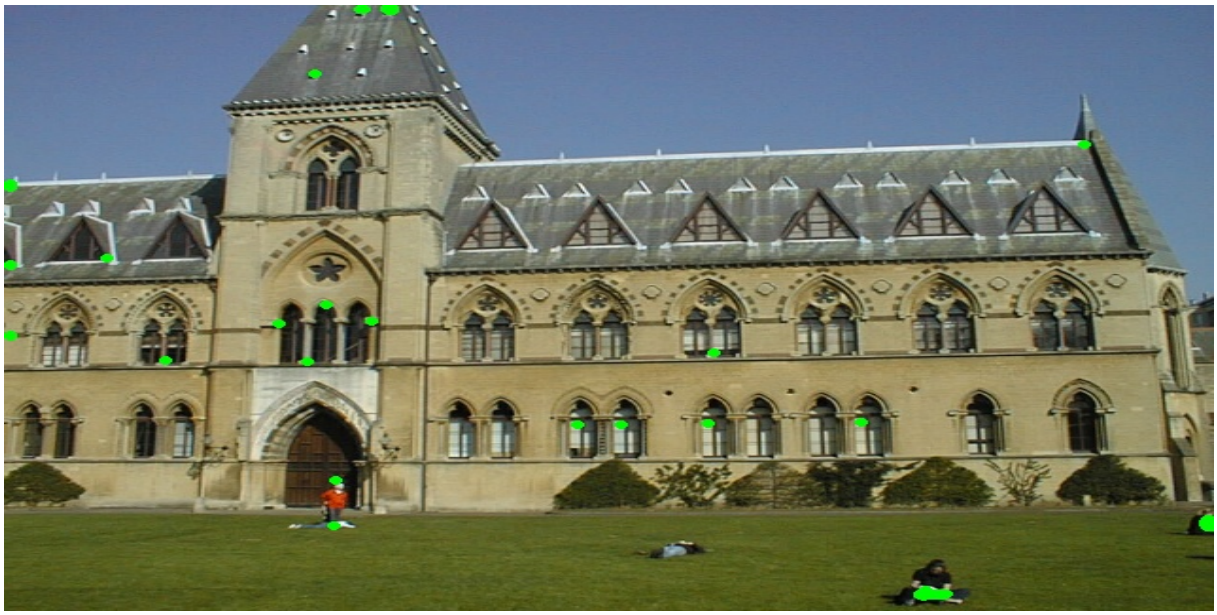

Figure 45: **Harris corner detection with sigma = 1.2**

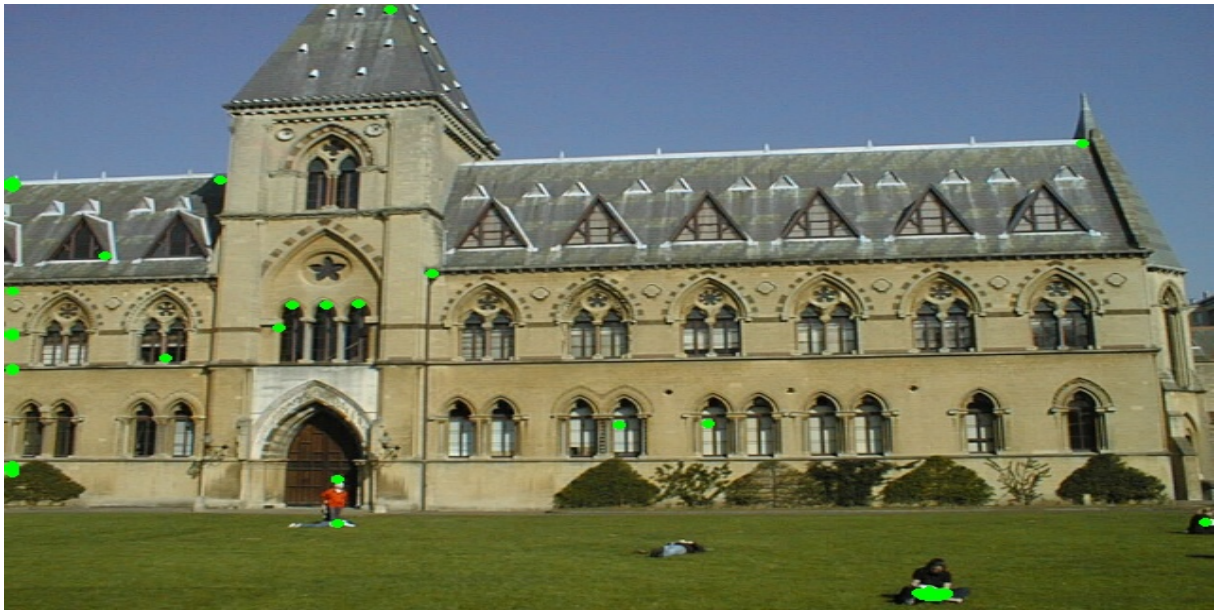

Figure 46: **Harris corner detection with sigma = 1.8**

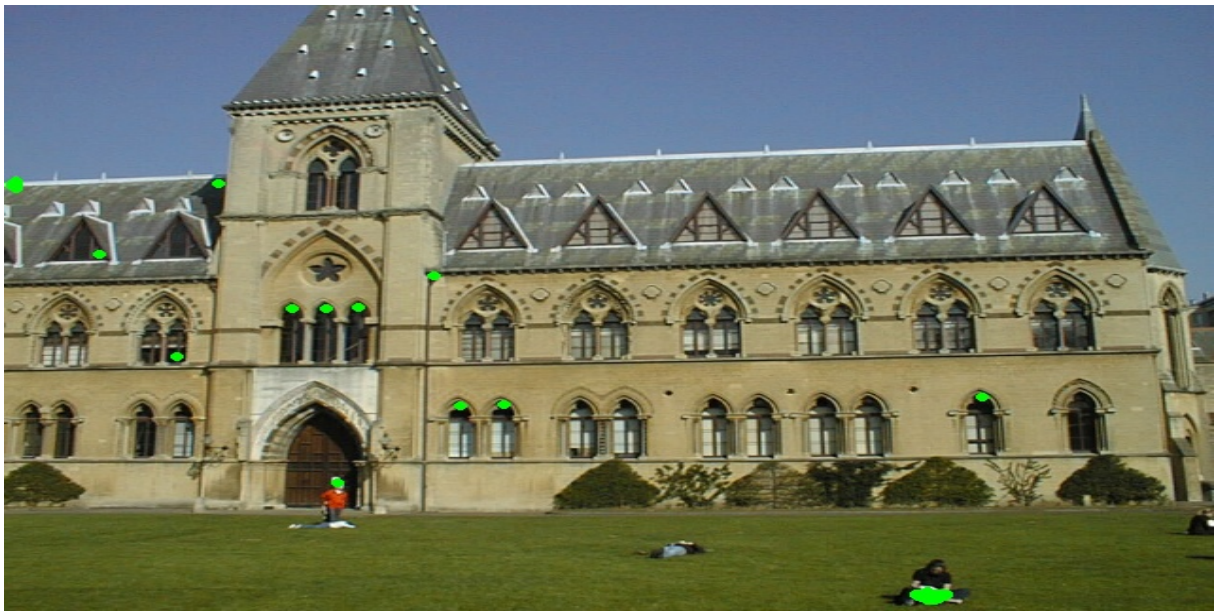

Figure 47: **Harris corner detection with sigma = 2.4**

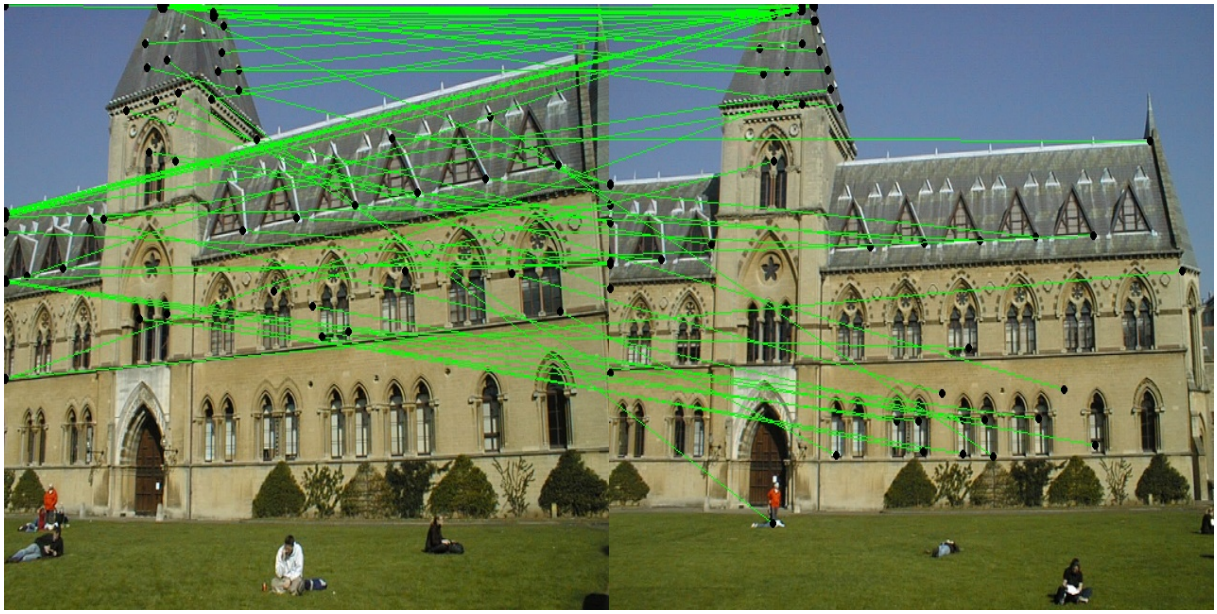

Figure 48: **SSD correspondence at sigma = 0.7**

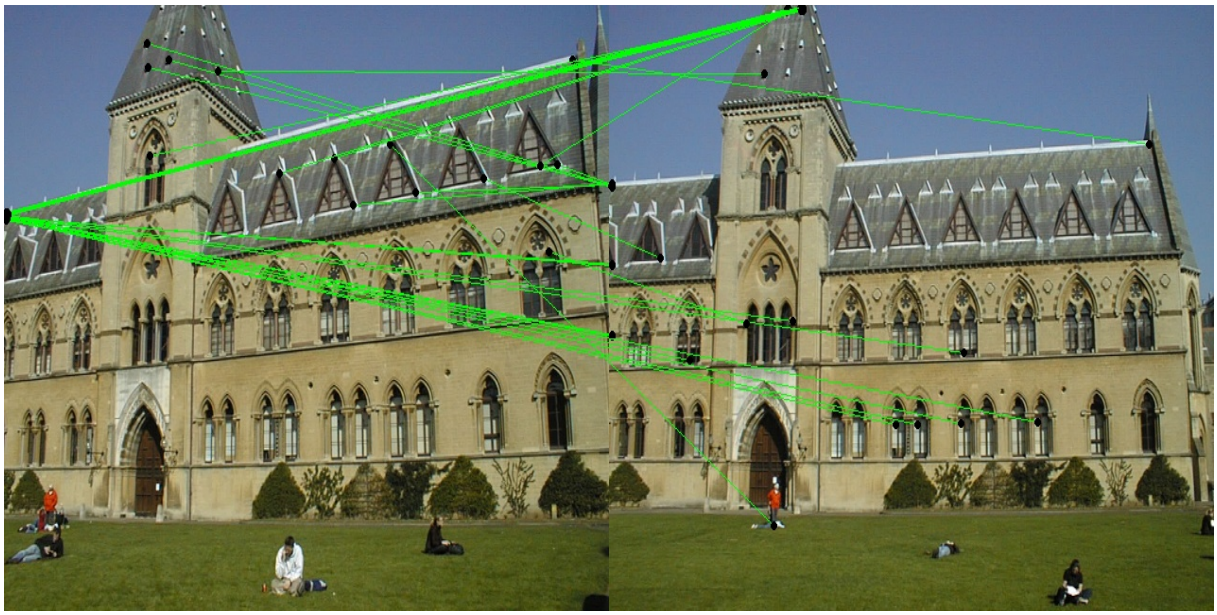

Figure 49: **SSD correspondence at sigma = 1.2**

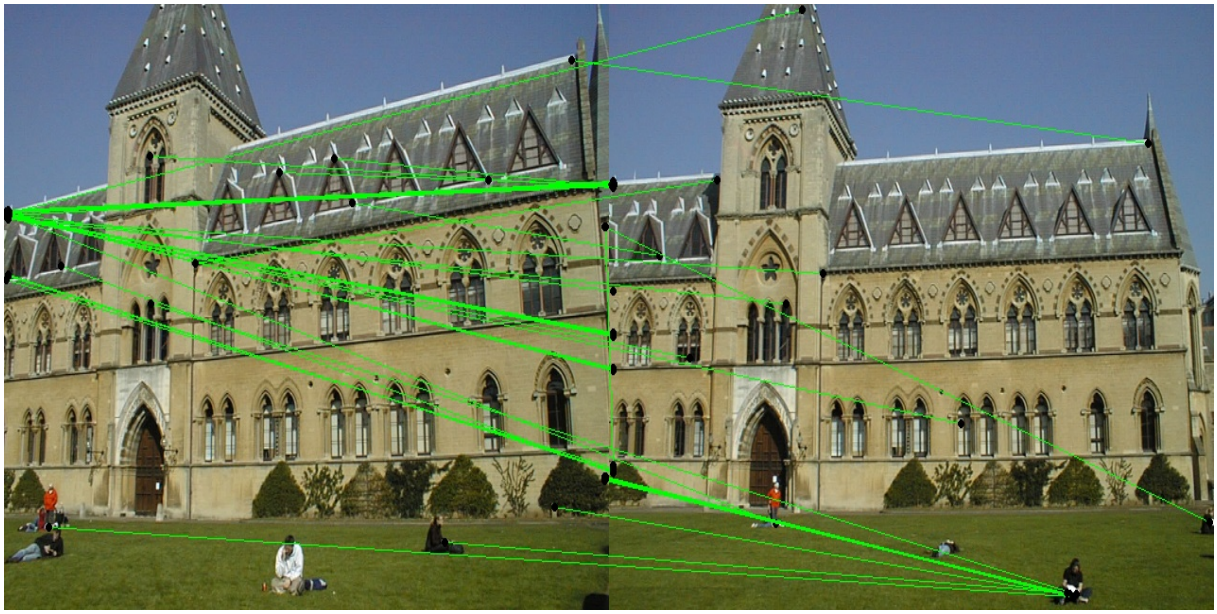

Figure 50: **SSD correspondence at sigma = 1.8**

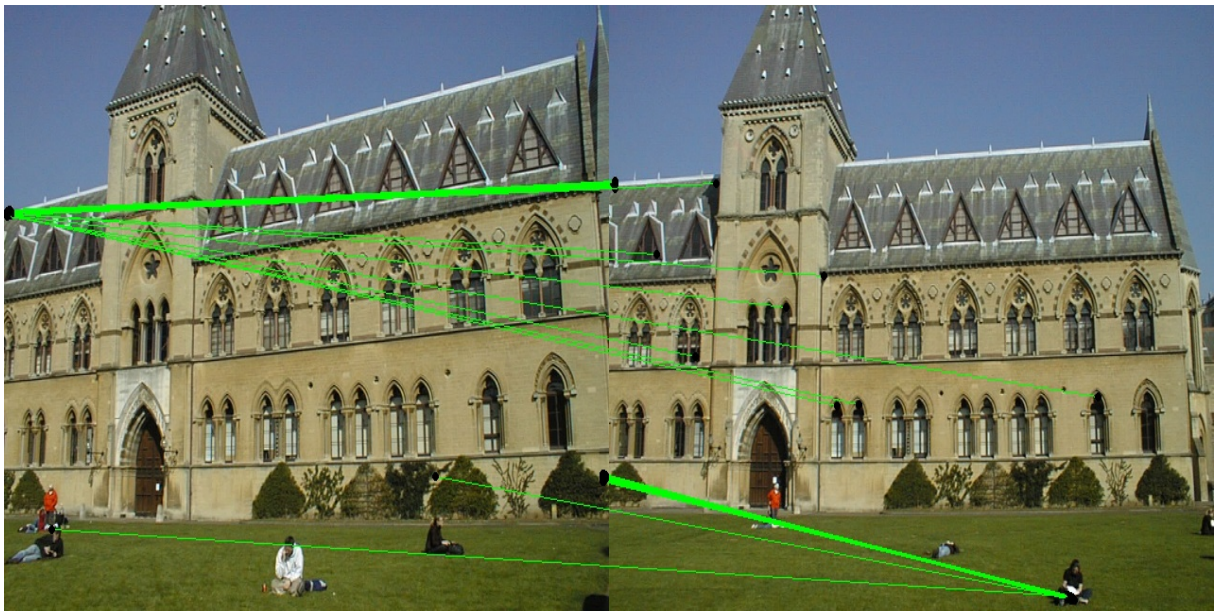

Figure 51: **SSD correspondence at sigma = 2.4**

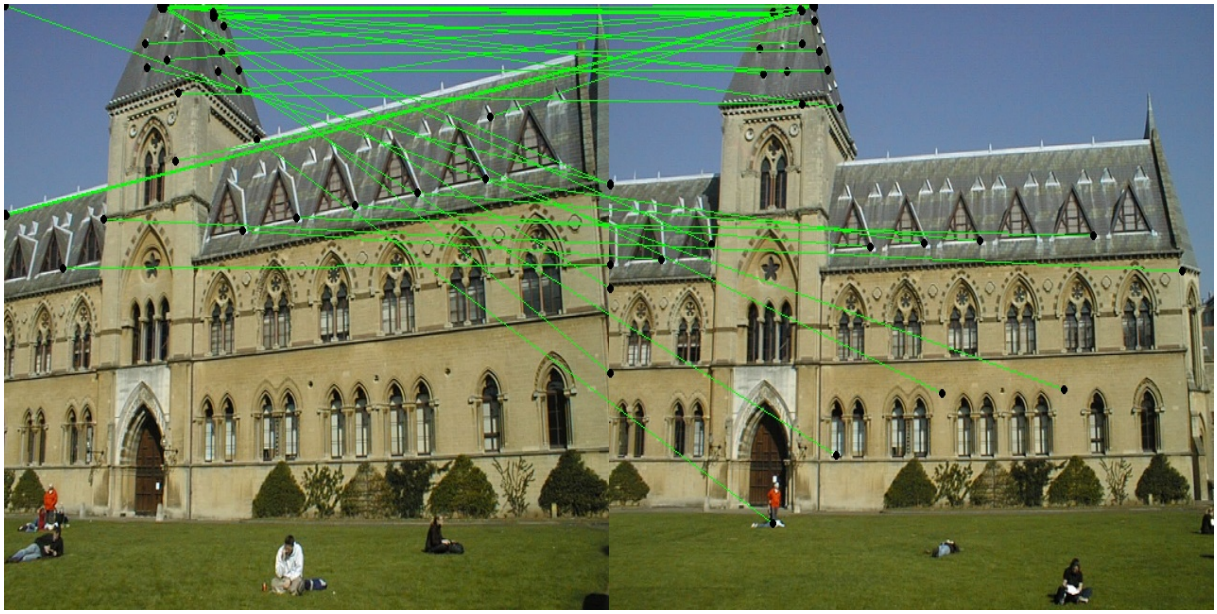

Figure 52: **NCC correspondence at sigma = 0.7**

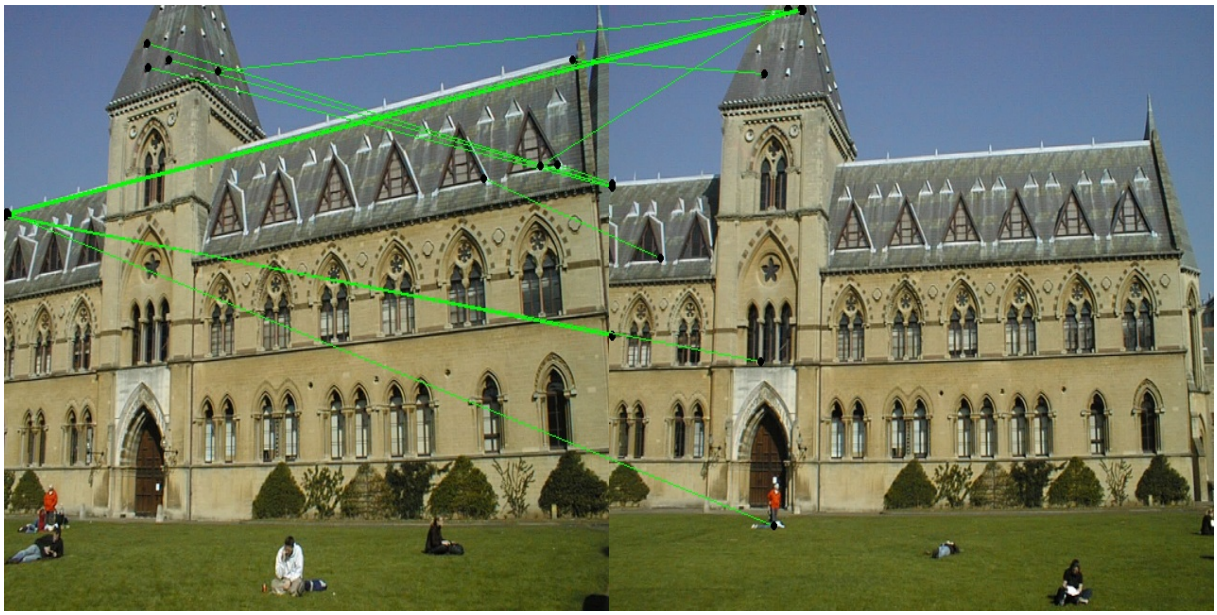

Figure 53: **NCC correspondence at sigma = 1.2**

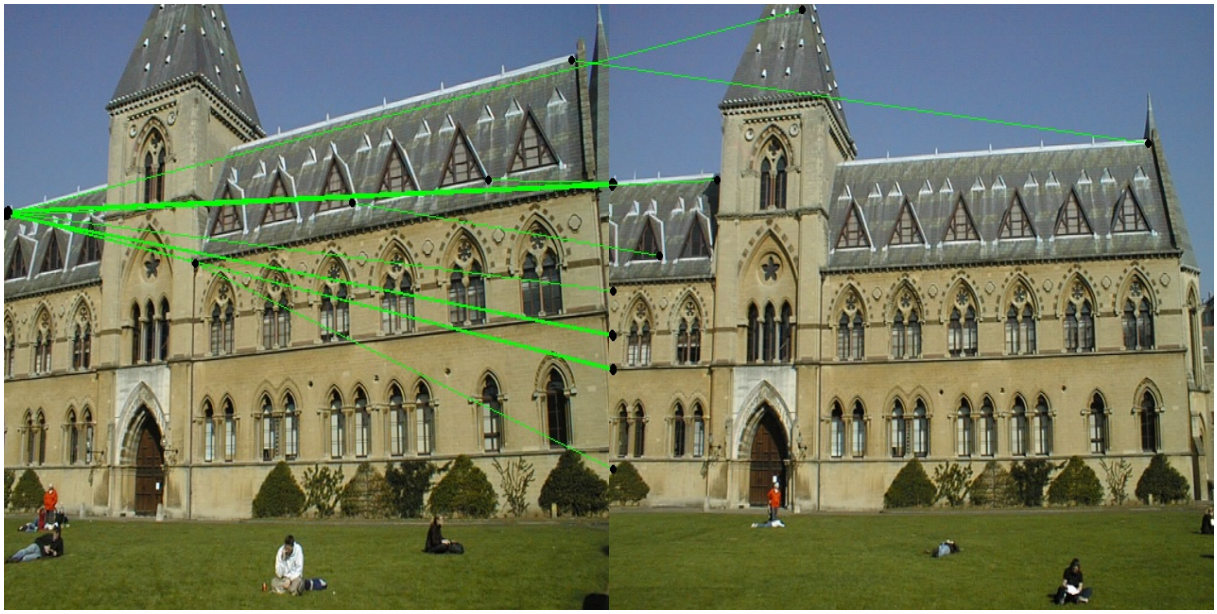

Figure 54: **NCC correspondence at sigma = 1.8**

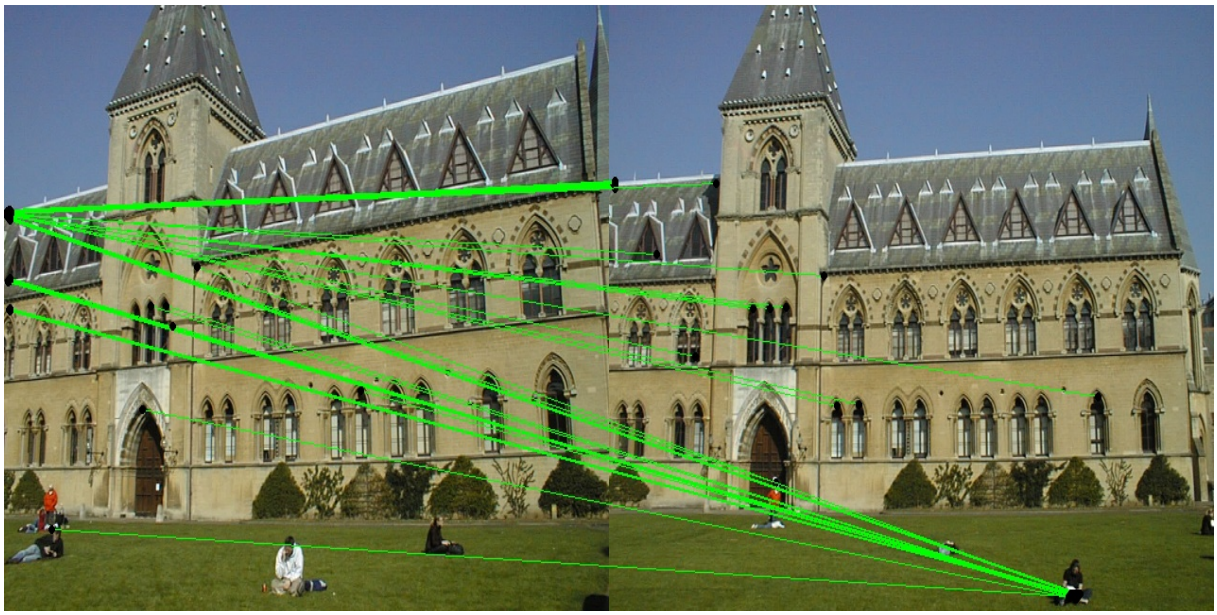

Figure 55: **NCC correspondence at sigma = 2.4**

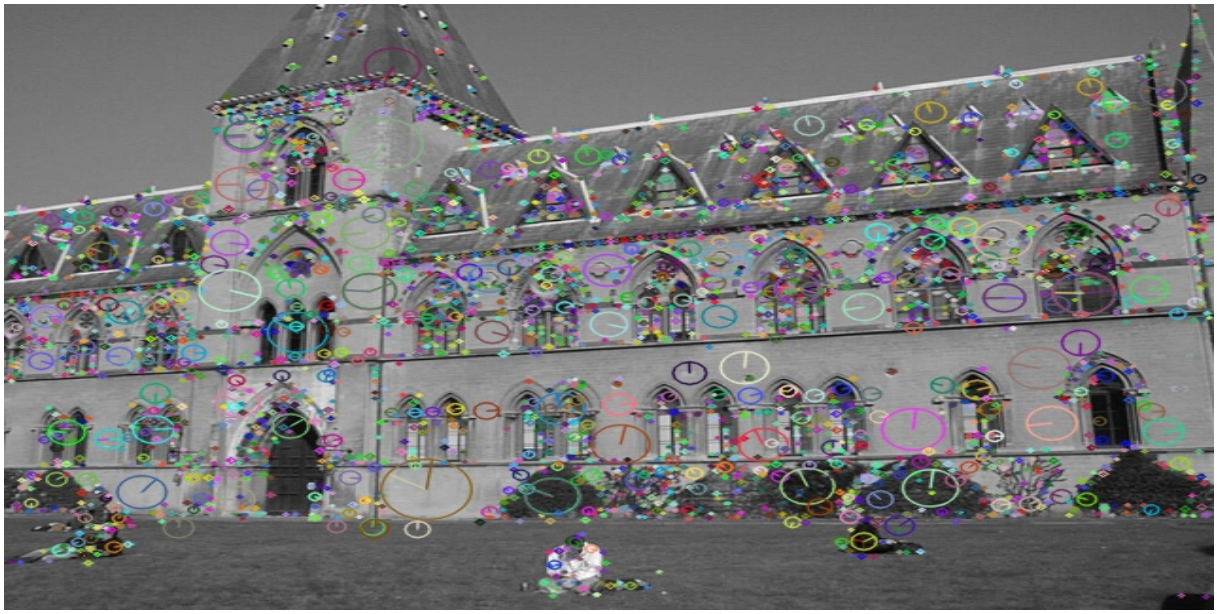

Figure 56: **Sift corner detection**

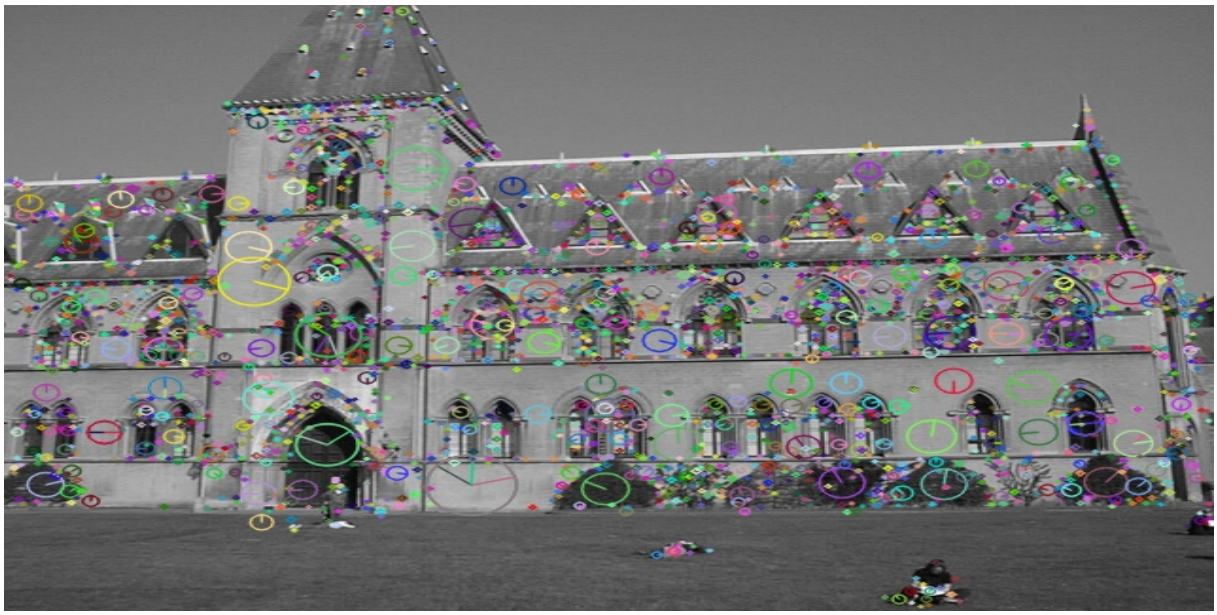

Figure 57: **Sift corner detection**

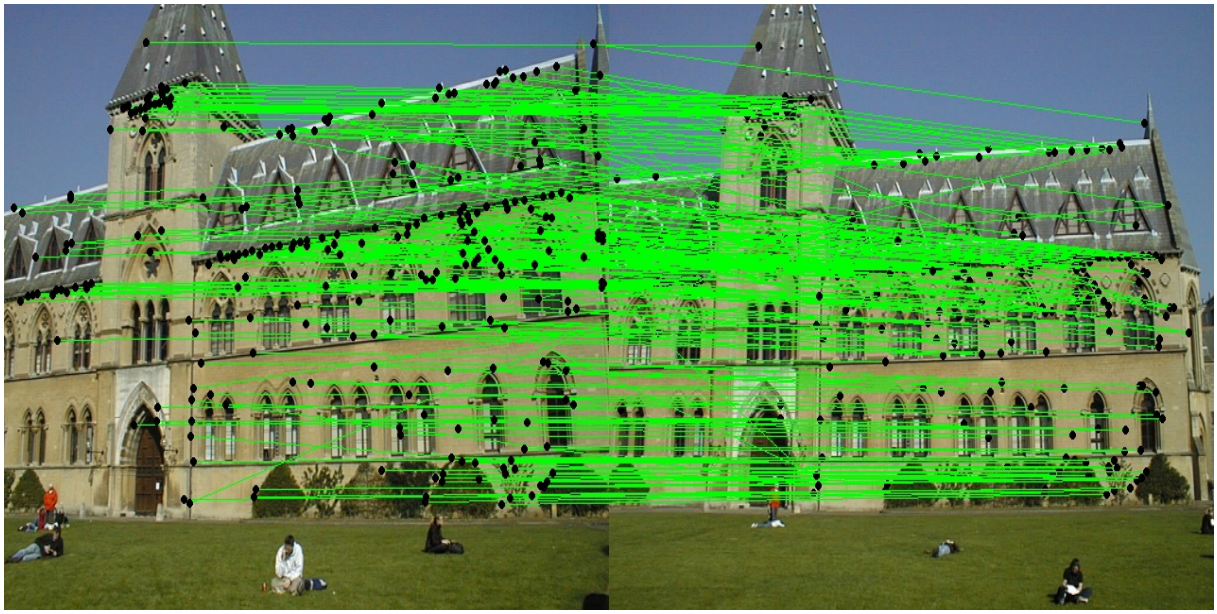

Figure 58: **Eucledian matching for SIFT**

Figure 59: **Harris corner detection with sigma = 0.7**

Figure 60: **Harris corner detection with sigma = 1.2**

Figure 61: **Harris corner detection with sigma = 1.8**

Figure 62: **Harris corner detection with sigma = 2.4**

Figure 63: **Harris corner detection with sigma = 0.7**

Figure 64: **Harris corner detection with sigma = 1.2**

Figure 65: **Harris corner detection with sigma = 1.8**

Figure 66: **Harris corner detection with sigma = 2.4**

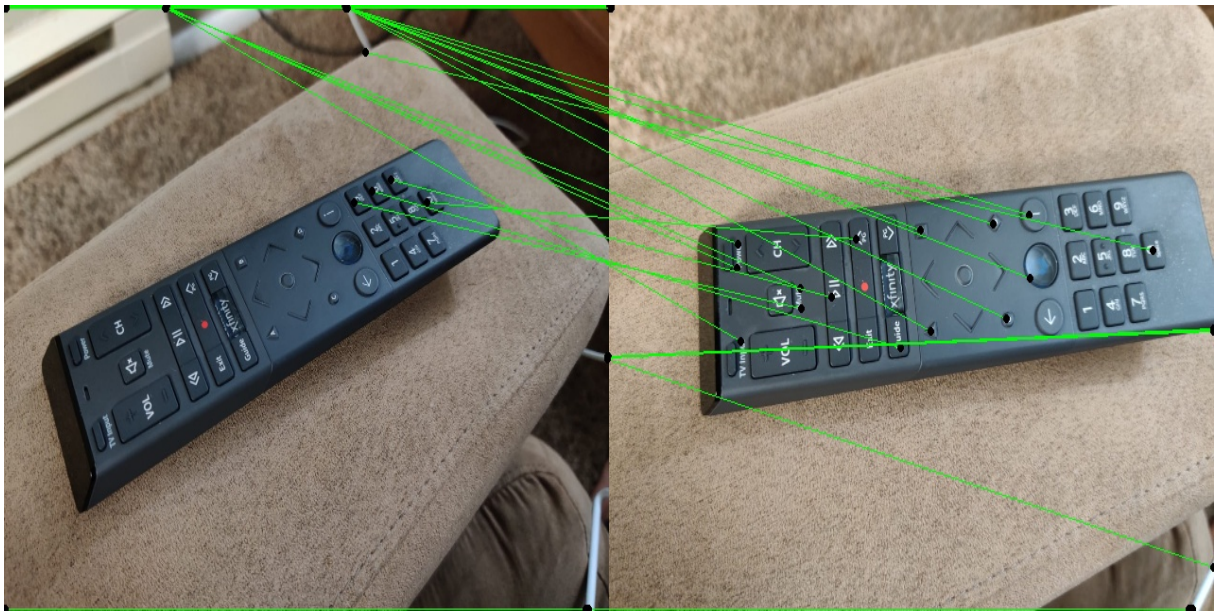

Figure 67: **SSD correspondence at sigma = 0.7**

Figure 68: **SSD correspondence at sigma = 1.2**

Figure 69: **SSD correspondence at sigma = 1.8**

Figure 70: **SSD correspondence at sigma = 2.4**

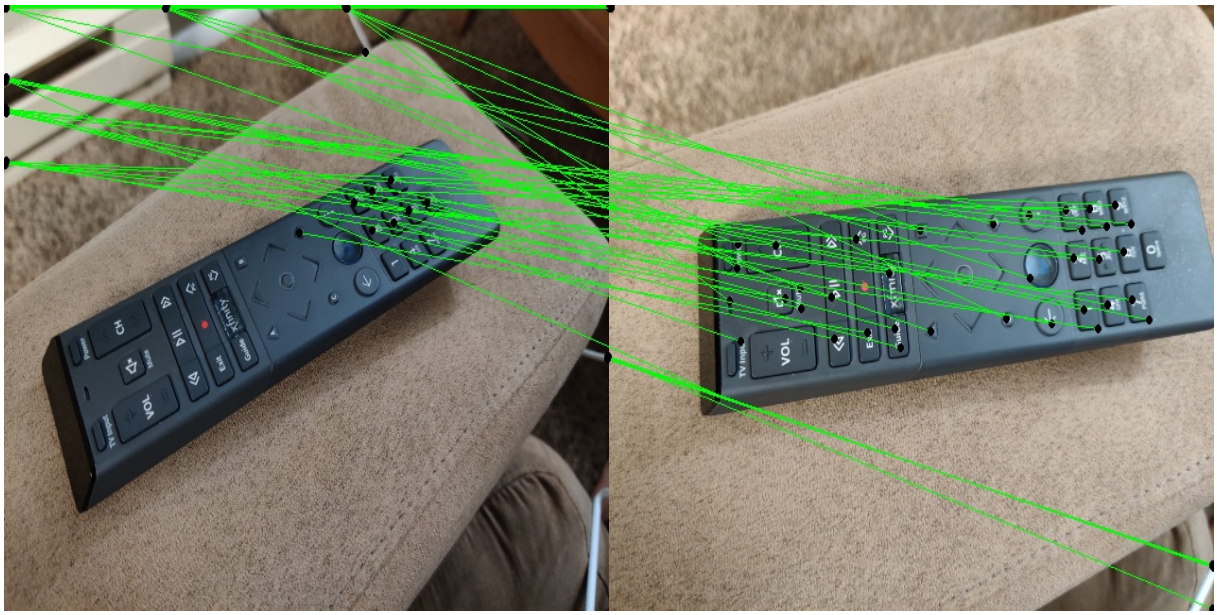

Figure 71: **NCC correspondence at sigma = 0.7**

Figure 72: **NCC correspondence at sigma = 1.2**

Figure 73: **NCC correspondence at sigma = 1.8**

Figure 74: **NCC correspondence at sigma = 2.4**

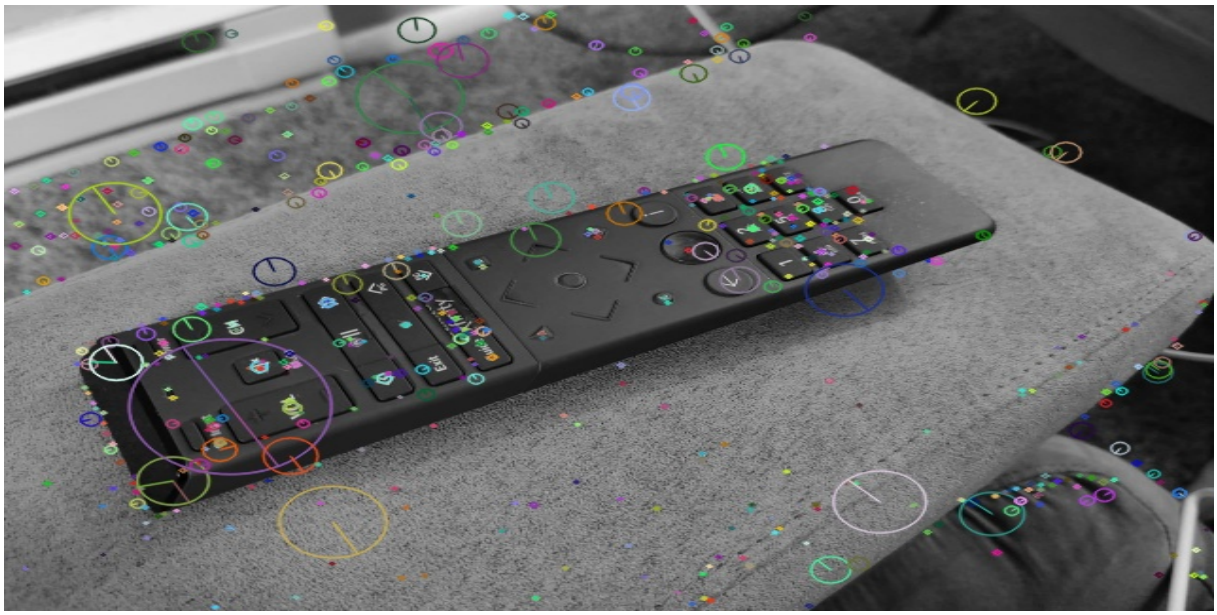

Figure 75: **Sift corner detection**

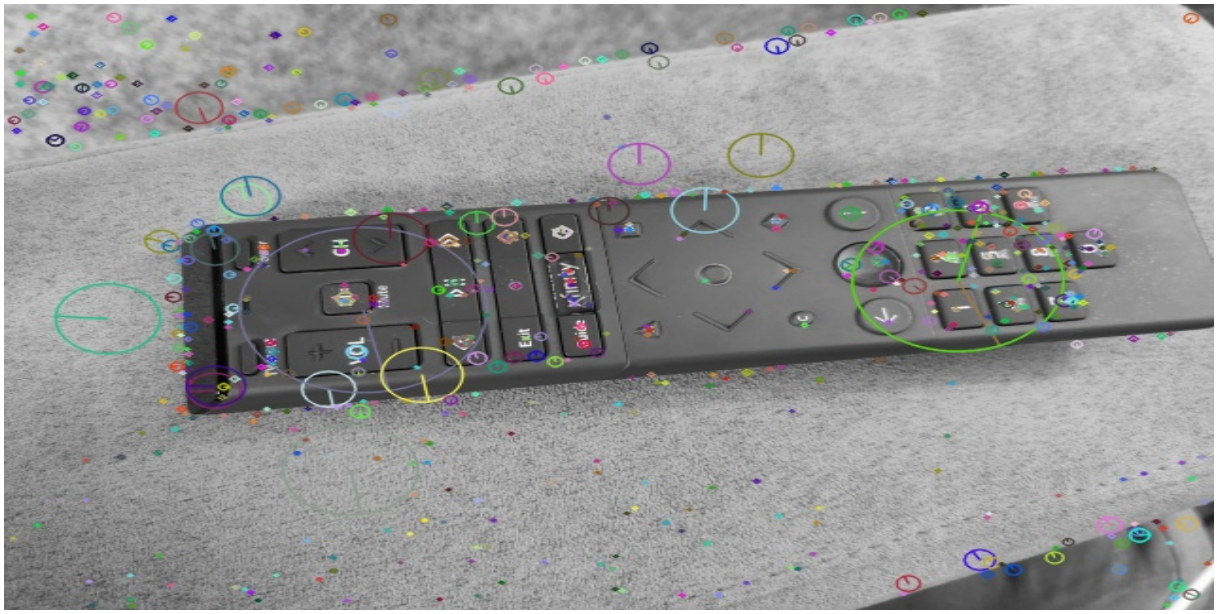

Figure 76: **Sift corner detection**

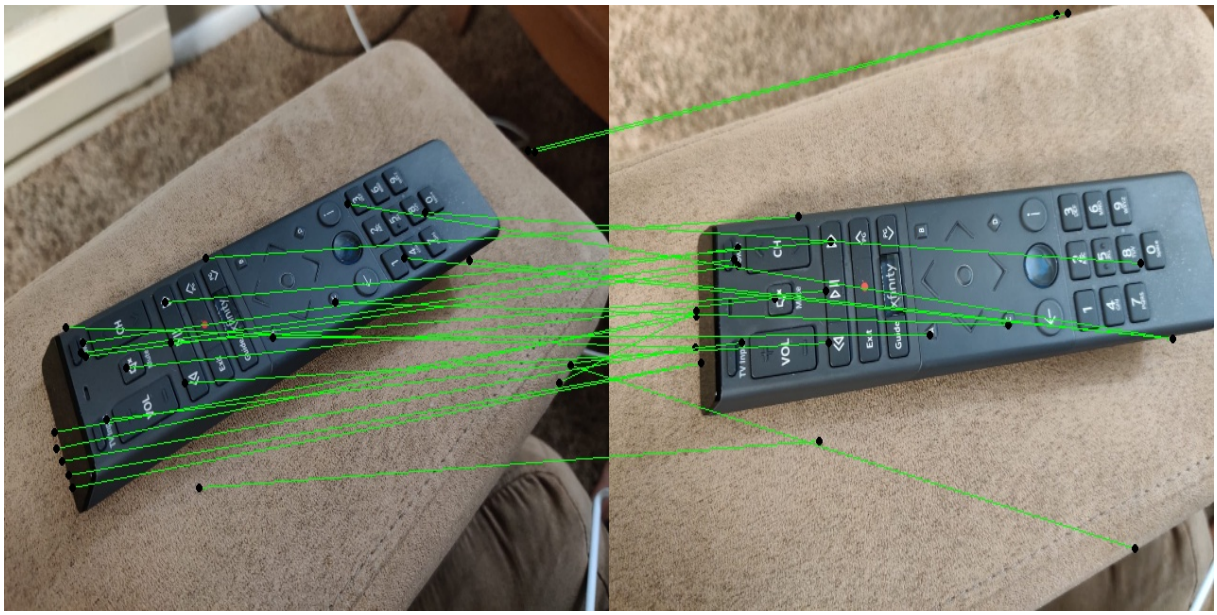

Figure 77: **Eucledian matching for SIFT**

Figure 78: **Harris corner detection with sigma = 0.7**

Figure 79: **Harris corner detection with sigma = 1.2**

Figure 80: **Harris corner detection with sigma = 1.8**

Figure 81: **Harris corner detection with sigma = 2.4**

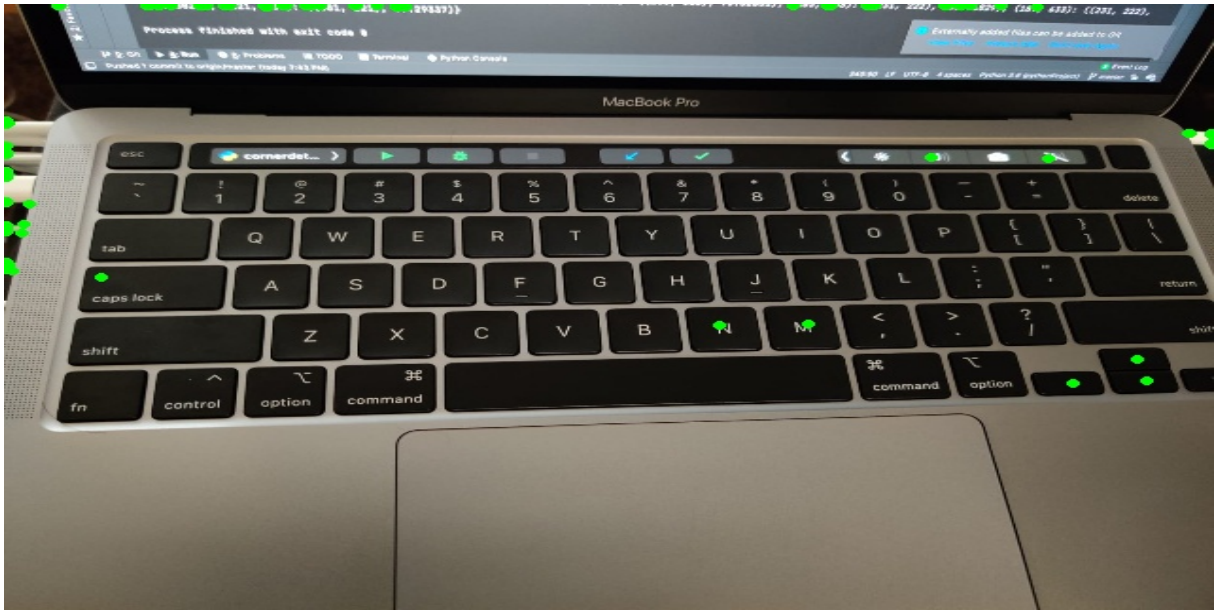

Figure 82: **Harris corner detection with sigma = 0.7**

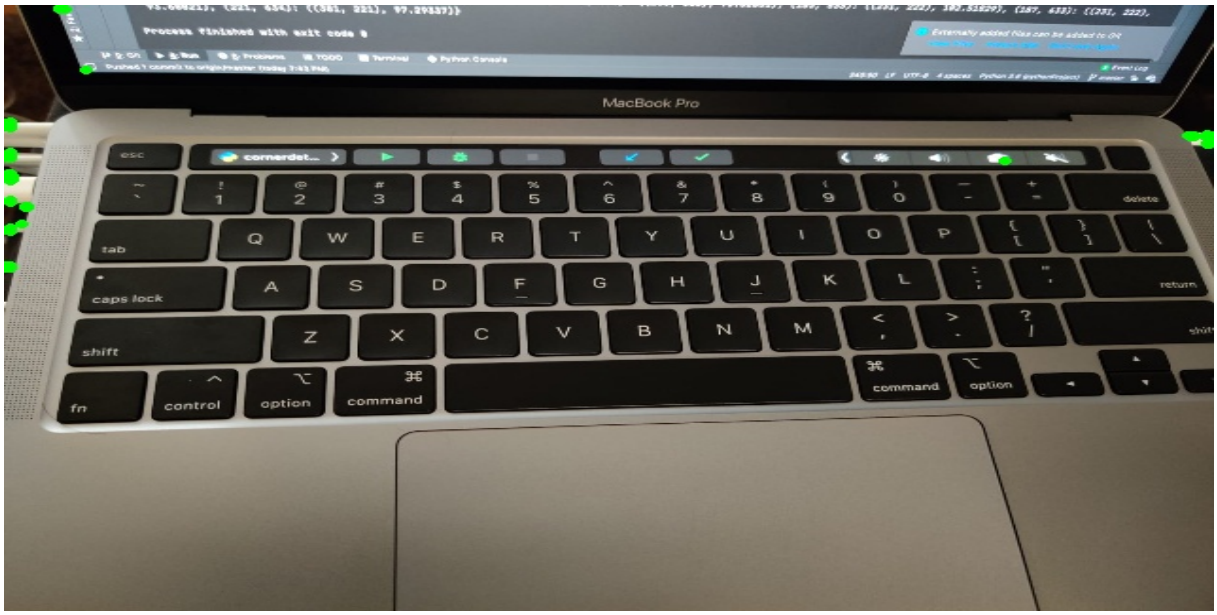

Figure 83: **Harris corner detection with sigma = 1.2**

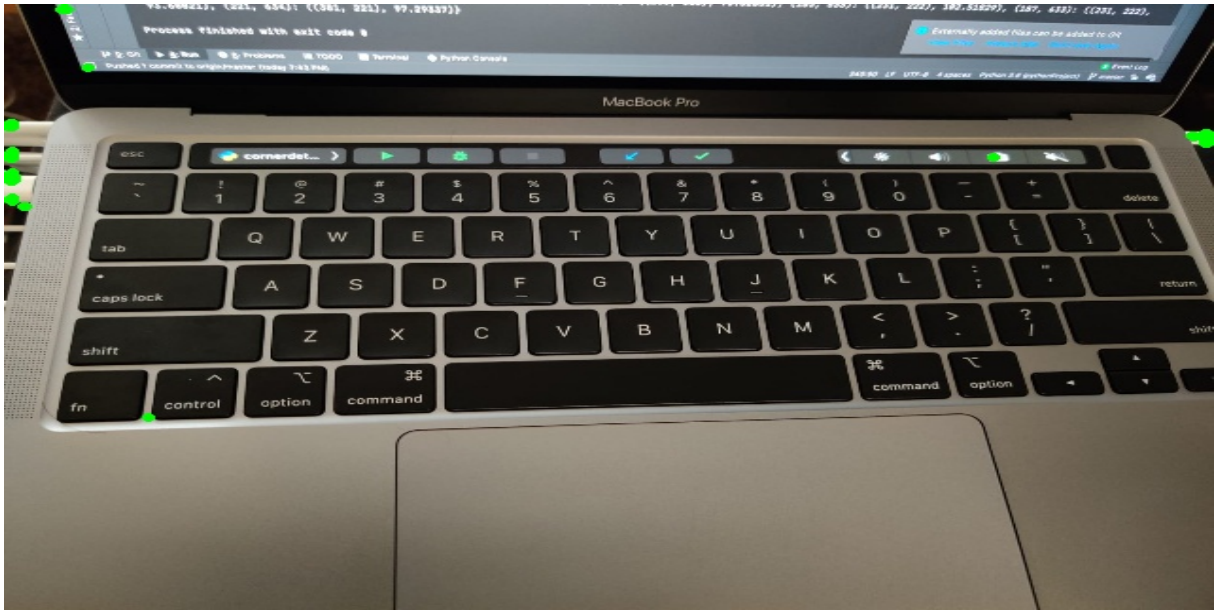

Figure 84: **Harris corner detection with sigma = 1.8**

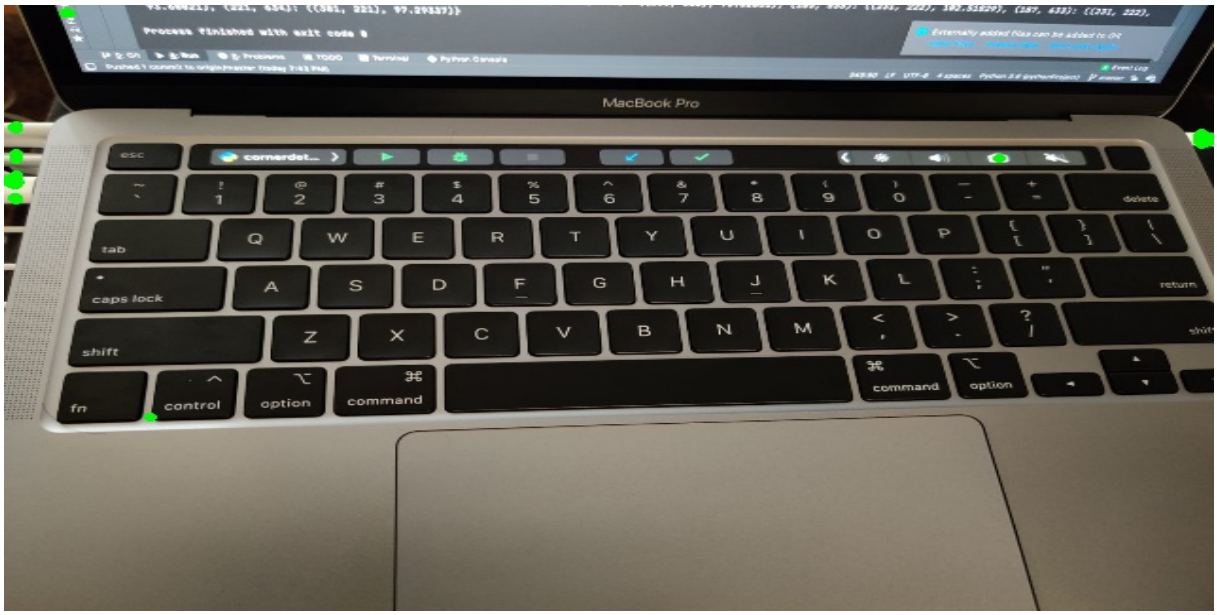

Figure 85: **Harris corner detection with sigma = 2.4**

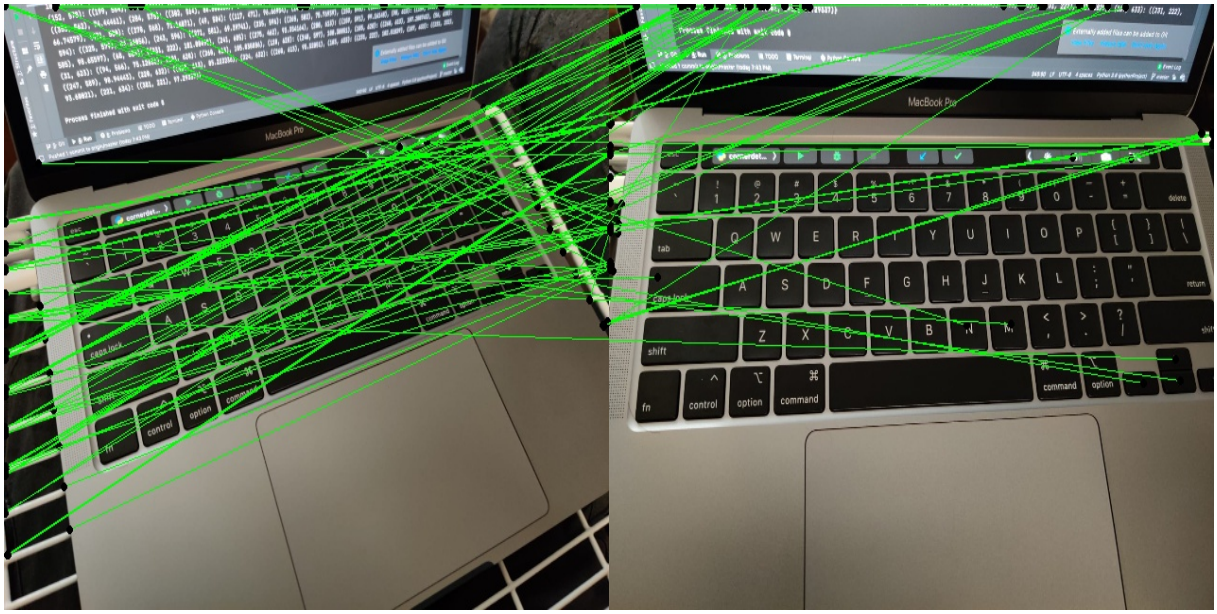

Figure 86: **SSD correspondence at sigma = 0.7**

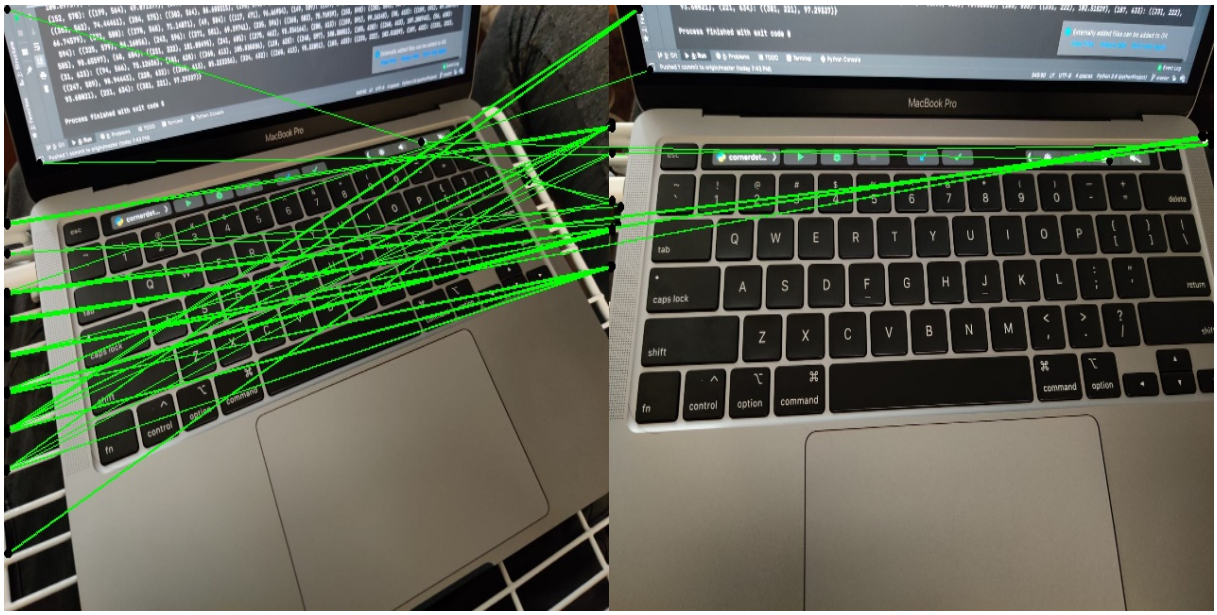

Figure 87: **SSD correspondence at sigma = 1.2**

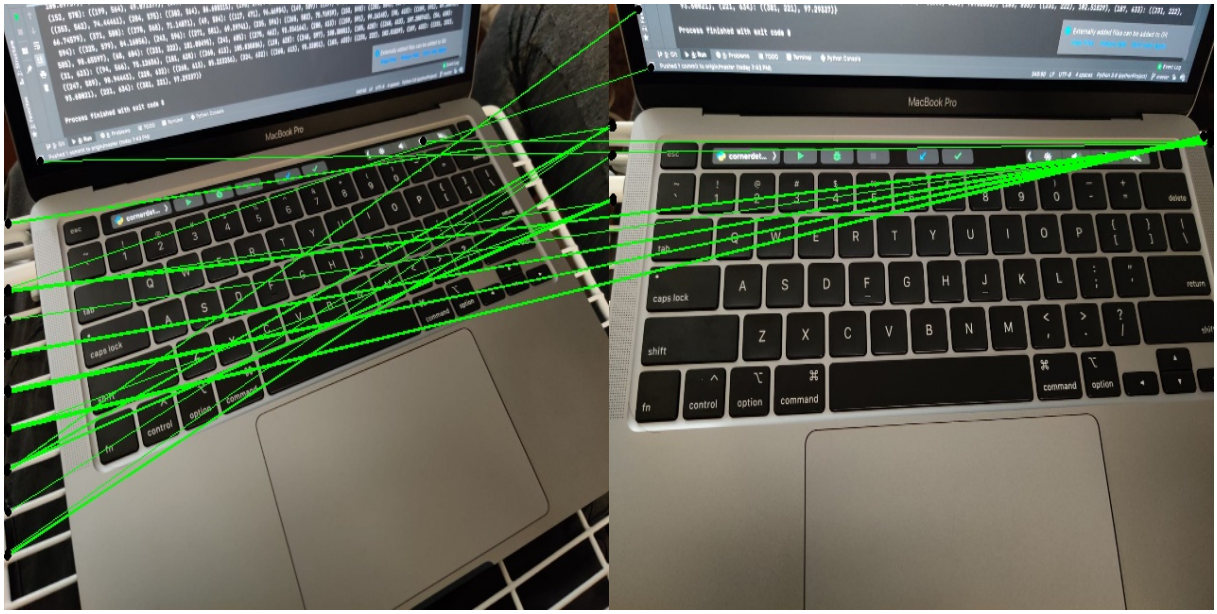

Figure 88: **SSD correspondence at sigma = 1.8**

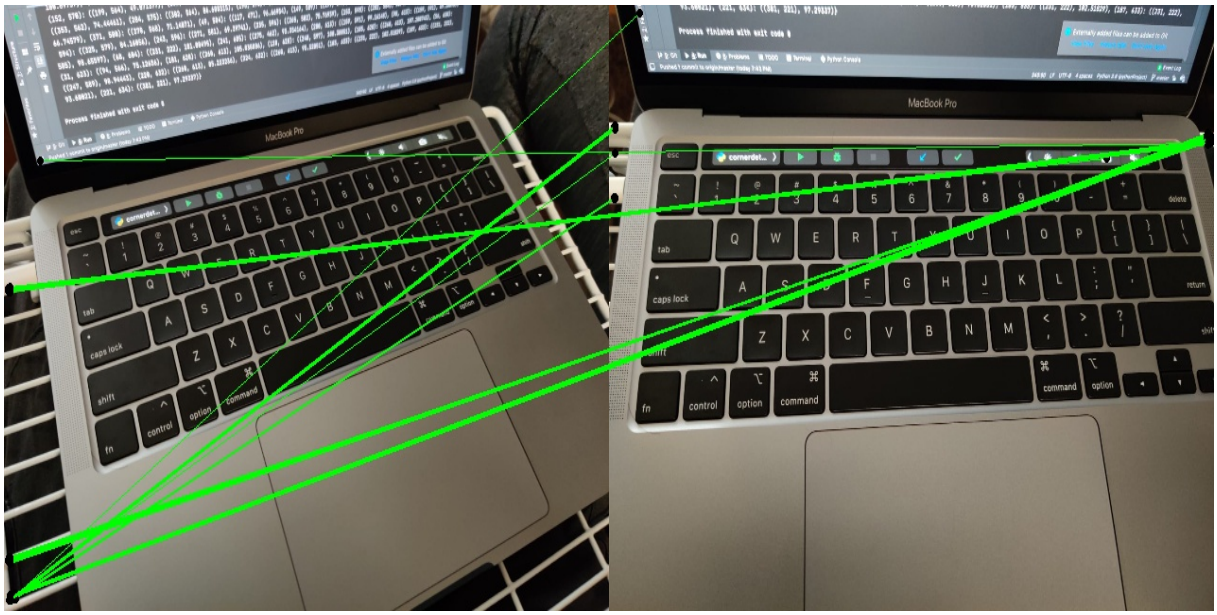

Figure 89: **SSD correspondence at sigma = 2.4**

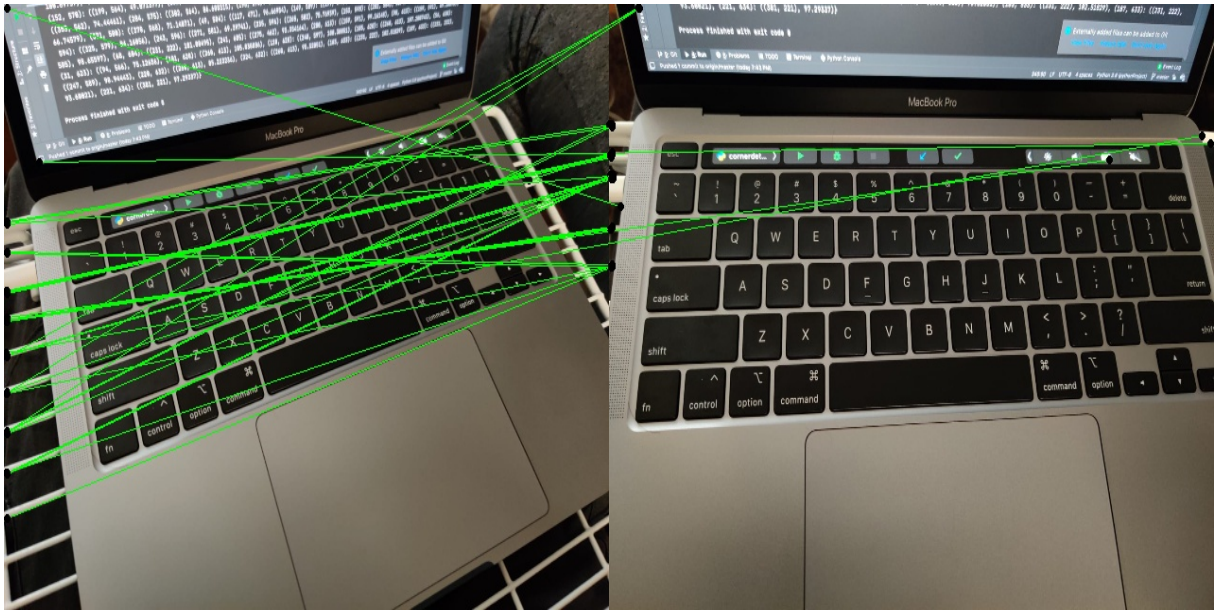

Figure 90: **NCC correspondence at sigma = 1.2**

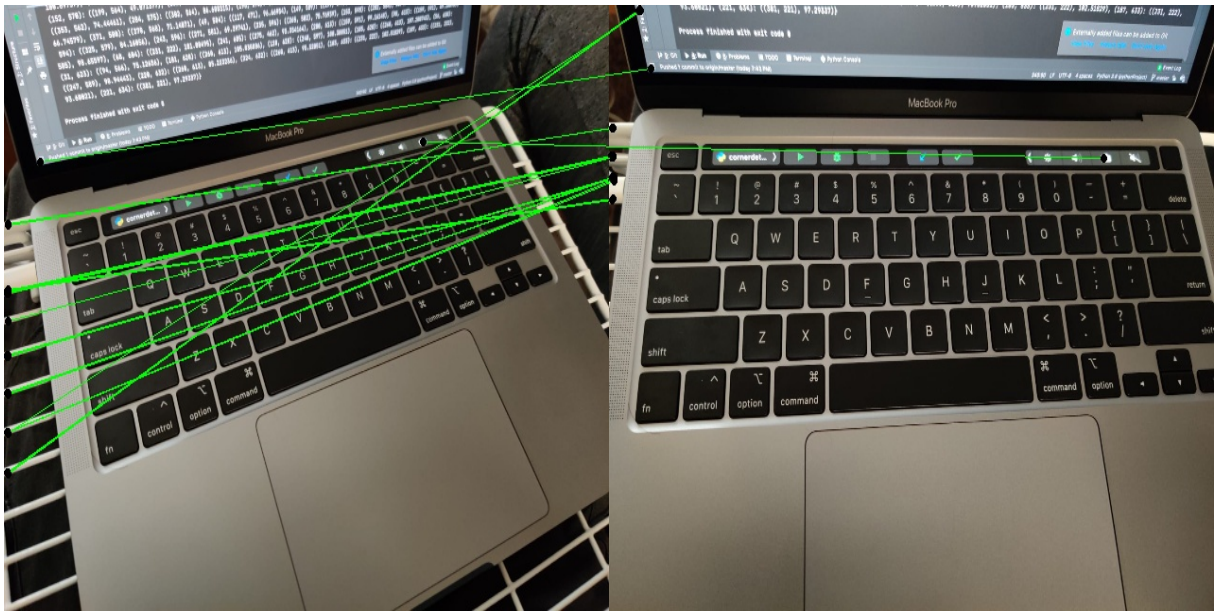

Figure 91: **NCC correspondence at sigma = 1.8**

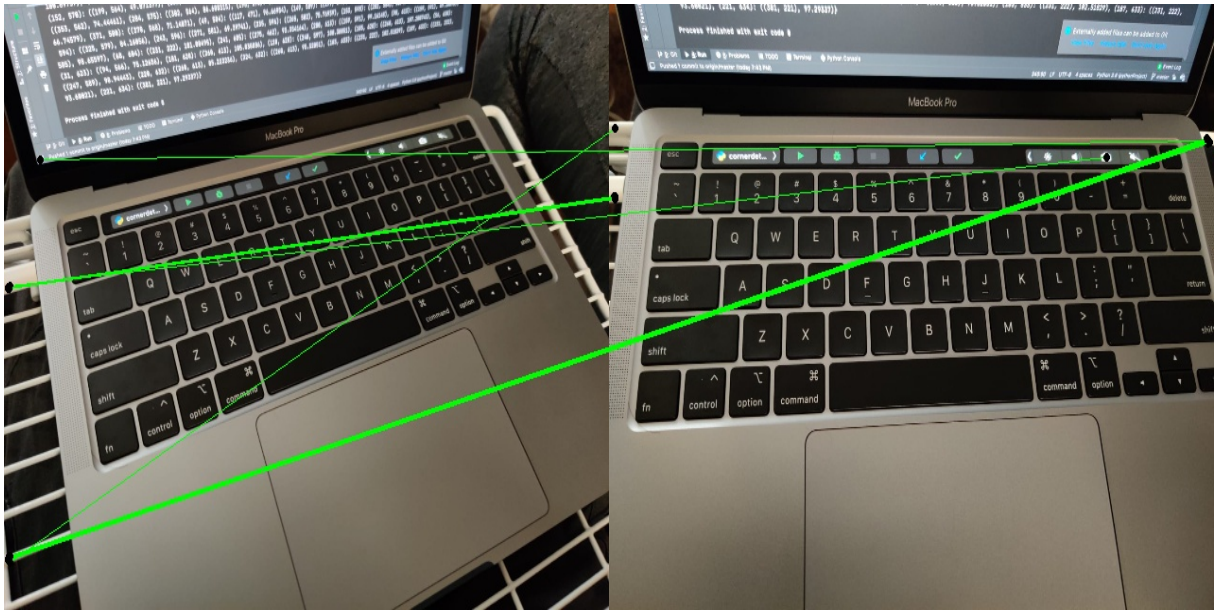

Figure 92: **NCC correspondence at sigma = 2.4**

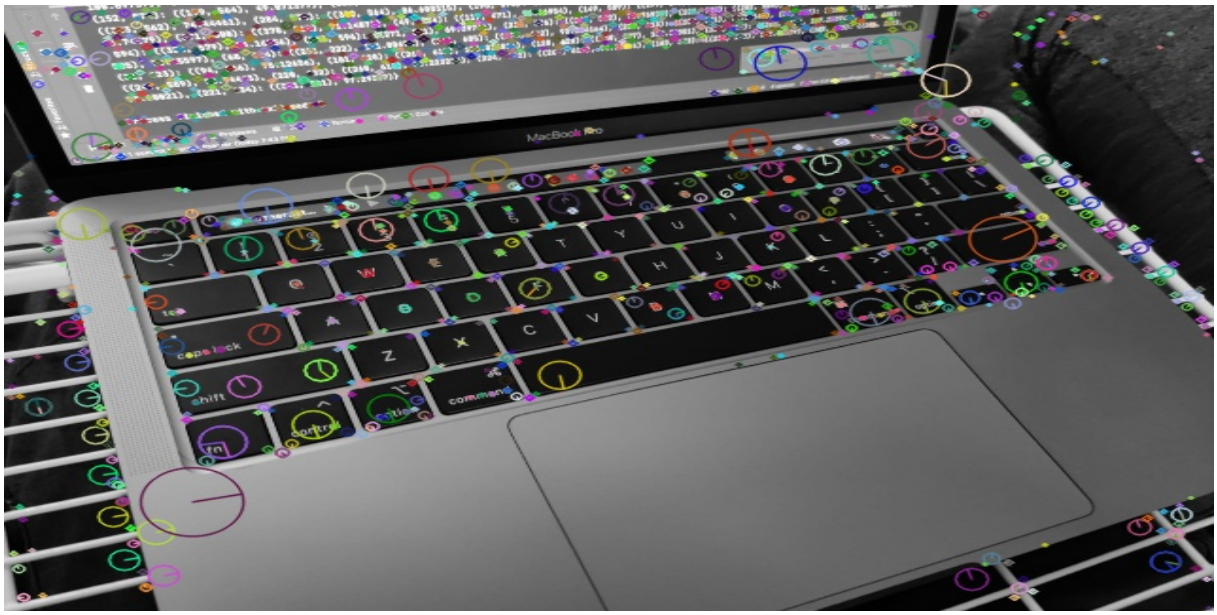

Figure 93: **Sift corner detection**

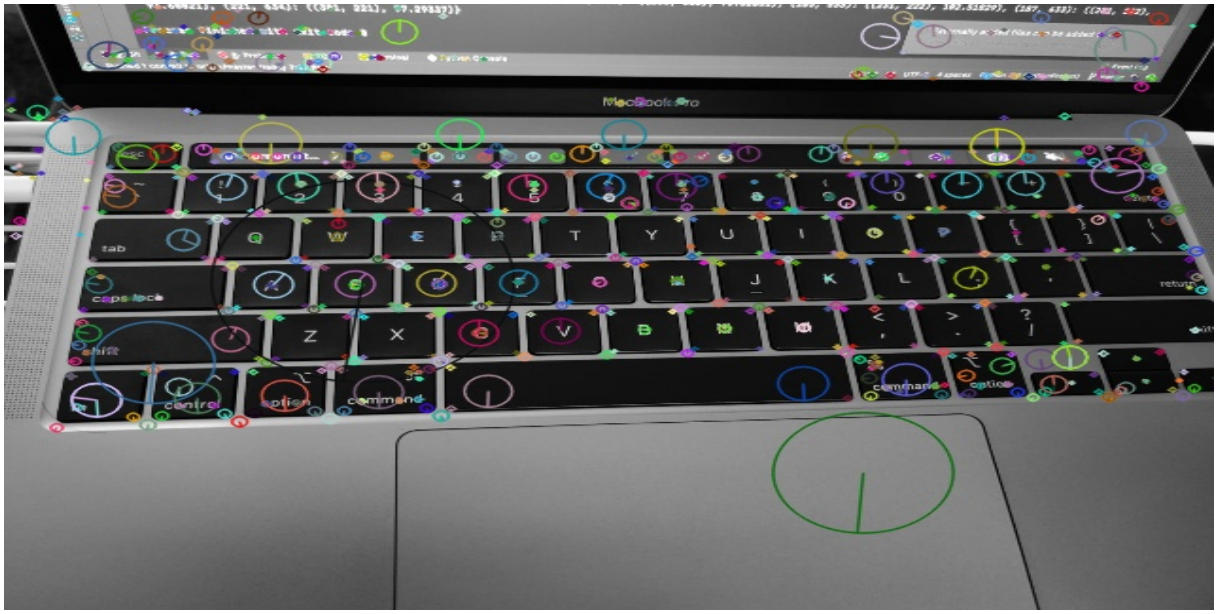

Figure 94: **Sift corner detection**

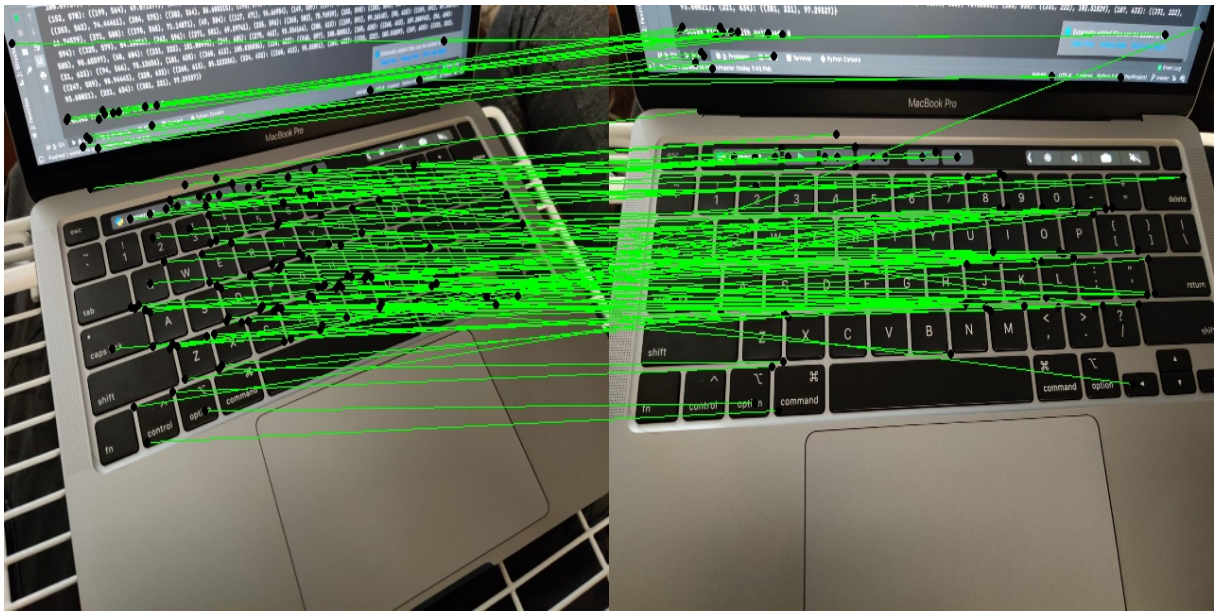

Figure 95: **Eucledian matching for SIFT**

Observations on the results

- We can observe that the number of corners detected decreases in number and also accuracy as we increase the sigma value. This is because of the smoothing effect. It increases as sigma increases.
- In our results, there is no major difference between SSD and NCC feature matching. But, we notice that NCC is bit better in some of the cases as it matches. If we were to pick between the two, based on our results, we can say that NCC is better.
- We can see that the Harris detector performs better on Image 3 compared to the other images. Image 3 has very rigid and bold features. The features are well lit compared to the other two images.
- SIFT detector results are obviously much better than the custom built Harris detector.
- For images with a lot of similar features like the one seen in Image pair 2, I would select a sigma value of 1.2-1.8. This produced better results than a sigma value of 0.7 or lower.
- For images with a very good and rigid scene, with a lot of unique features like the one we see in Image pair 3, a sigma value of 0.7 produced the best results. Higher sigma values produced less effective results for this pair of images.

Source Code

```
1
```

```
"""
Computer Vision - Purdue University - Homework 4

Author : Arjun Kramadhati Gopi, MS-Computer & Information
    Technology, Purdue University.
Date: September 28, 2020
```

```
6
7
8  [TO RUN CODE]: python3 cornerdetector.py
9  Output:
10     [jpg]: [Transformed images]
11 """
12
13 import cv2 as cv
14 import math
15 import numpy as np
16 import time
17 from scipy import signal as sg
18 import tqdm
19 import copy
20 import threading
21
22
23 class FeatureOperator:
24
25     def __init__(self, image_addresses, scale, kvalue=0.04):
26         """
27         Initialise the FeatureOperator object. This class can
               detect corner in an image either by the custom
28         built Harris detector or the OpenCV SIFT corner detector.
               The class can also detect correspondence in
29         a given pair of similar pictures either by SSD or NCC or
               the eucledian distance method.
30         :param image_addresses: List of images to work with. In
               this case it a pair of images.
31         :param scale: Scale value used to detecting the corners.
32         :param kvalue: K value is the constant used in the
               equation to calculate the Harris response.
33         """
34         self.image_addresses = image_addresses
35         self.scale = scale
36         self.originalImages = []
37         self.grayscaleImages = []
38         self.imagesizes = []
39         self.filters = {}
40         self.cornerpointdict = {}
41         self.slidingwindowdict = {}
42         self.correspondence = {}
43         self.kvalue = kvalue
44         for i in range(len(self.image_addresses)):
45             self.originalImages.append(cv.resize(cv.imread(self.
                   image_addresses[i]),(640,480)))
46             self.grayscaleImages.append(cv.resize(cv.cvtColor(cv.
                   imread(self.image_addresses[i]), cv.COLOR_BGR2GRAY
                   ),(640,480)))
47         self.siftobject = cv.SIFT_create()
48
49     def build_haar_filter(self):
50         """
```

```
51          Builds the two Haar filters to obtain dx and dy values by
                convolving over the images
52          :return: None. Adds the filters to a global dictionary
               self.filters
53          """
54          mvalue = int(np.ceil(4 * self.scale))
55          mvalue = mvalue + 1 if (mvalue % 2) > 0 else mvalue
56          blankfilter = np.ones((mvalue, mvalue))
57          blankfilter[:, :int(mvalue / 2)] = -1
58          self.filters["HaarFilterX"] = blankfilter
59          blankfilter = np.ones((mvalue, mvalue))
60          blankfilter[int(mvalue / 2):, :] = - 1
61          self.filters["HaarFilterY"] = blankfilter
62
63      def filter_corner_points(self, rscore, windowsize, queueImage
            , tag):
64          """
65          From the list of the proposed potential corner points, we
                apply a type of non-maxima suppression
66          to avoid the case of 'overlapping interest points'. That
               is, we pick the most promisin corner point
67          by picking only the one with the highest R-score or
               Harris response in a given window.
68          :param rscore: Harris response as calculated for all the
               pixels in the image
69          :param windowsize: This is currently set to a 29x29 pixel
                window size.We filter for corner points
70          within these windows.
71          :param queueImage: Index of the location at which the
               image under consideration is stored in the list
72          :param tag: Values to access and store values by key in
               the dictionaries
73          :return: None. Adds values to a global dictionary self.
               cornerpointdict
74          """
75
76          window = int(windowsize / 2)
77          for column in range(window, self.grayscaleImages[
               queueImage].shape[1] - window, 1):
78              for row in range(window, self.grayscaleImages[
                   queueImage].shape[0] - window, 1):
79                  panwindow = rscore[row - window:row + window +1,
                       column - window : column + window +1]
80                  #print(panwindow.shape)
81                  if rscore[row, column] == np.amax(panwindow):
82                      pass
83                  else:
84                      rscore[row, column] = 0
85          # self.cornerpointdict[tag] = rscore
86          print("Here")
87          rscoretemp = copy.deepcopy(rscore)
88          rscoretemp =rscoretemp.flatten()
89          Rcutoffvalue = rscoretemp[np.argsort(rscoretemp)
```

```
            [-100:]][0]
90          self.cornerpointdict[tag] = np.asarray(np.where(rscore >=
                Rcutoffvalue))
91          print("Here")
92          #print(len(np.asarray(np.where(rscore > 0))[0]))
93
94      def draw_corner_points(self, queueImage, tag):
95          """
96          We use this function to draw the corner points detected
               by the custom built Harris corner
97          detector.
98          :param queueImage: Index of the location at which the
               image under consideration is stored in the list
99          :param tag: Values to access and store values by key in
               the dictionaries
100         :return: Returns the image with the corner points drawn
101         """
102         points = self.cornerpointdict[tag].flatten()
103         pointXs = points[:int(len(points)/2)]
104         pointYs = points[int(len(points) / 2):]
105         image = copy.deepcopy(self.originalImages[queueImage])
106         for index in range(len(pointXs)):
107             cv.circle(image, (int(pointYs[index]), int(pointXs[
                   index])), 2, [0, 255, 0], 2)
108         return image
109
110     def determine_corners(self, type, queueImage, tag):
111         """
112         This function is used to calculate the Rscore values or
               the Harris response values. From these
113         values, we determine the final list of corner points
               after filtering them in the filter_corner_points()
114         function.
115         :param type: To specify the type of method being deployed
                to find the corner.
116         :param queueImage: Index of the location at which the
               image under consideration is stored in the list
117         :param tag: Values to access and store values by key in
               the dictionaries
118         :return: None. Calls the filter_corner_points() function
               by giving the Rscore values.
119         """
120         if type == 1:
121             # Harris Corner Method
122             dx = sg.convolve2d(self.grayscaleImages[queueImage],
                   self.filters["HaarFilterX"], mode='same')
123             dy = sg.convolve2d(self.grayscaleImages[queueImage],
                   self.filters["HaarFilterY"], mode='same')
124             dxsquared = dx * dx
125             dysquared = dy * dy
126             dxy = dx * dy
127             windowsize = int(5 * self.scale)
128             windowsize = windowsize if (windowsize % 2) > 0 else
```

```
                windowsize + 1
129             window = np.ones((windowsize, windowsize))
130             sumofdxsquared = sg.convolve2d(dxsquared, window,
                    mode='same')
131             sumofdysquared = sg.convolve2d(dysquared, window,
                    mode='same')
132             sumofdxdy = sg.convolve2d(dxy, window, mode='same')
133             detvalue = (sumofdxsquared * sumofdysquared) - (
                    sumofdxdy * sumofdxdy)
134             tracevalue = sumofdysquared + sumofdxsquared
135             if self.kvalue == 0:
136                 self.kvalue = detvalue / (tracevalue * tracevalue
                        + 0.000001)
137                 self.kvalue = np.sum(self.kvalue) / (
138                             self.grayscaleImages[queueImage].
                                shape[0] * self.grayscaleImages[
                                queueImage].shape[1])
139             Rscore = detvalue - (self.kvalue * tracevalue *
                    tracevalue)
140             Rscore = np.where(Rscore < 0, 0, Rscore)
141             print(Rscore.shape)
142             self.filter_corner_points(Rscore, 29, queueImage, tag
                    )
143
144     def get_sliding_windows(self, windowsize, queueImage,tag,
            dict, dicttag):
145         """
146         This function generates the neighborhood boxes around
                each corner point in the pair of images.
147         These neighborhood boxes or some windowsize (21X21) and
                they are used to calculate the SSD/NCC values
148         to determine correspondence points in the given pair of
                images.
149         :param windowsize: Size of neighborhood boxes
150         :param queueImage: Index of the location at which the
                image under consideration is stored in the list
151         :param tag: Values to access and store values by key in
                the dictionaries
152         :param dict: Empty dictionary object for the two
                simultaneous thread that run this function.
153         :param dicttag: Values to access and store values by key
                in the dictionaries
154         :return: None. Stores the windows in the global
                dictionary self.slidingwindowdict
155         """
156         points = self.cornerpointdict[tag].flatten()
157         pointXs = points[:int(len(points)/2)]
158         pointYs = points[int(len(points) / 2):]
159         for index in range(len(pointXs)):
160             row = pointXs[index]
161             column = pointYs[index]
162             array = self.grayscaleImages[queueImage][row:row+
                    windowsize, column:column+windowsize]
```

```
163             if array.shape == (29,29):
164                 dict[(row, column)] = array
165             else:
166                 resultarray = np.zeros((29,29))
167                 resultarray[:array.shape[0],:array.shape[1]] =
                        array
168                 dict[(row, column)] = resultarray
169
170         self.slidingwindowdict[dicttag] = dict
171
172     def calculate_correspondence(self, style, tags):
173         """
174         This function calculates the correspondence between the
                two images in the pair.
175         The methods employed in this function are 1) SSD and 2)
                NCC. We use either one to
176         calculte the correspondence.
177         :param style: Either SSD or NCC method to calculate
                correspondence.
178         :param tags: Values to access and store values by key in
                the dictionaries
179         :return: None. Stores the correpondece/the matching pairs
                 of points in the a global dictionary
180         self.correspondence
181         """
182         windowsone = copy.deepcopy(self.slidingwindowdict[tags
                [0]])
183         windowstwo = copy.deepcopy(self.slidingwindowdict[tags
                [1]])
184         list = []
185         list2 = []
186         list3=[]
187         if style =="SSD":
188             list3=[]
189             for id1 in windowsone:
190                 list =[]
191                 list2 =[]
192                 for id2 in windowstwo:
193                     if id2 in list3:
194                         pass
195                     else:
196                         difference = windowsone[id1]-windowstwo[
                                id2]
197                         ssd = np.sum(difference*difference)
198                         list.append(ssd)
199                         list2.append(id2)
200                 ssd = min(list)
201                 id = list2[list.index(ssd)]
202                 list3.append(id)
203                 windowsone[id1] = (id,ssd)
204             self.correspondence[tags[2]] = windowsone
205         if style =="NCC":
206             list3=[]
```

```
207             for id1 in windowsone:
208                 list =[]
209                 list2=[]
210                 for id2 in windowstwo:
211                     if id2 in list3:
212                         pass
213                     else:
214                         meanvalueone = np.mean(windowsone[id1])
215                         meanvaluetwo = np.mean(windowstwo[id2])
216                         diffone = windowsone[id1] -meanvalueone
217                         difftwo = windowstwo[id2] - meanvaluetwo
218                         ssd = np.sum(diffone*difftwo)
219                         ssdone = np.sum(diffone*diffone)
220                         ssdtwo = np.sum(difftwo*difftwo)
221                         list.append(ssd/(np.sqrt(ssdone*ssdtwo)
                                +0.000001))
222                         list2.append(id2)
223                 ncc = max(list)
224                 id = list2[list.index(ncc)]
225                 list3.append(id)
226                 windowsone[id1]=(id,ncc)
227             self.correspondence[tags[2]]=windowsone
228 
229     def draw_correspondence(self, tags, cutoffvalue, style):
230         """
231         This function draws the correspondence between the corner
                points in the pair of images. We denote each
232         corner point by a small circle around it. The
               correspondence is denoted by a line connecting the two
                points.
233         Before drawing the points, we first filter the
               correspondences based on a cutoff value so that we
               retain only
234         fairly accurate matches and not completely-off matches.
235         :param tags: Values to access and store values by key in
               the dictionaries
236         :param cutoffvalue: Value used to filter the list of
               matched corner points
237         :param style: Either filter values above the cutoff value
                or filter the values below it.
238         :return: Returns the resultant stitched image with the
               denoted correspondence lines.
239         """
240         copydict = copy.deepcopy(self.correspondence[tags[0]])
241         print(copydict)
242         for (key,value) in self.correspondence[tags[0]].items():
243             if style == 'greaterthan':
244                 if value[1]> cutoffvalue:
245                     copydict.pop(key)
246             elif style == 'lesserthan':
247                 if value[1]< cutoffvalue:
248                     copydict.pop(key)
249         resultImage = np.hstack((self.originalImages[0], self.
```

```
            originalImages[1]))
250         horizontaloffset = 640
251         print(copydict)
252         for (key,value) in copydict.items():
253             # print((key,value))
254             columnvalueone = key[1]
255             rowvalueone = key[0]
256             columnvaluetwo = value[0][1] + horizontaloffset
257             rowvaluetwo = value[0][0]
258             cv.line(resultImage, (columnvalueone,rowvalueone), (
                    columnvaluetwo,rowvaluetwo), [0, 255, 0], 1 )
259             cv.circle(resultImage,(columnvalueone, rowvalueone),
                    2, [0, 0, 0], 2)
260             cv.circle(resultImage, (columnvaluetwo, rowvaluetwo),
                     2, [0, 0, 0], 2)
261         return resultImage
262
263     def sift_corner_detect(self, queueImage, tag):
264         """
265         This function detected and computes the sift keypoints
                and the descriptors. We use the generated keypoints
266         to draw them on the picture. These are the detected
                corners.
267         :param queueImage: Index of the location at which the
                image under consideration is stored in the list
268         :param tag: Values to access and store values by key in
                the dictionaries
269         :return: None. Stores the image.
270         """
271         keypoint, descriptor = self.siftobject.detectAndCompute(
                self.grayscaleImages[queueImage], None)
272         self.cornerpointdict[tag] = (keypoint, descriptor)
273         img = cv.drawKeypoints(self.grayscaleImages[queueImage],
                keypoint, copy.deepcopy(self.originalImages[queueImage
                ]), flags=cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
274         cv.imwrite(tag + '.jpg', img)
275
276     def sift_correpondence(self, queueImages, tags, method):
277         """
278         This function estimates the correspondences between the
                sift corners detected in the pair of images.
279         We have two options to estimate this: 1) Using BFMatcher
                function of OpenCV to get K-Nearest
280         Neighbors or 2) Use a custom built
281         eucledian distance based estimator
282         :param queueImages: Index of the location at which the
                image under consideration is stored in the list
283         :param tags: Values to access and store values by key in
                the dictionaries
284         :param method: Use BFMatcher or custom built eucledian
                matcher.
285         :return: None. Stores the matched keypoints in a global
                dictionary self.correspondence
```

```
286             """
287             (keypoint1, descriptor1) = self.cornerpointdict[tags[0]]
288             (keypoint2, descriptor2) = self.cornerpointdict[tags[1]]
289             if method =='OpenCV':
290                 matchedpoints = cv.BFMatcher().knnMatch(descriptor1,
                        descriptor2, k=2)
291                 filteredmatchedpoints = []
292                 for pointone, pointtwo in matchedpoints:
293                     if pointone.distance < (pointtwo.distance * 0.75)
                            :
294                         filteredmatchedpoints.append([pointone])
295                 result = cv.drawMatchesKnn(self.grayscaleImages[
                        queueImages[0]], keypoint1, self.grayscaleImages[
                        queueImages[1]],keypoint2, filteredmatchedpoints,
                        None, flags=2)
296                 cv.imwrite("Sift_Correspondence.jpg", result)
297             elif method =='Custom':
298                 tempdict = dict()
299                 for index,element in enumerate(descriptor1):
300                     list = []
301                     list2 = []
302                     for index2, element2 in enumerate(descriptor2):
303                         eucledianddistance = np.sqrt(np.sum(np.square
                                ((element-element2))))
304                         list.append(eucledianddistance)
305                         list2.append(keypoint2[index2])
306                     minimumvalue = min(list)
307                     id = list2[list.index(minimumvalue)]
308                     tempdict[(int(keypoint1[index].pt[1]),int(
                            keypoint1[index].pt[0]))]=((int(id.pt[1]),int(
                            id.pt[0])),minimumvalue)
309                 self.correspondence[tags[2]] = tempdict
310
311
312
313 if __name__ == "__main__":
314     """
315     Code starts here. After each run, change images to get
            results for a new set of image pair.
316     """
317
318     tester = FeatureOperator(['hw4_Task1_Images/pair5/1.jpg','
            hw4_Task1_Images/pair5/2.jpg'], 0.7)
319     tester.build_haar_filter()
320     tester.determine_corners(1, 0, "Harris1")
321     tester.determine_corners(1, 1, "Harris2")
322     image = tester.draw_corner_points(0,"Harris1")
323     cv.imwrite("1.jpg", image)
324     image = tester.draw_corner_points(1, "Harris2")
325     cv.imwrite("2.jpg", image)
326     thread_image_one = threading.Thread(target=tester.
            get_sliding_windows, args=(21,0,"Harris1",dict()," 
            Image1HarrisSW",))
```

```
327     thread_image_two = threading.Thread(target=tester.
           get_sliding_windows, args=(21, 1, "Harris2", dict(), "
           Image2HarrisSW", ))
328     thread_image_one.start()
329     thread_image_two.start()
330     thread_image_one.join()
331     thread_image_two.join()
332     # tester.calculate_correspondence("SSD", ("Image1HarrisSW", "
           Image2HarrisSW","Image1to2SSD", "Image1to2SSDValues"))
333     tester.calculate_correspondence("SSD", ("Image1HarrisSW", "
           Image2HarrisSW", "Image1to2NCC", "Image1to2NCCValues"))
334     image = tester.draw_correspondence(("Image1to2NCC", "
           Image1to2NCCValues"), 8000000, 'greaterthan')
335     cv.imwrite("result.jpg", image)
336
337     tester.sift_corner_detect(0, "Sift1")
338     tester.sift_corner_detect(1, "Sift2")
339     tester.sift_correpondence((0, 1), ("Sift1","Sift2","
           Image1to2Eucledian"),'Custom')
340     image=tester.draw_correspondence(("Image1to2Eucledian","
           Image1to2Eucledianvalues"),80, 'greaterthan')
341     cv.imwrite("result.jpg", image)
```