ECE 661 Computer Vision HW 4

Raghav Malik

September 28, 2020

1 Theory

1.1 Theory Question

LoG and DoG Equivalence

The LoG, or Laplacian of Gaussian operator, takes the form $\nabla^2 \tilde{f}(x, y)$ where $\tilde{f}$ is the original image f convolved with a Gaussian kernel for smoothing and the ∇^2 operator represents the sum of the second partial in the x direction and the second partial in the y direction (in other words, $\frac{\partial^2 \tilde{f}}{\partial x^2} + \frac{\partial^2 \tilde{f}}{\partial y^2}$). Since the Gaussian smoothing depends on the scale factor σ, we can bind this into the function and write the smoothed image as:

$$\tilde{f}(x, y) = \iint_{-\infty}^{\infty} f(x', y') g_\sigma(x - x', y - y') dx' dy'$$

Taking the partial derivative with respect to σ, we now see that

$$\frac{\partial \tilde{f}(x, y, \sigma)}{\partial \sigma} = \sigma \nabla^2 \tilde{f}(x, y, \sigma)$$

which is exactly the form of the LoG. Thus, we can compute the LoG by numerically estimating this derivative. Practically speaking, this means smoothing the image at two different scales, σ and $\sigma + \delta_\sigma$, and finding the difference (i.e. a Difference of Gaussians, or DoG).

Why DoG is more efficient

Using the DoG method to estimate the LoG is often far more efficient than directly computing the LoG, especially for large values of σ. The reason behind this is that in order to directly compute the LoG, we need to find the second order partials which is done by convolving the entire image with the second order Sobel operators. The number of steps required to compute a single pixel in the convolution is proportional to σ^2. However, when using the DoG method, no explicit Sobel operators are required for the convolution. The Gaussian is separable, which means that computing the 2D convolution of the entire image with a Gaussian kernel is equivalent to computing two 1D convolutions of the image (rows and then columns) and combining them. Thus, the number of steps is simply proportional to σ. As the scale factor grows larger, using the DoG method has better asymptotic complexity. Furthermore, often we can get similar results with a smaller scale factor when doing DoG as compared to LoG.

1.2 Harris Corner Detection

Algorithm

The Harris Corner Detection algorithm is a primitive algorithm for finding interest points in an image. It attempts to identify points in an image that contain corners by using the x and y gradients. If a single one of these gradients is high (or if they are scalar multiples of each other) that suggests the presence of a

single edge. However, if both gradients are high and independent, this suggests two non-parallel edges in the neighborhood and makes that pixel a likely candidate for a corner. The implementation of the algorithm is as follows: First, a scale σ is selected for the image. The scale controls how "closely" the algorithm looks at features to balance between identifying the true corners and getting spurious hits from noise. Next, the x-gradient I_x and the y-gradient I_y of the image are computed. This is done by convolving with either the corresponding Sobel operator or a Haar wavelet generated using the scale. Next, for each pixel, we look at a $5\sigma \times 5\sigma$ window around it in both I_x and I_y to find their correlation for that neighborhood. This amounts to constructing the matrix

$$M = \begin{bmatrix} \sum_w d_x^2 & \sum_w d_x d_y \\ \sum_w d_x d_y & \sum_w d_y^2 \end{bmatrix}$$

If this matrix is singular, that means the window contains at most a single edge. Thus, we find all the points such that this matrix is nonsingular and tag them as corners.

Implementation Notes

Computing the eigenvalues of a 2×2 matrix to determine if its nonsingular is an expensive process for each pixel. Instead, we threshold on the ratio of its determinant and the square of its trace. An image K is generated that contains the value

$$\frac{\sum d_x^2 \sum d_y^2 - (\sum d_x d_y)^2}{(\sum d_x^2 + \sum d_y^2)^2}$$

at each pixel (where the sums are computed over the windows around the pixel). We then find k to be a certain of K (empirically chosen as 70, with a higher percentile resulting in fewer corners identified), construct $R = \det M - k \times Tr(M)$, and identify the pixels for which this value is strictly positive. Finally, we do a nonmaximum suppression on the window around each identified pixel to pinpoint the corners.

1.3 Corner Matching with Harris

In order to identify correspondence points between two images, we first use the Harris algorithm to find the corner points in each one. Then, a neighborhood (empirically chosen to be of size 21×21) around each pair corner points is considered and a distance metric is computed between them. Every corner point in the first image is paired with the corner point in the second image that yields the best (lowest) distance score. The distance metrics are as follows:

NCC (Normalized Cross-Correlation)

The NCC metric computes the 2D cross-correlation between the neighborhoods of each corner point. if $f[x, y]$ is the neighborhood around the first, $g[x, y]$ is the neighborhood around the second, and μ_f and μ_g represent the means of each neighborhood, then the NCC metric is computed as follows:

$$NCC = \frac{\sum\sum(f[x,y] - \mu_f)(g[x,y] - \mu_g)}{\sqrt{\sum\sum(f[x,y] - \mu_f)^2 \sum\sum(g[x,y] - \mu_g)^2}}$$

SSD (Squared Sum of Differences)

The SSD metric computes the square of the Euclidean norm of the difference of the two neighborhoods. It is computed as follows:

$$SSD = \sum_x \sum_y (f[x,y] - g[x,y])^2$$

1.4 The SIFT Algorithm

The DoG Pyramid

The first step of the SIFT algorithm is to construct the DoG pyramid. Convolve the image with a Gaussian kernel with an initial scale. Then downsample by a factor of two, double the scale (so you can use the same kernel), and repeat for several iterations. Within each image size, interpolate the scale values and convolve with the new kernels to produce a stack of images at each resolution, or a pyramid. Finally, within each resolution stack (octave), take the difference of successive smoothed images and use them to create a new pyramid of DoG images.

Local Extrema

Find the discrete local extrema in each octave of the DoG pyramid. Each point is compared to its 26 neighbors (8 around it, 9 above, and 9 below) and flagged if it is the maximum or minimum. In order to find at least two extrema per octave, we need four image stacks, which in turn translates to finding a minimum of 5 different smoothing scales per octave for the Gaussian pyramid.

Interpolation

An extremum identified in a higher octave has considerably less resolution than one identified in a lower octave. To make up for this, we construct a second order Taylor expansion around it and attempt to interpolate the true maximum or minimum in the area. The Taylor expansion is over vectors, so we use the Jacobian and Hessian matrices to compute the first and second partial derivatives, respectively.

Descriptors

First, the identified extrema are thresholded based on the value of the Taylor expansion at those points. Typically, extrema where $|D(\vec{x})| < 0.03$ are excluded. Next, an orientation is determined for each point by taking the gradient of the Gaussian of the image at the same scale as the extremum. The gradient magnitude and direction is computed for each pixel in a neighborhood around the point, and the dominant vector in this neighborhood is computed. (Each angle is weighted by its magnitude, and whichever angle 'bin' has the highest total sum wins). Finally, we compute the descriptor vectors. We take a 16×16 neighborhood around the interest point and divide it into 4×4 blocks of 4×4 pixels each. The gradient magnitudes are weighted by a Gaussian to reduce the importance of pixels far from the extremum. For each cell, an 8-bin histogram is calculated from the weighted orientation values of the pixels in the cell. These 16 histograms become the 128-element feature vector for that interest point, which is normalized to make it invariant to illumination before returning.

2 Results

2.1 Harris Corners

Figure 1: pair1 corners at $\sigma = 9$

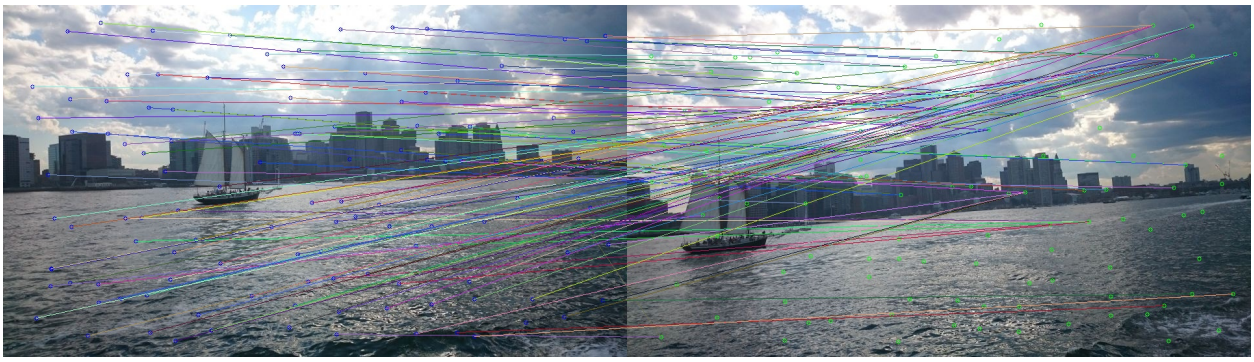


Figure 2: pair1 correspondences at $\sigma = 9$ using SSD

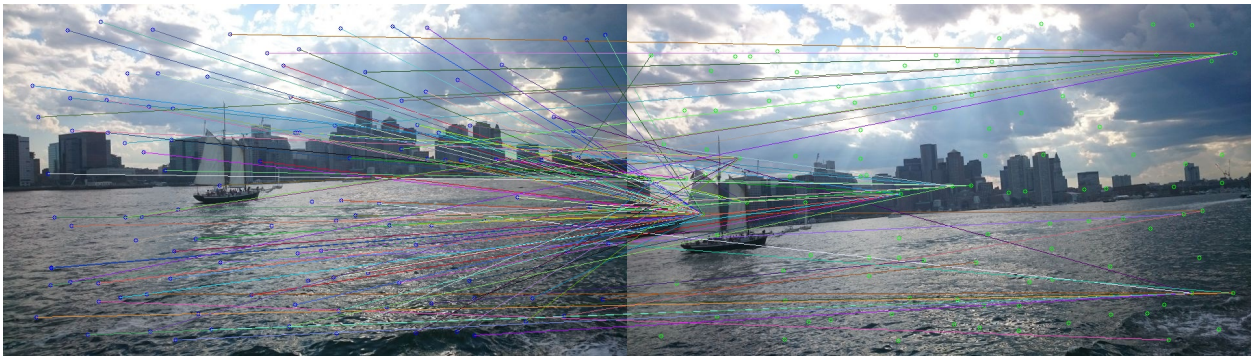


Figure 3: pair1 correspondences at $\sigma = 9$ using NCC

Figure 4: pair1 corners at $\sigma = 11$

Figure 5: pair1 correspondences at $\sigma = 11$ using SSD

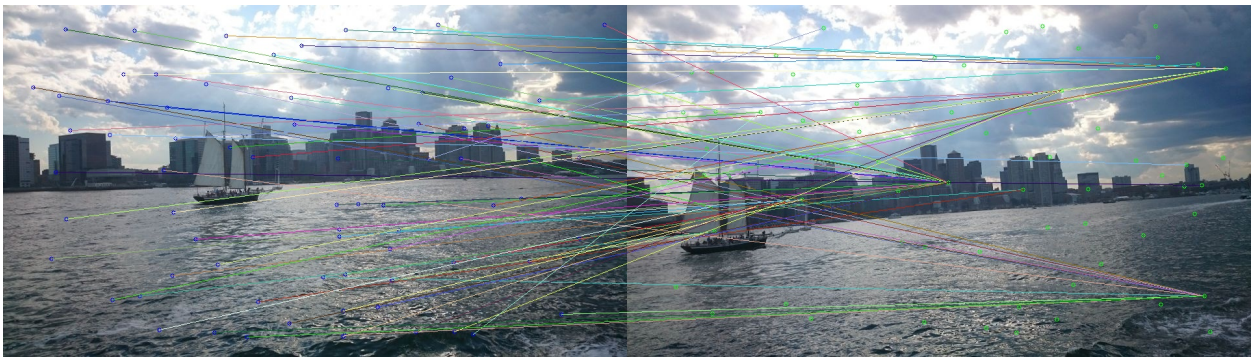


Figure 6: pair1 correspondences at $\sigma = 11$ using NCC

Figure 7: pair1 corners at $\sigma = 13$

Figure 8: pair1 correspondences at $\sigma = 13$ using SSD

Figure 9: pair1 correspondences at $\sigma = 13$ using NCC

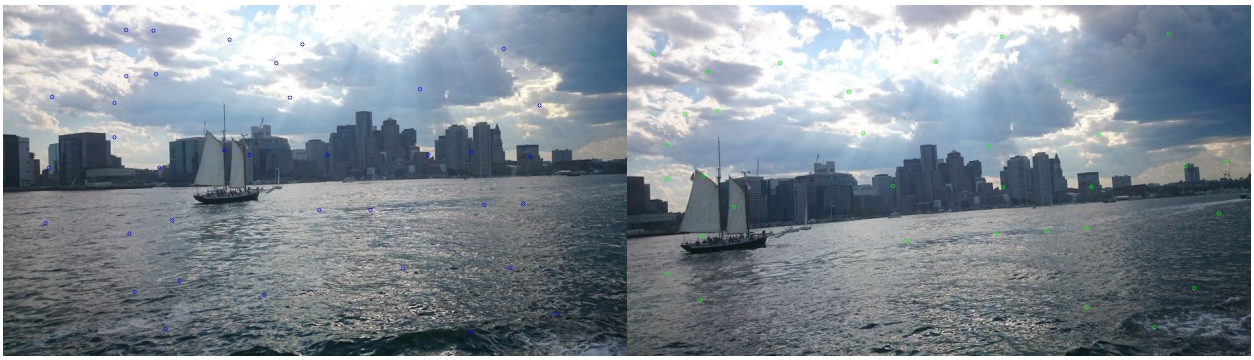


Figure 10: pair1 corners at $\sigma = 15$

Figure 11: pair1 correspondences at $\sigma = 15$ using SSD

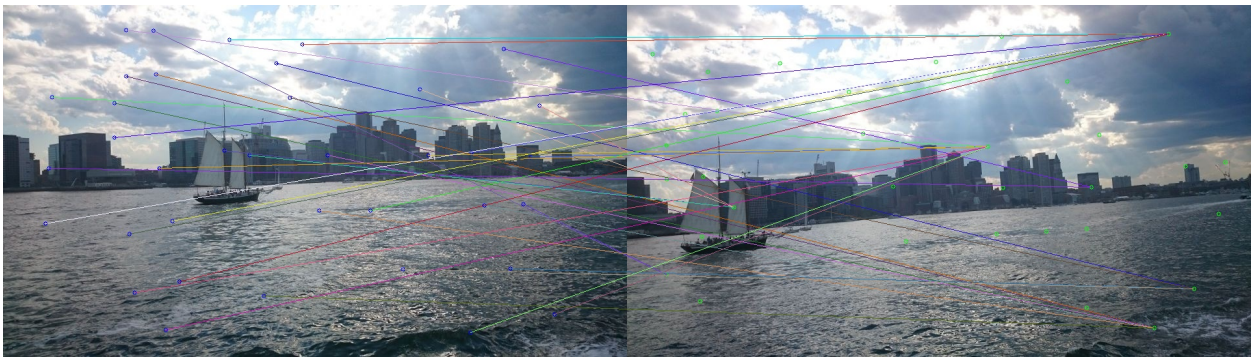


Figure 12: pair1 correspondences at $\sigma = 15$ using NCC

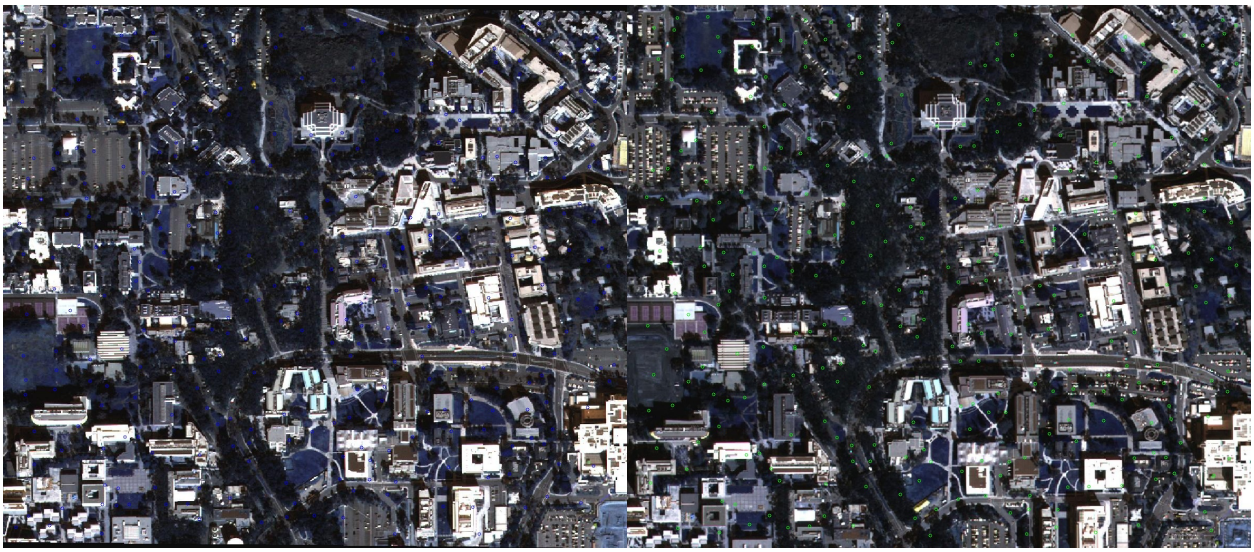


Figure 13: pair2 corners at $\sigma = 9$

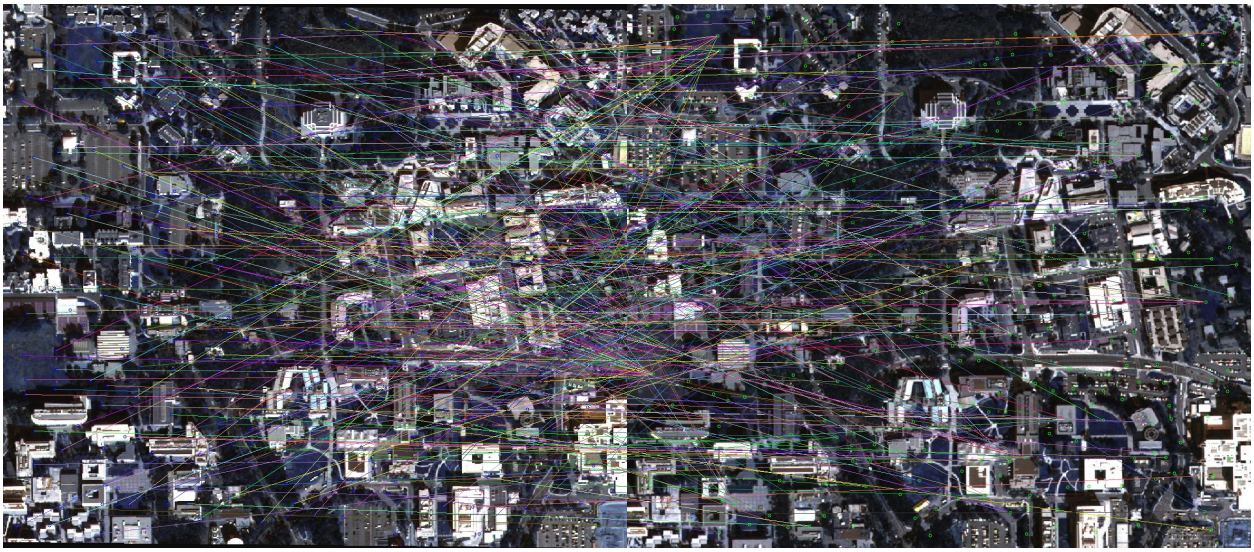


Figure 14: pair2 correspondences at $\sigma = 9$ using SSD

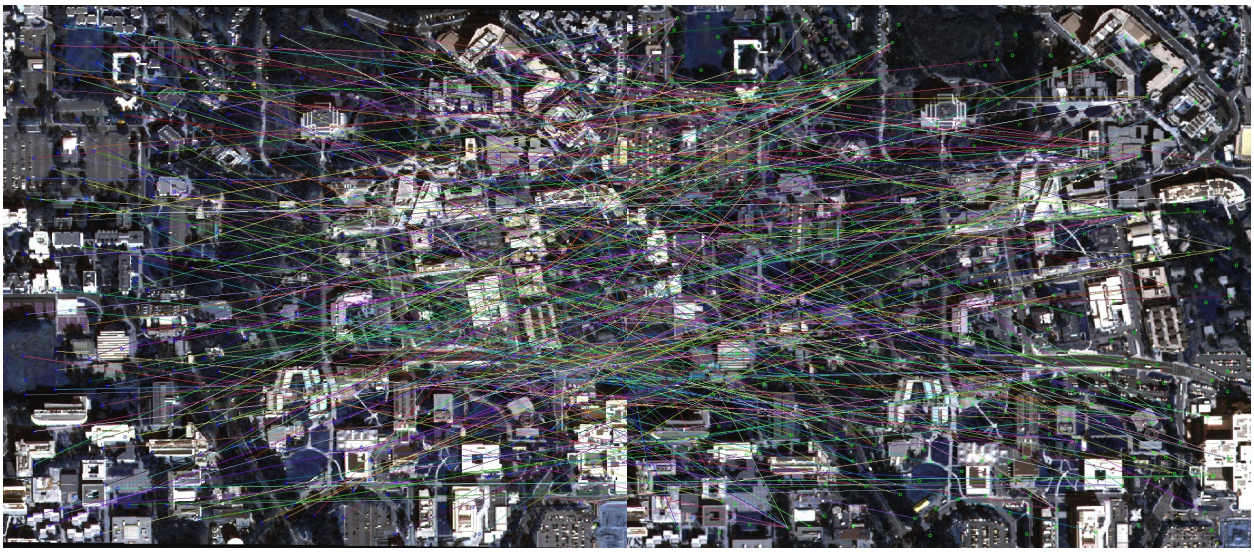


Figure 15: pair2 correspondences at $\sigma = 9$ using NCC

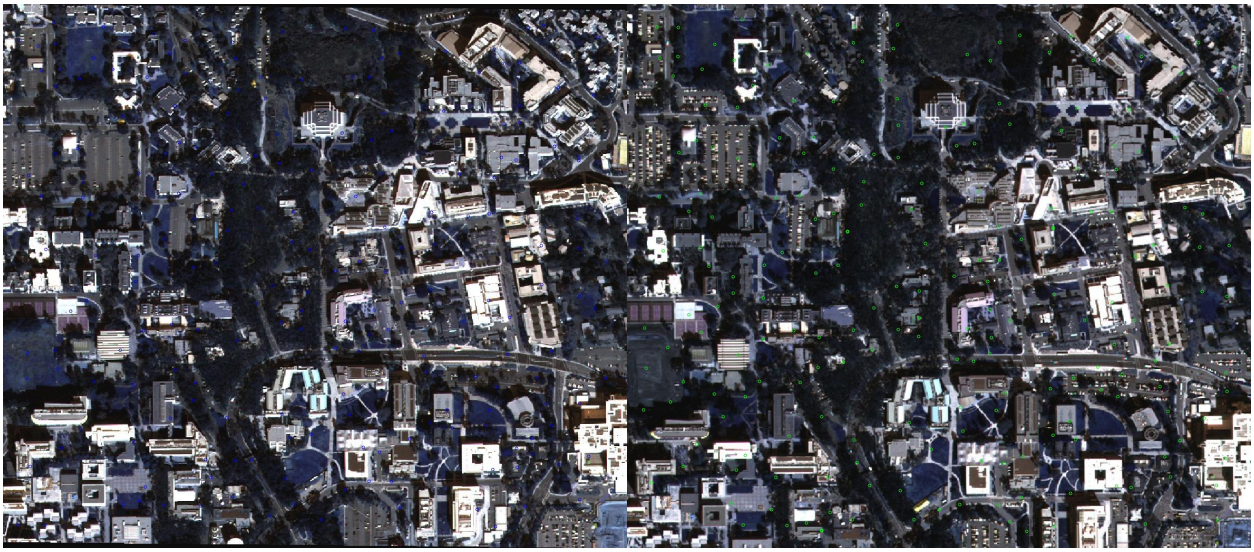


Figure 16: pair2 corners at $\sigma = 11$

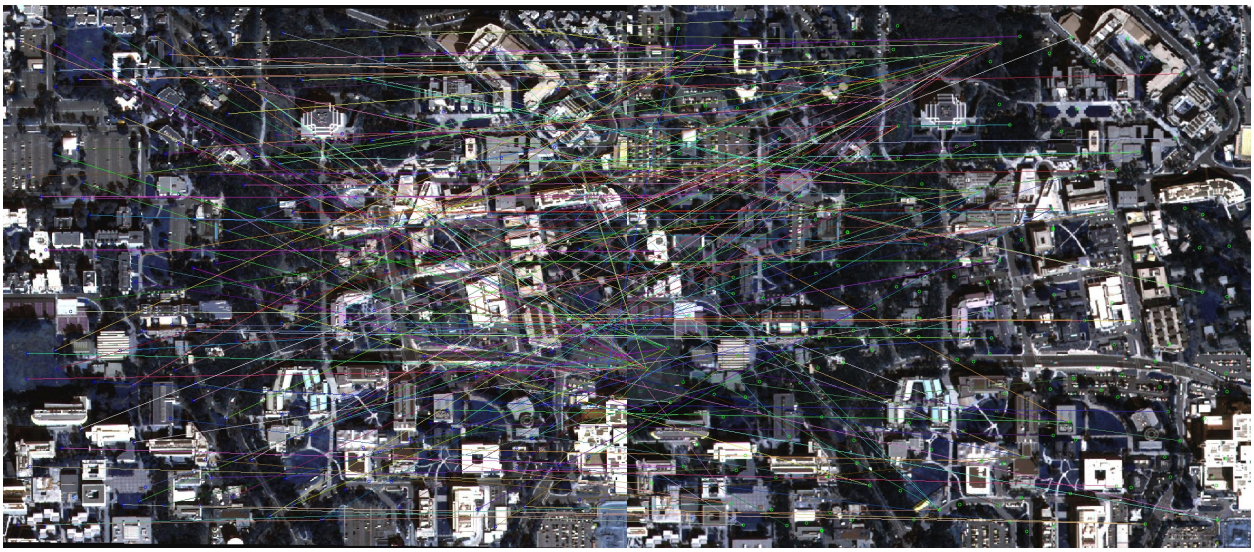


Figure 17: pair2 correspondences at $\sigma = 11$ using SSD

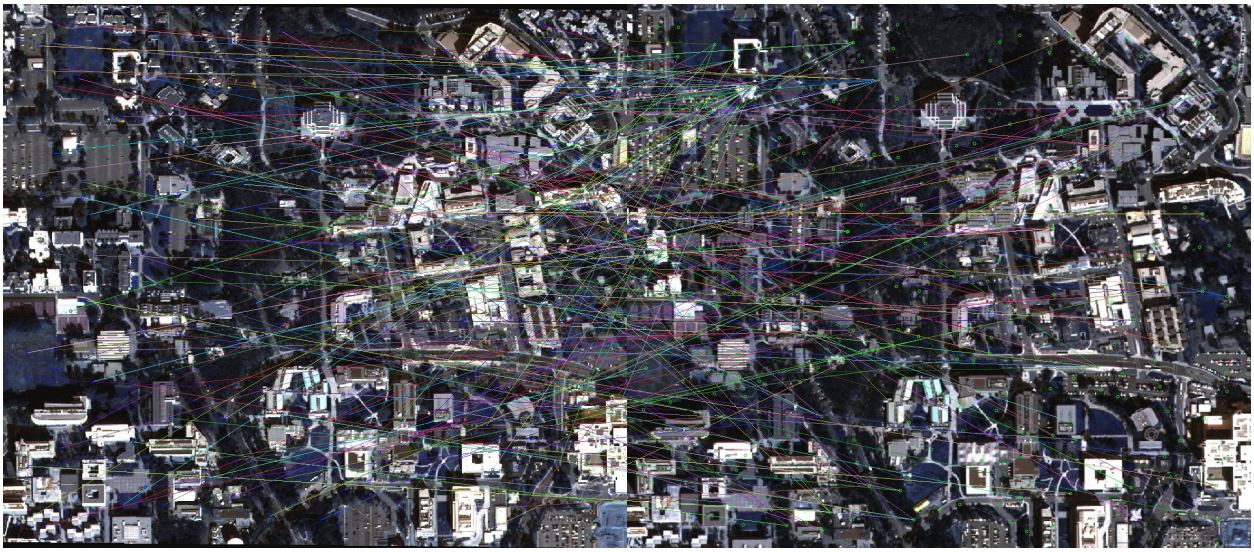


Figure 18: pair2 correspondences at $\sigma = 11$ using NCC

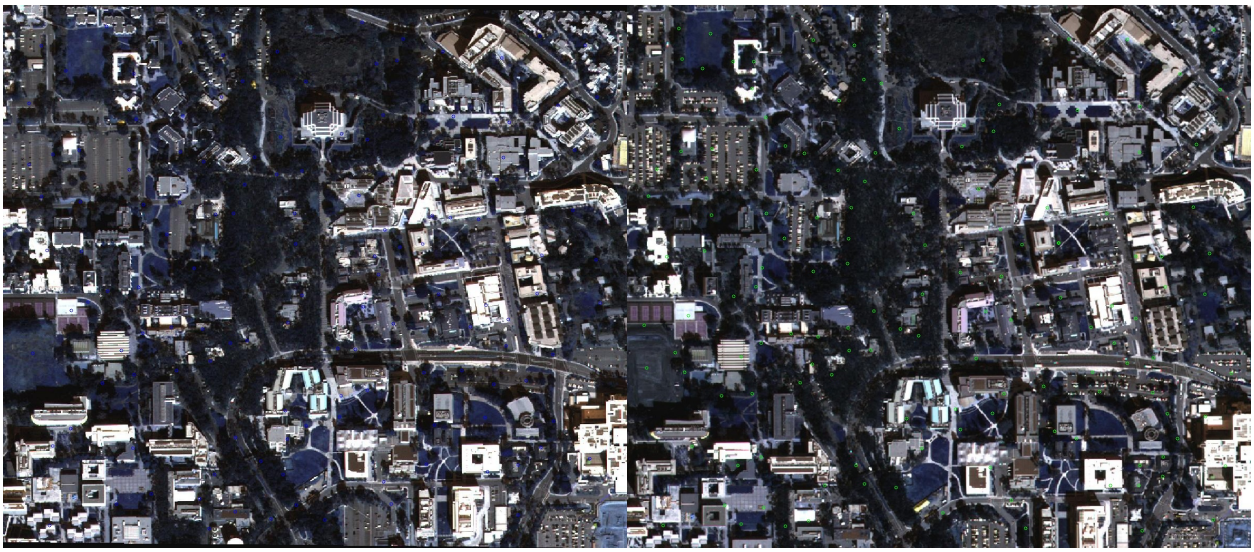


Figure 19: pair2 corners at $\sigma = 13$

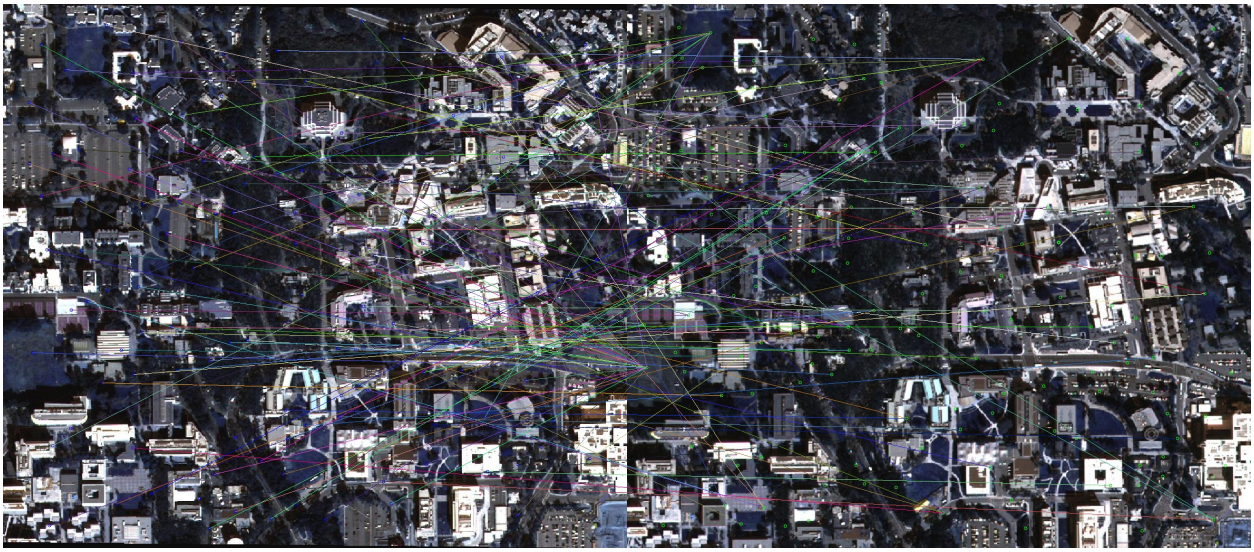


Figure 20: pair2 correspondences at $\sigma = 13$ using SSD

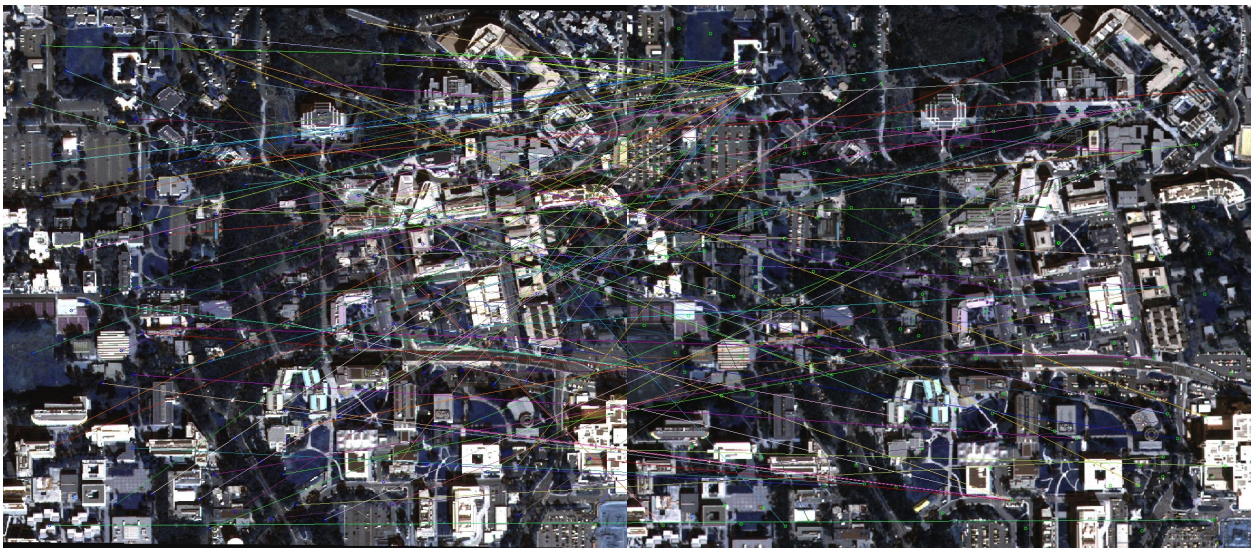


Figure 21: pair2 correspondences at $\sigma = 13$ using NCC

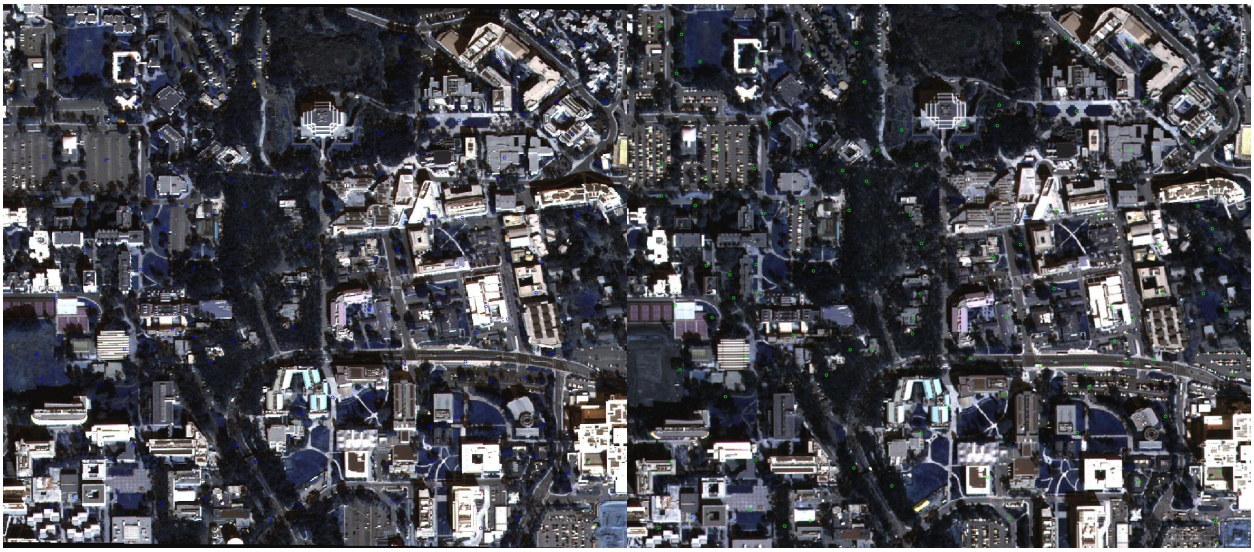


Figure 22: pair2 corners at $\sigma = 15$

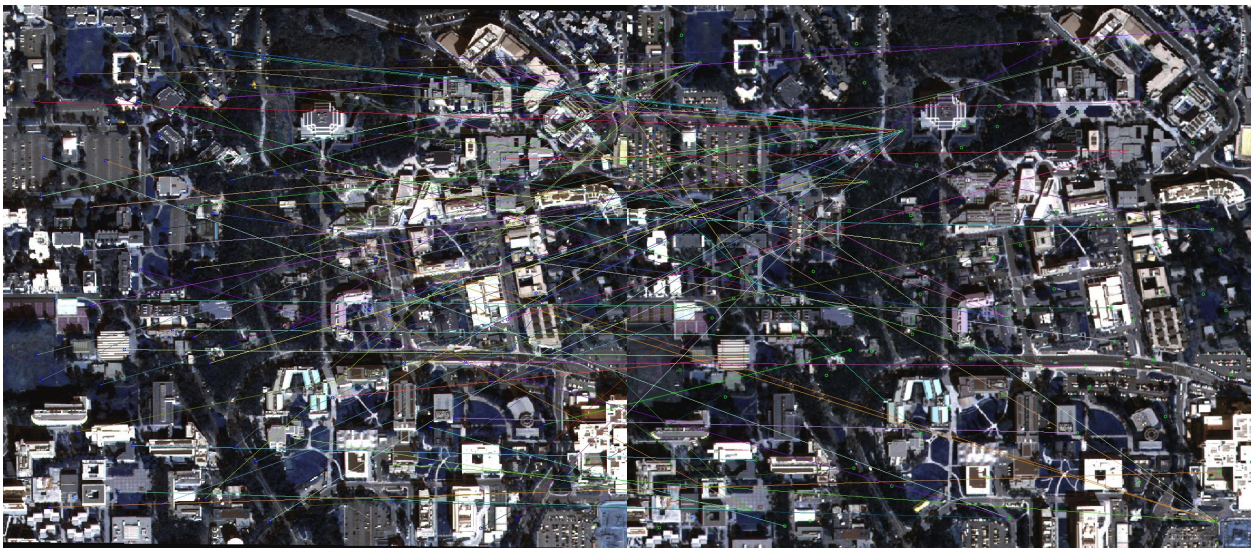


Figure 23: pair2 correspondences at $\sigma = 15$ using SSD

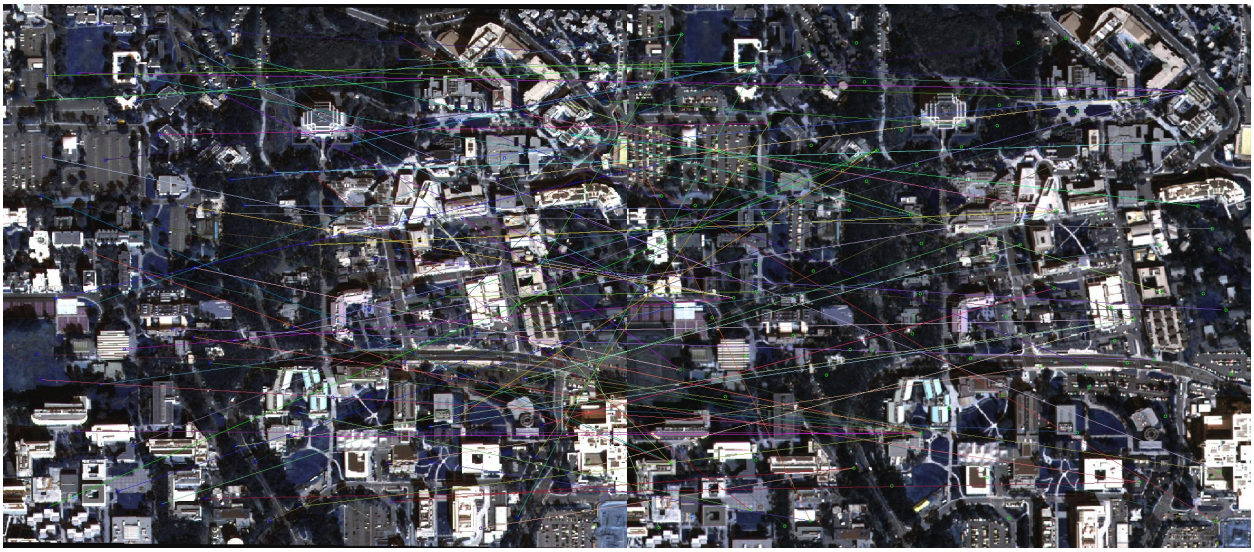


Figure 24: pair2 correspondences at $\sigma = 15$ using NCC

Figure 25: pair3 corners at $\sigma = 9$

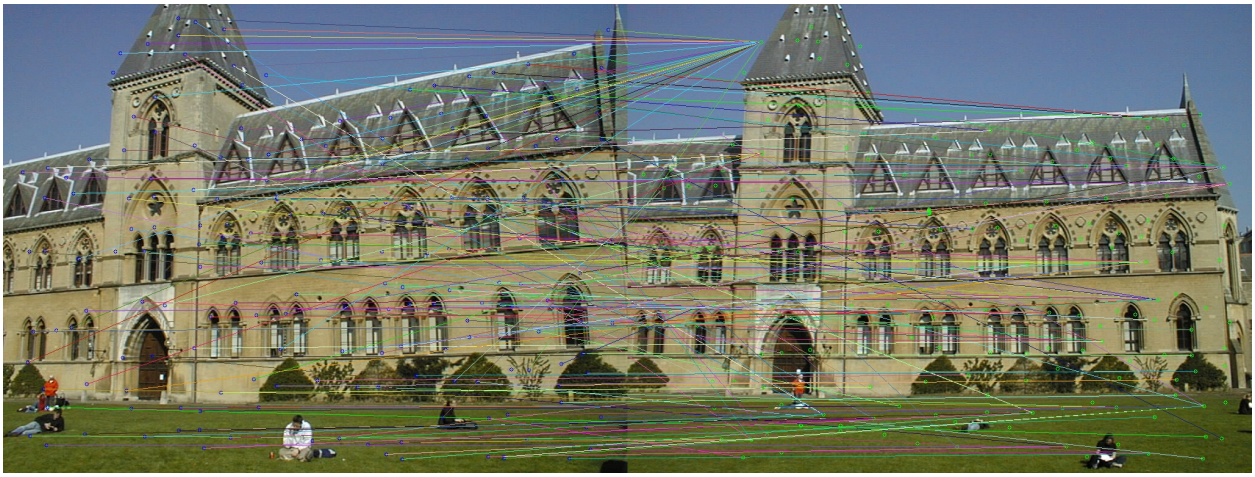


Figure 26: pair3 correspondences at $\sigma = 9$ using SSD

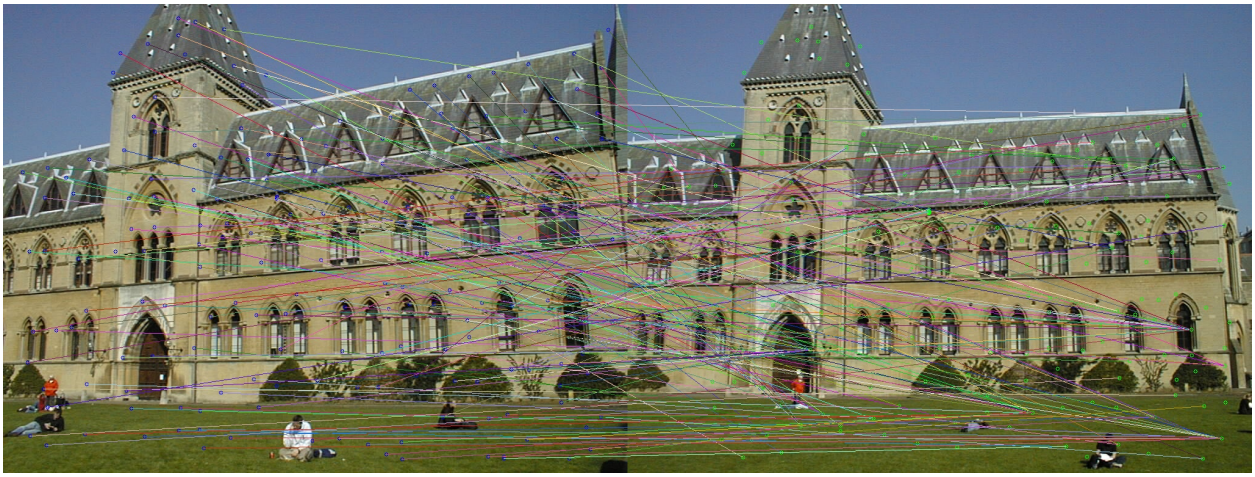


Figure 27: pair3 correspondences at $\sigma = 9$ using NCC

Figure 28: pair3 corners at $\sigma = 11$

Figure 29: pair3 correspondences at $\sigma = 11$ using SSD

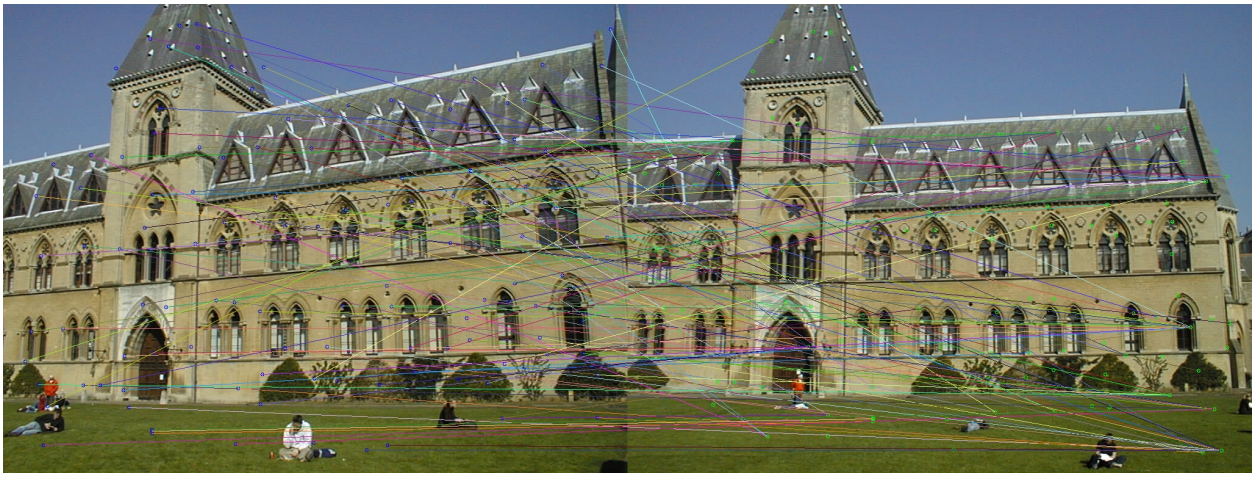


Figure 30: pair3 correspondences at $\sigma = 11$ using NCC

Figure 31: pair3 corners at $\sigma = 13$

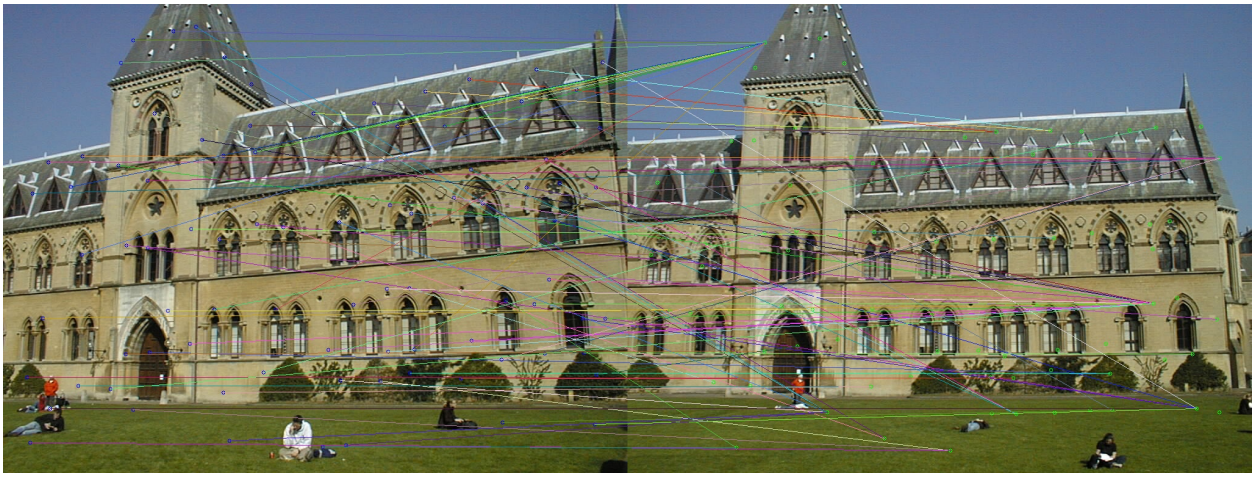


Figure 32: pair3 correspondences at $\sigma = 13$ using SSD

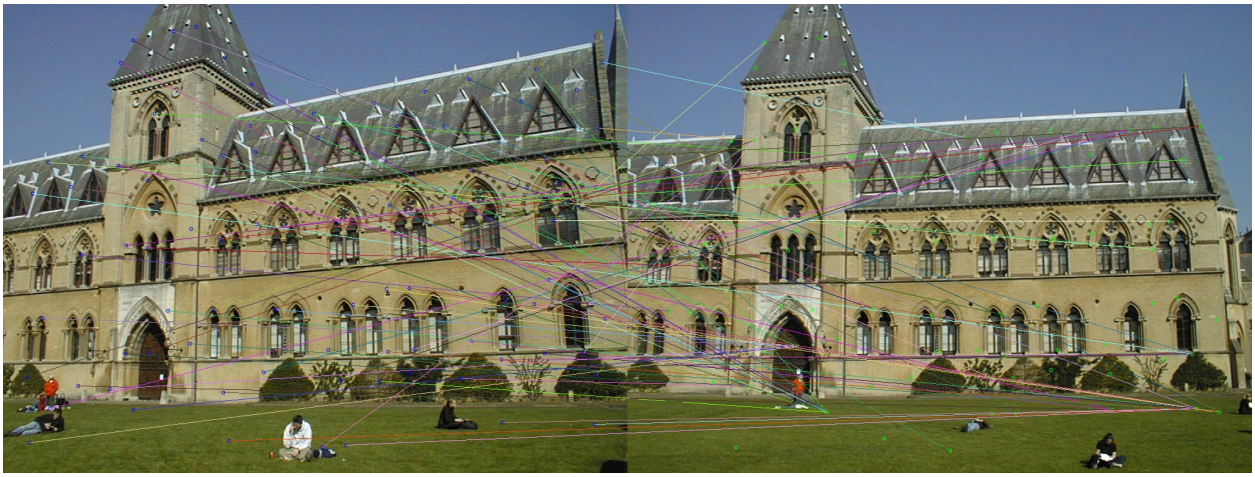


Figure 33: pair3 correspondences at $\sigma = 13$ using NCC

Figure 34: pair3 corners at $\sigma = 15$

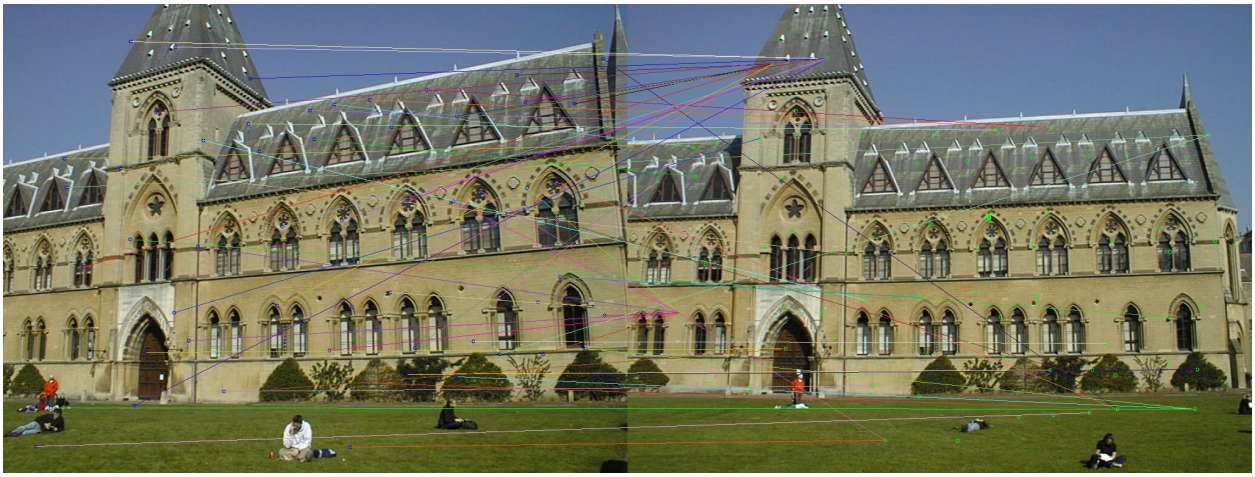


Figure 35: pair3 correspondences at $\sigma = 15$ using SSD

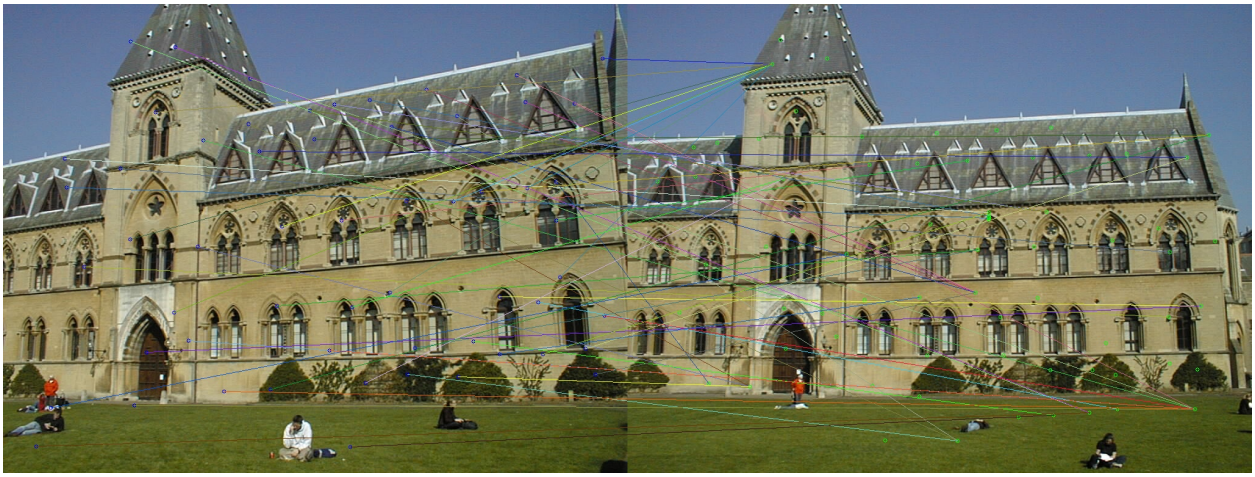


Figure 36: pair3 correspondences at $\sigma = 15$ using NCC

Figure 37: cars corners at $\sigma = 9$

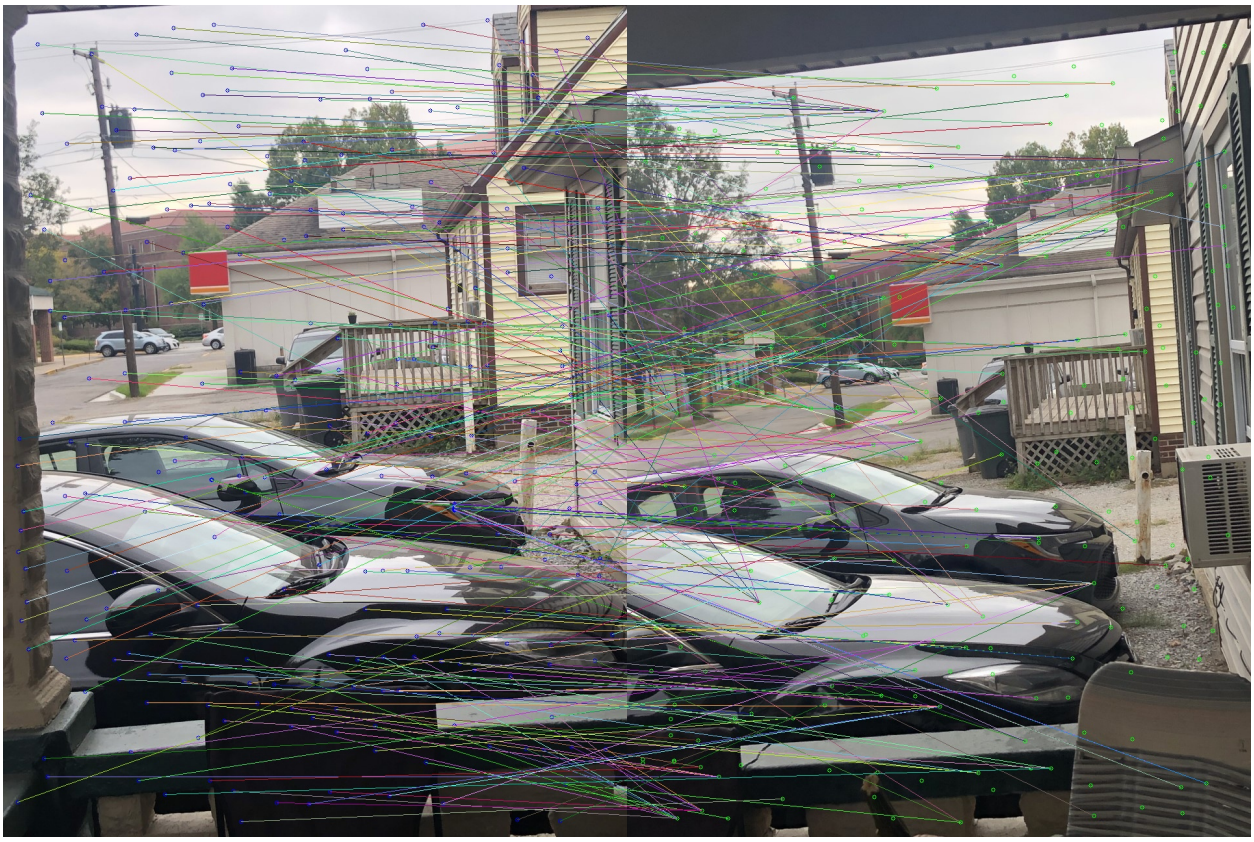


Figure 38: cars correspondences at $\sigma = 9$ using SSD

Figure 39: cars correspondences at $\sigma = 9$ using NCC

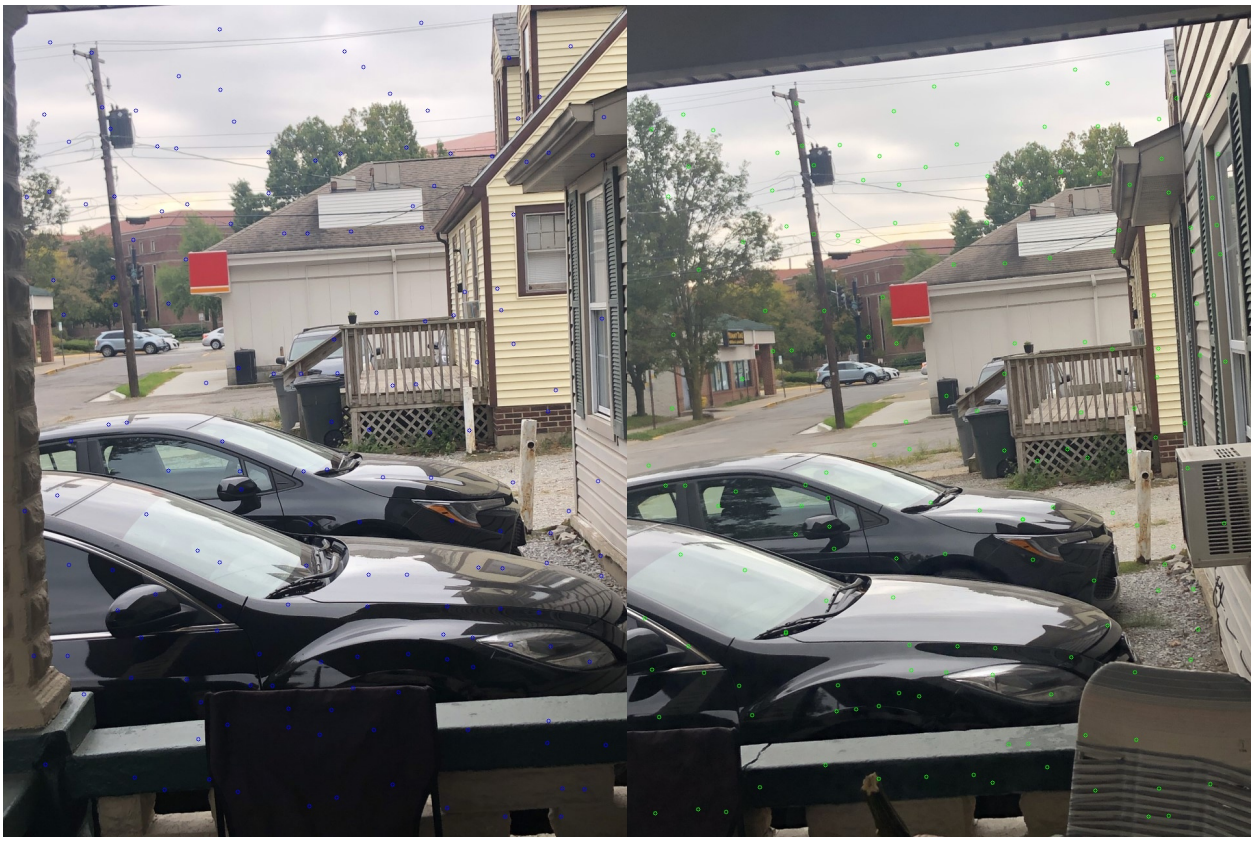


Figure 40: cars corners at $\sigma = 11$

Figure 41: cars correspondences at $\sigma = 11$ using SSD

Figure 42: cars correspondences at $\sigma = 11$ using NCC

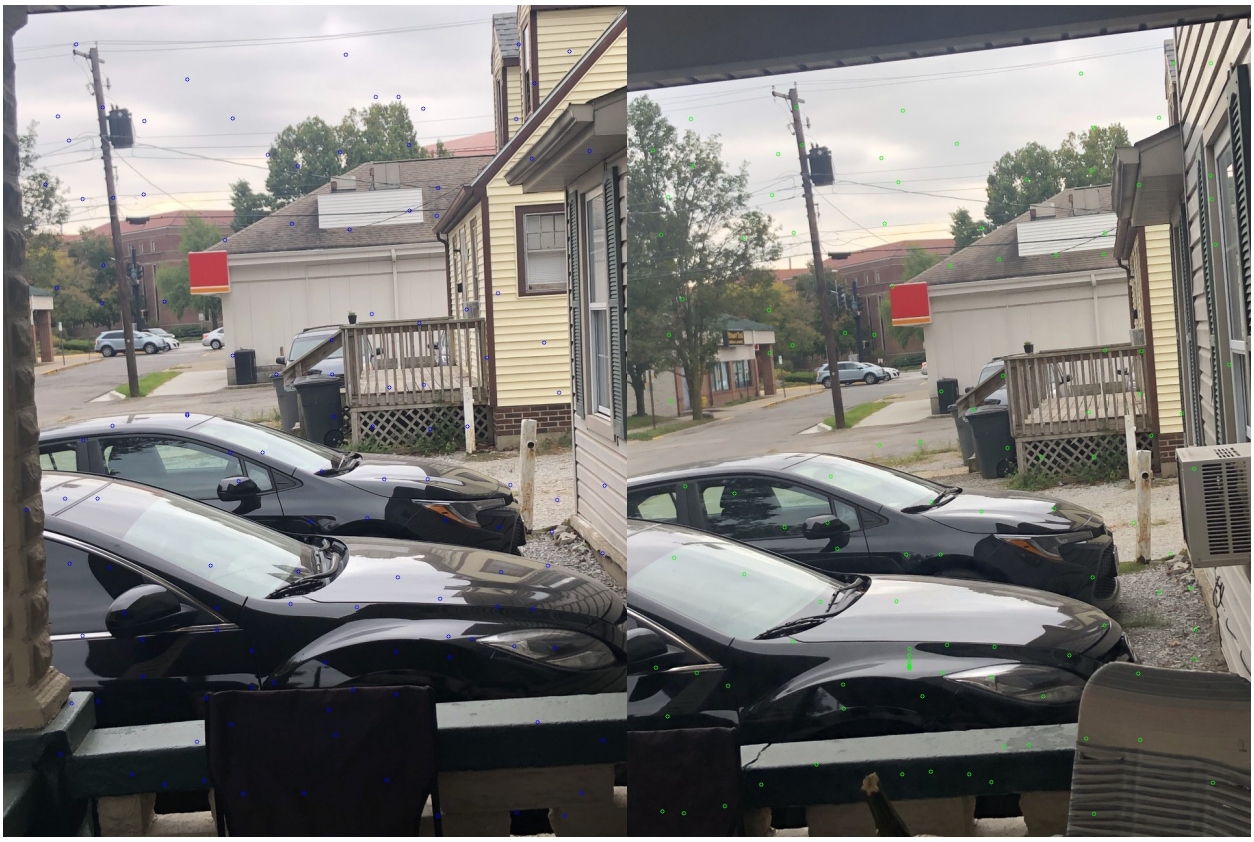


Figure 43: cars corners at $\sigma = 13$

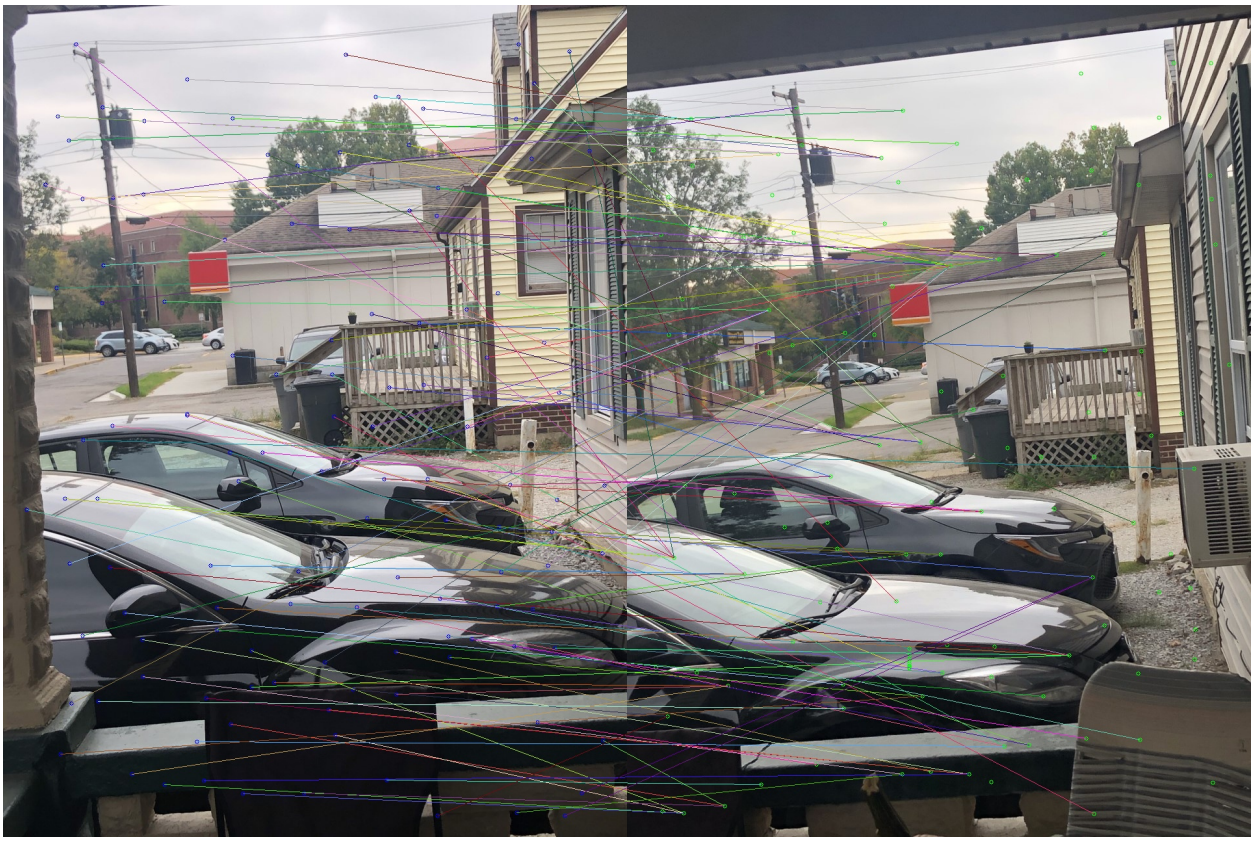


Figure 44: cars correspondences at $\sigma = 13$ using SSD

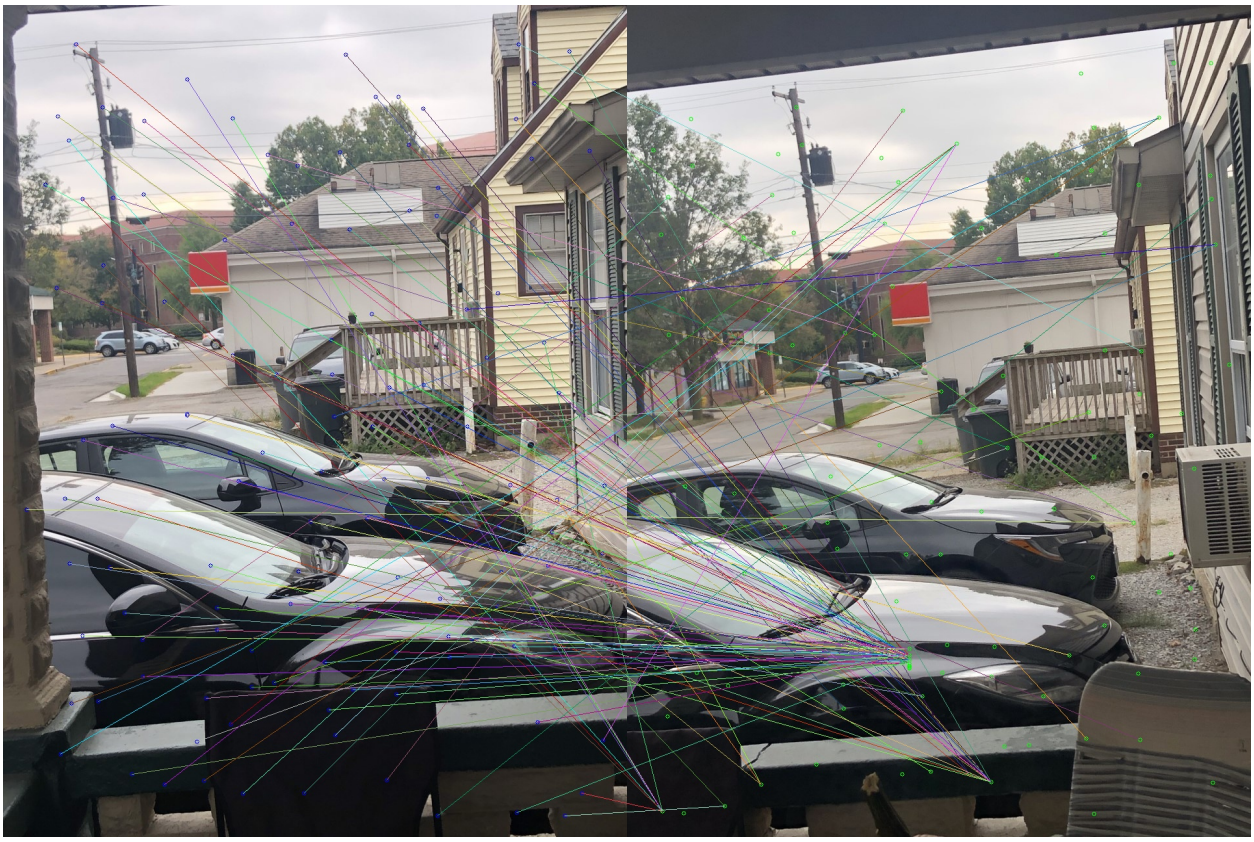


Figure 45: cars correspondences at $\sigma = 13$ using NCC

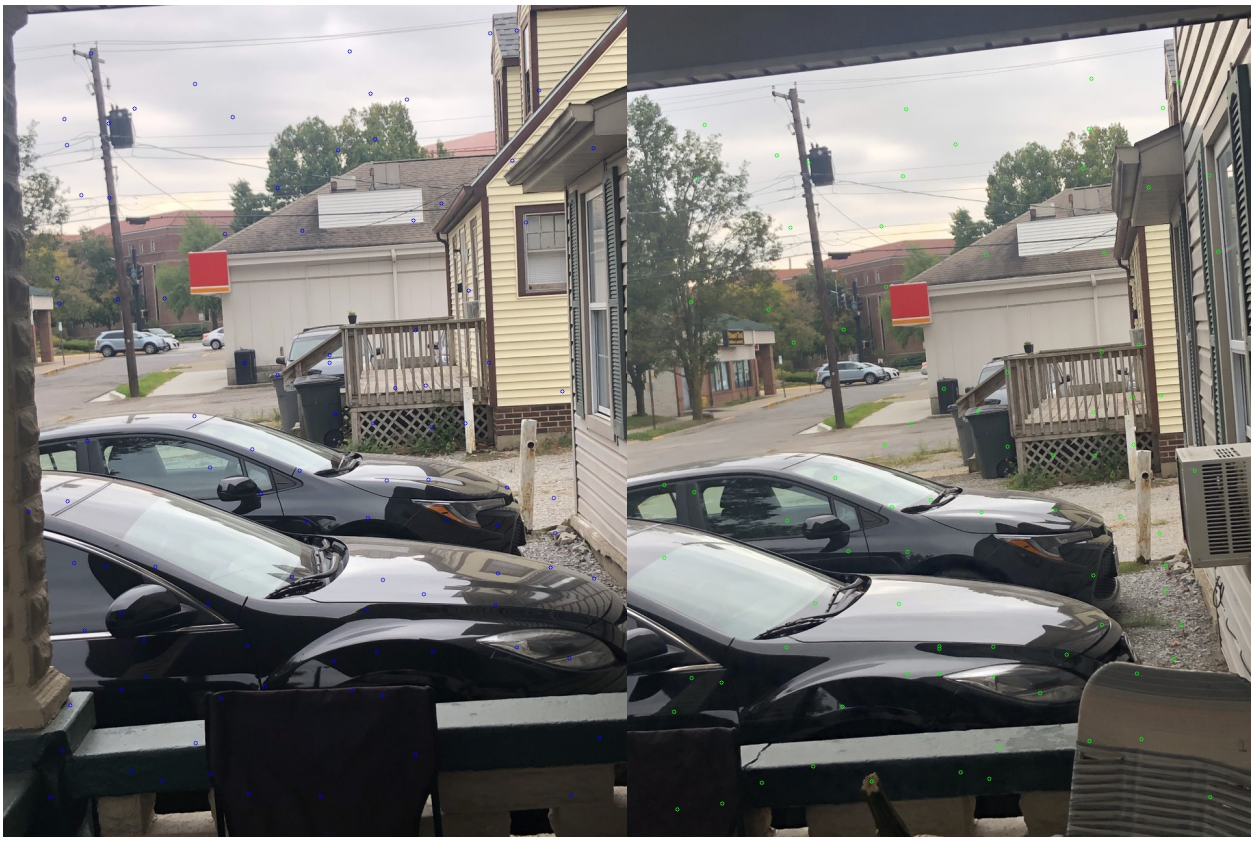


Figure 46: cars corners at $\sigma = 15$

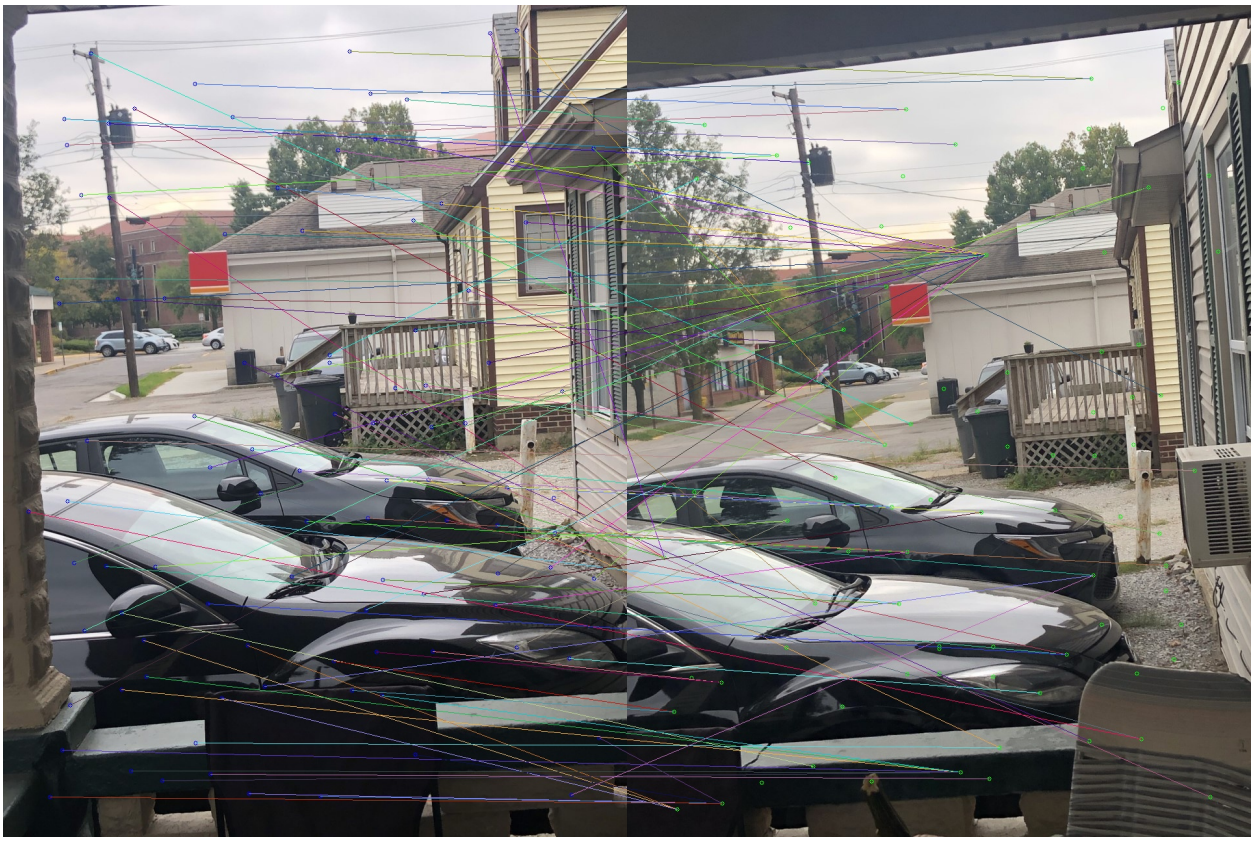


Figure 47: cars correspondences at $\sigma = 15$ using SSD

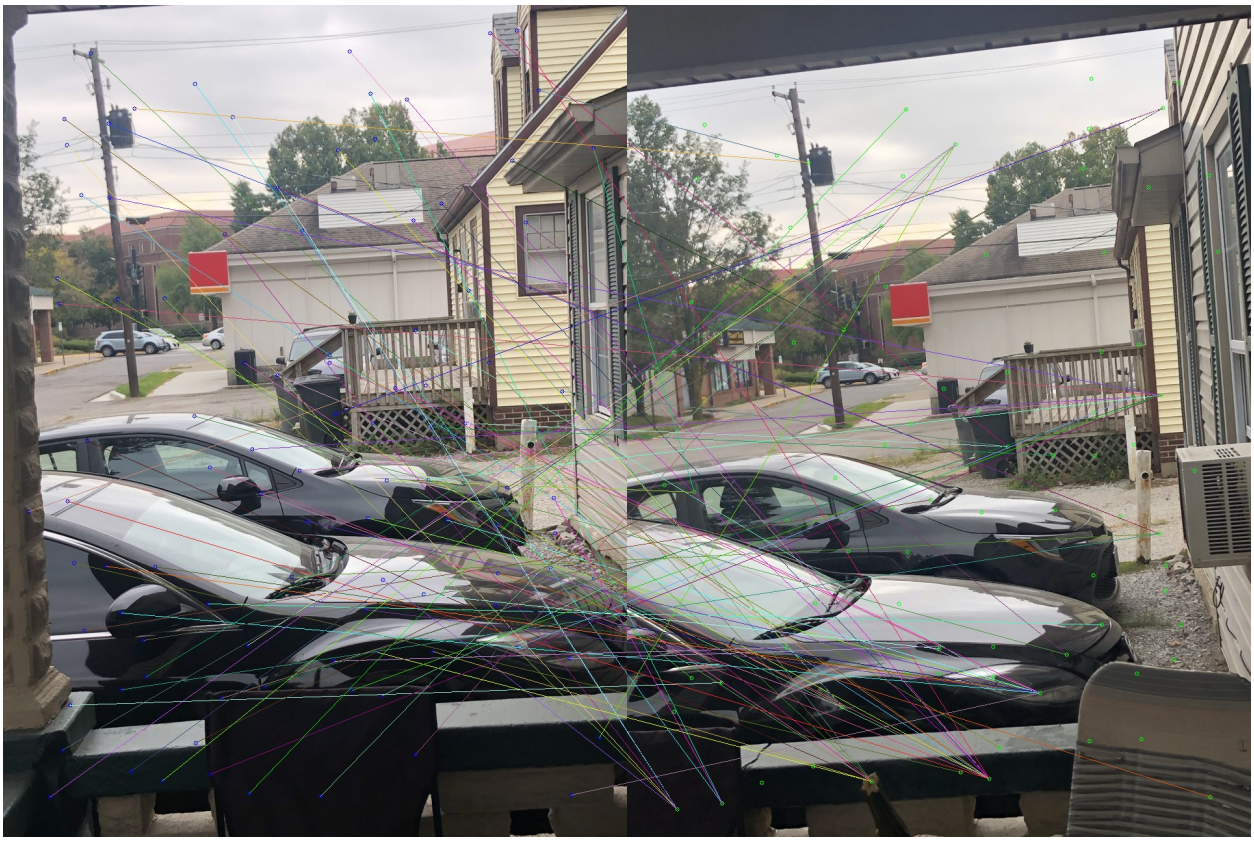


Figure 48: cars correspondences at $\sigma = 15$ using NCC

Figure 49: building corners at $\sigma = 9$

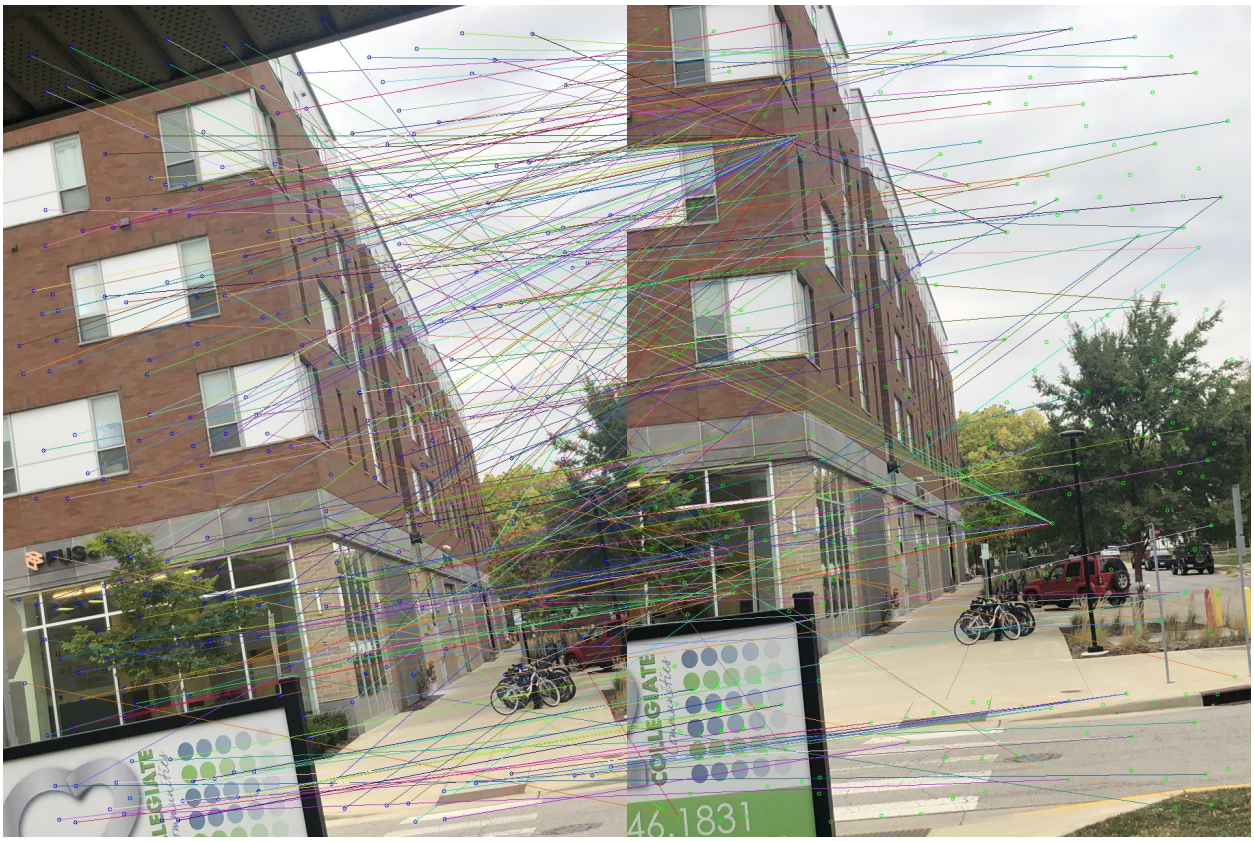


Figure 50: building correspondences at $\sigma = 9$ using SSD

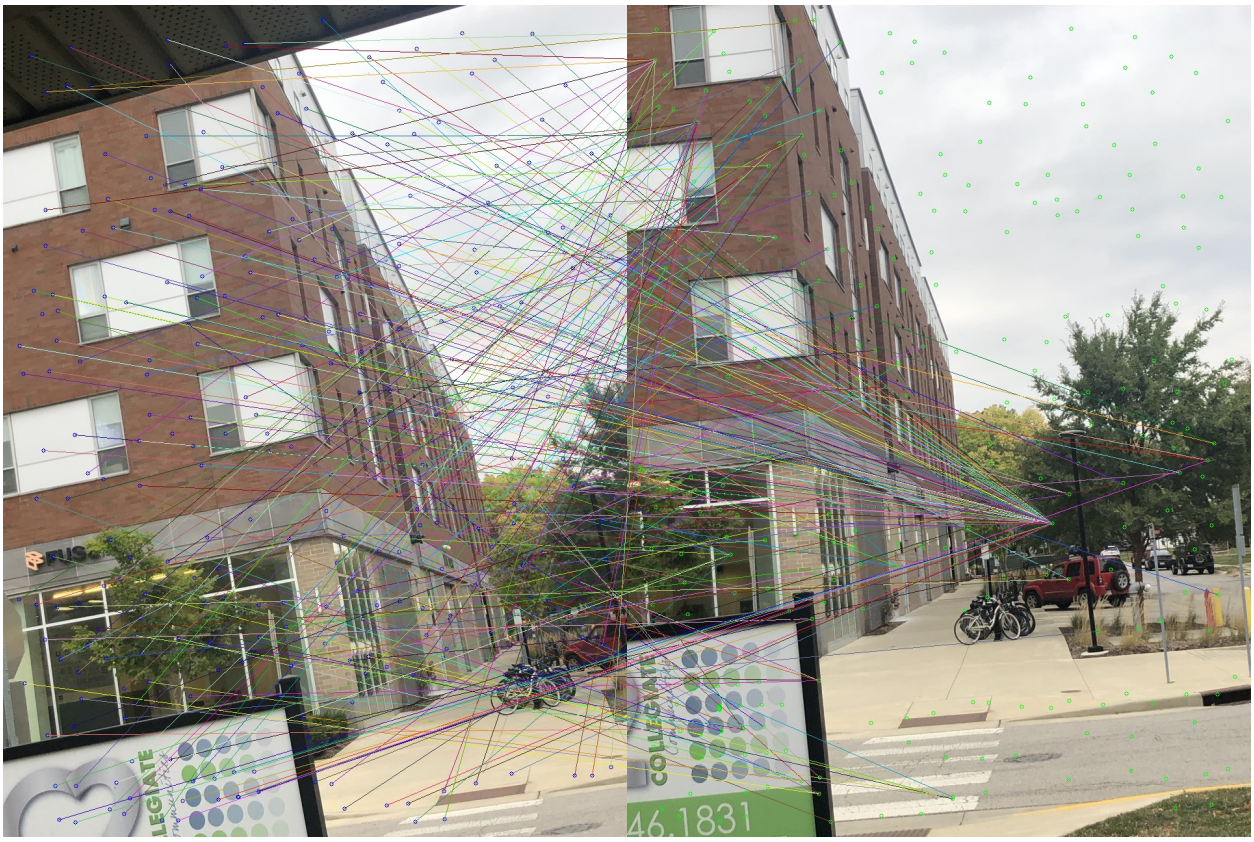


Figure 51: building correspondences at $\sigma = 9$ using NCC

Figure 52: building corners at $\sigma = 11$

Figure 53: building correspondences at $\sigma = 11$ using SSD

Figure 54: building correspondences at $\sigma = 11$ using NCC

Figure 55: building corners at $\sigma = 13$

Figure 56: building correspondences at $\sigma = 13$ using SSD

Figure 57: building correspondences at $\sigma = 13$ using NCC

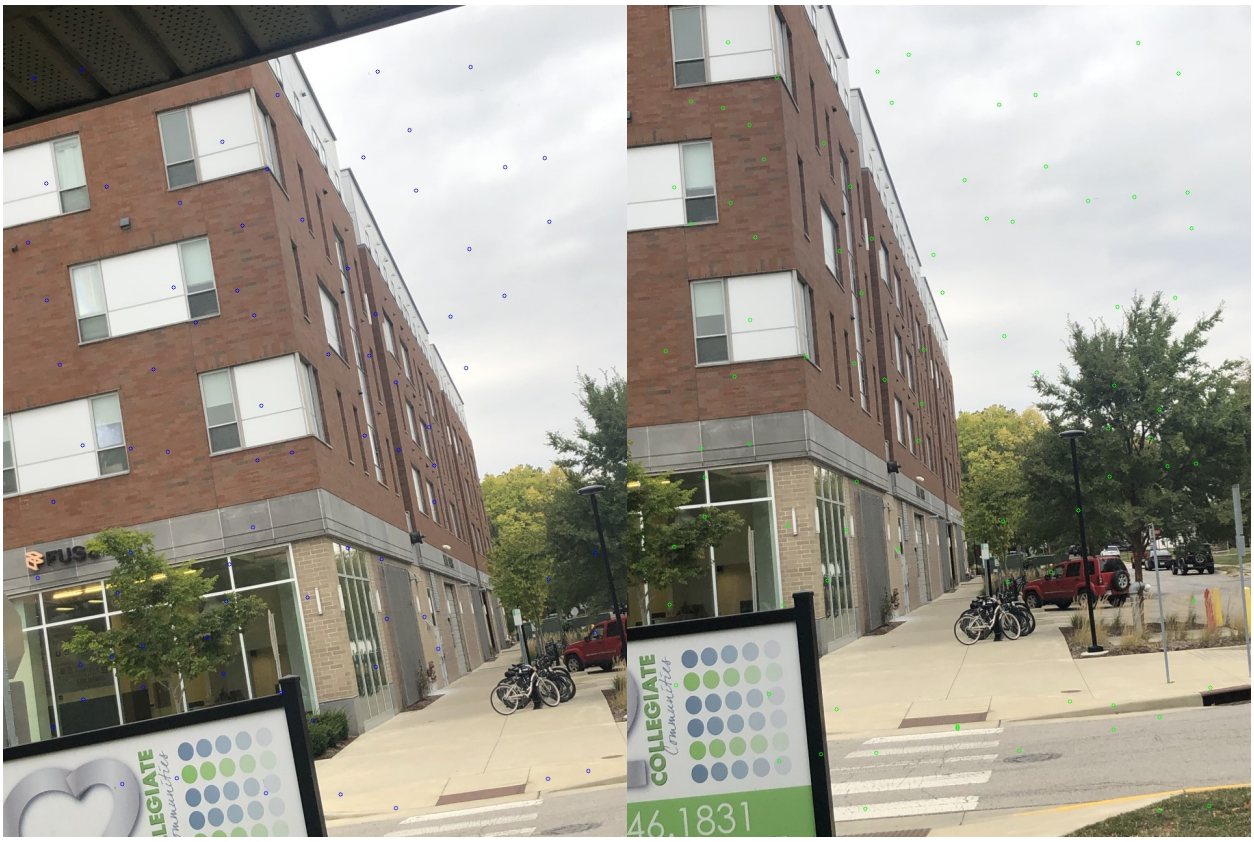


Figure 58: building corners at $\sigma = 15$

Figure 59: building correspondences at $\sigma = 15$ using SSD

Figure 60: building correspondences at $\sigma = 15$ using NCC

2.2 SIFT

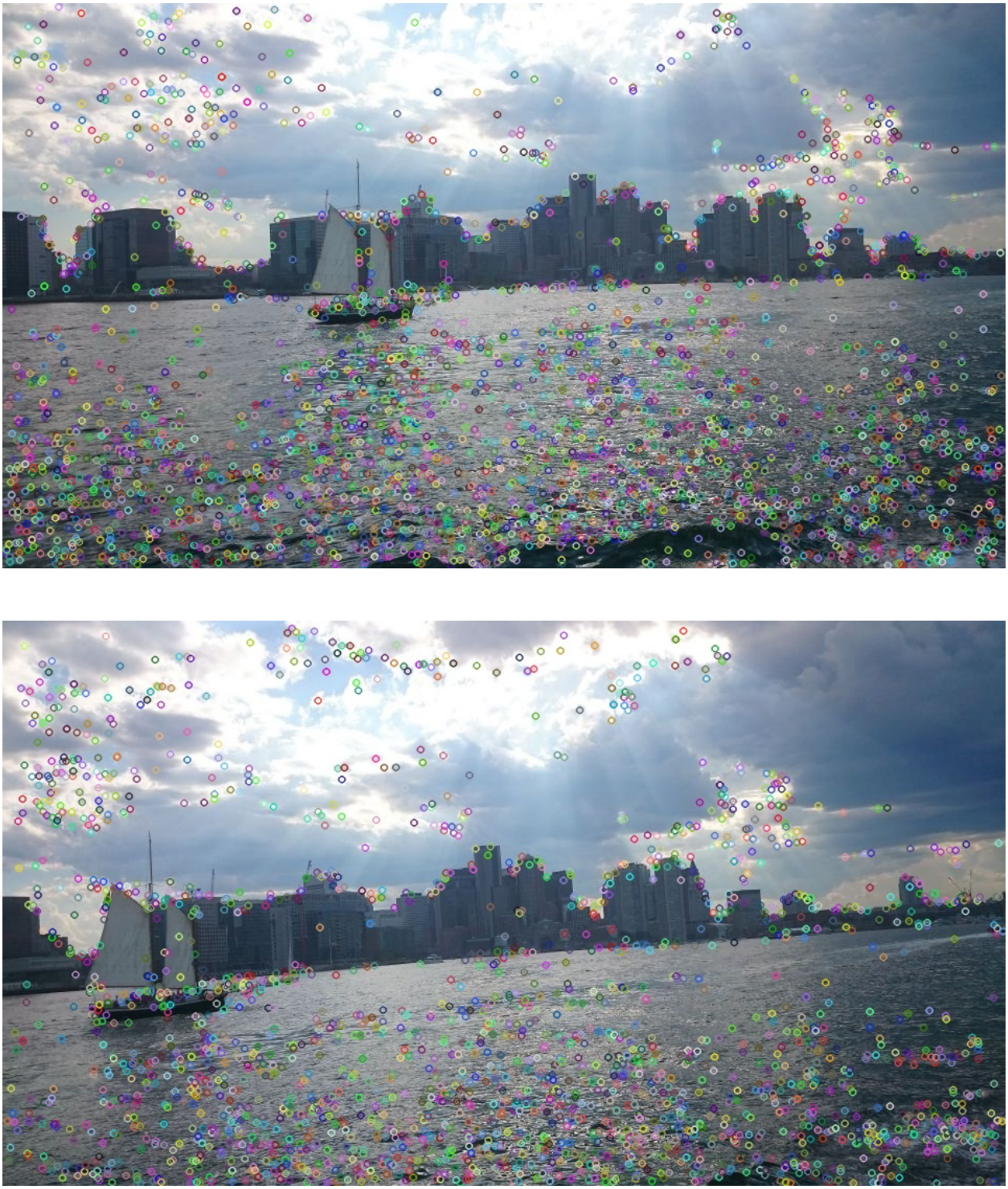
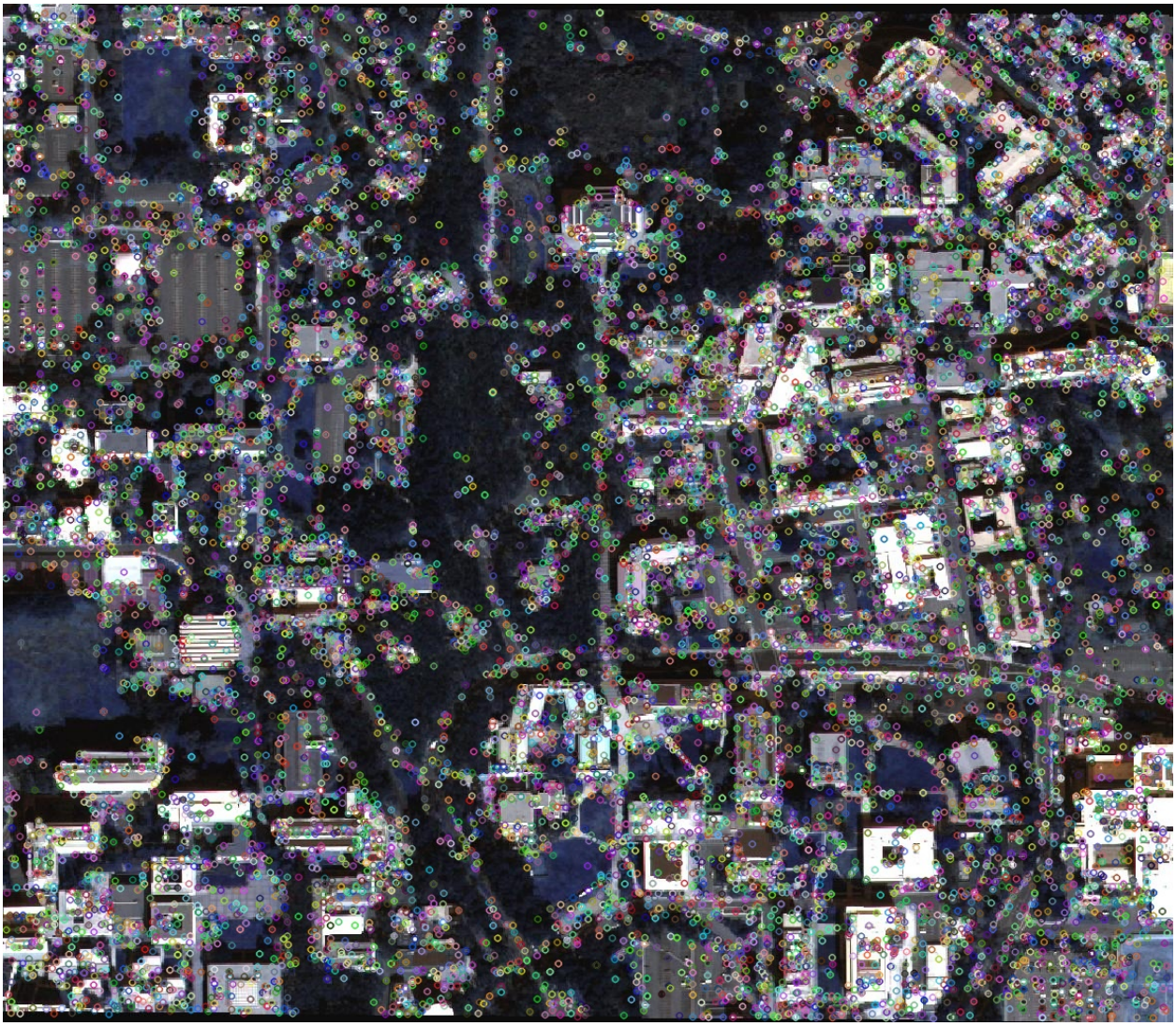


Figure 61: SIFT points for pair1

Figure 62: SIFT correspondences for pair1



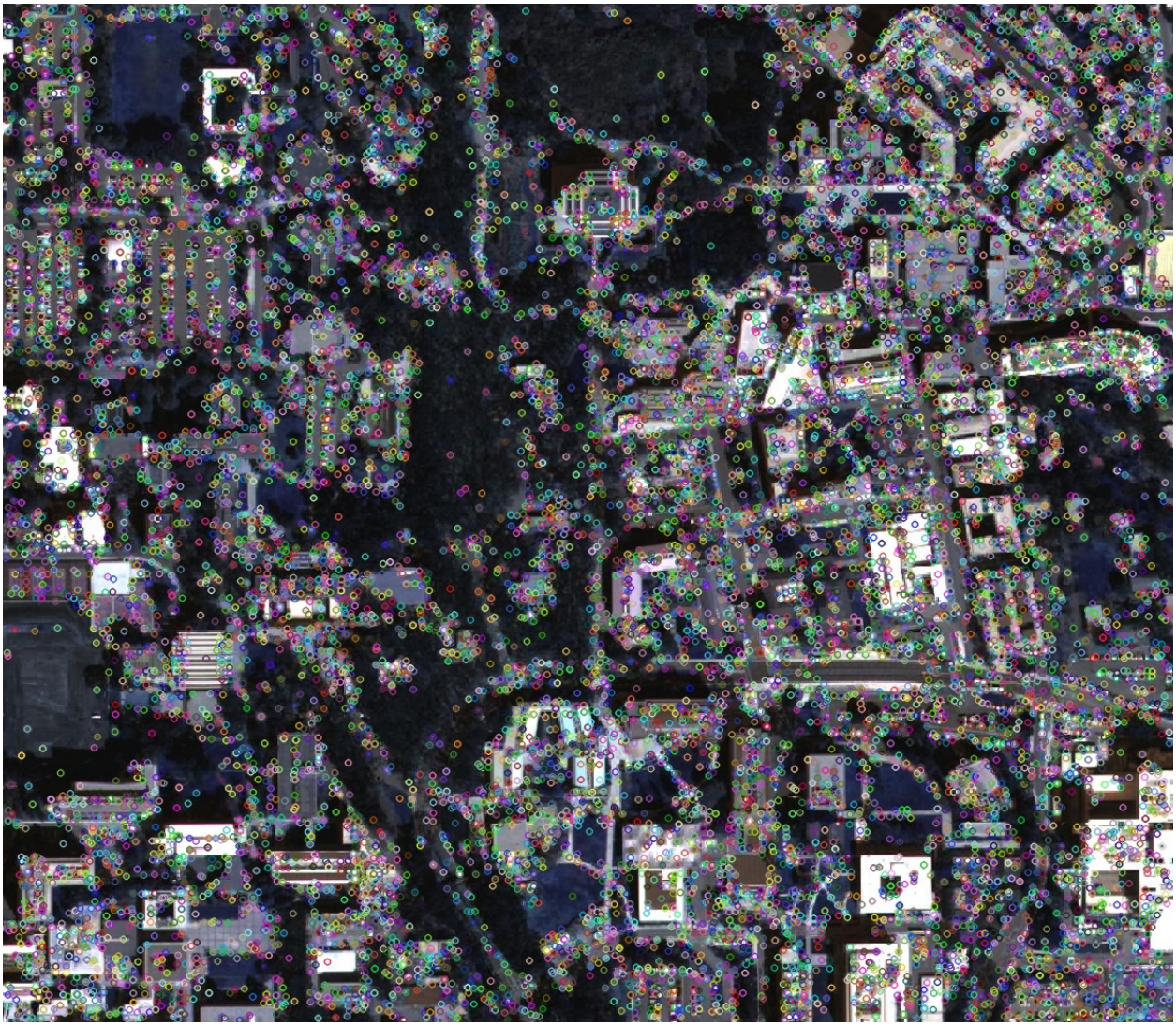


Figure 63: SIFT points for pair2

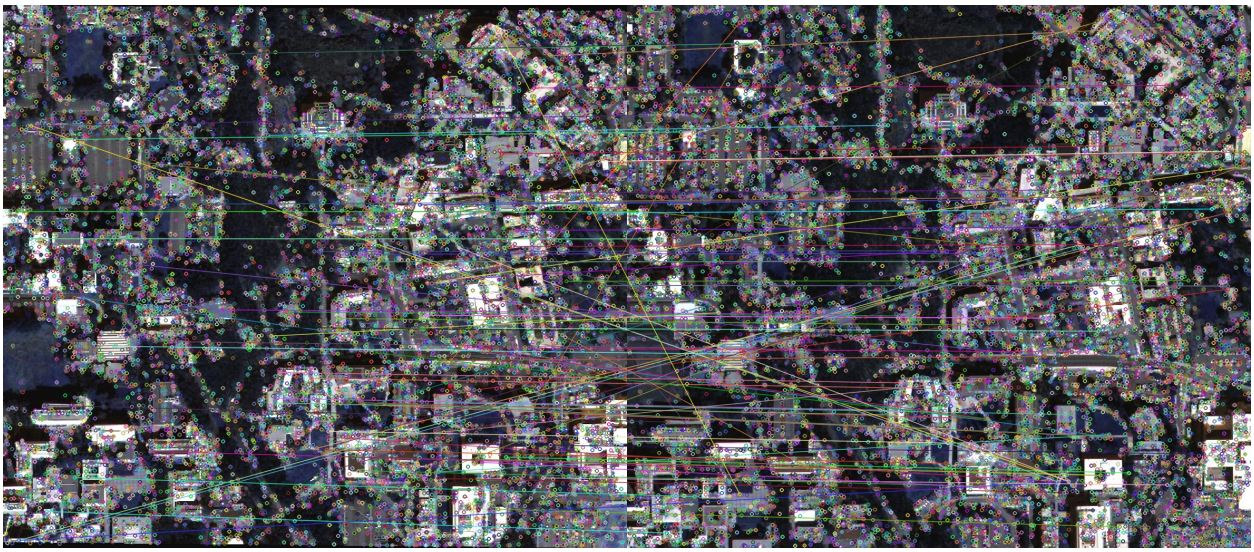


Figure 64: SIFT correspondences for pair2

Figure 65: SIFT points for pair3

Figure 66: SIFT correspondences for pair3

Figure 67: SIFT points for cars

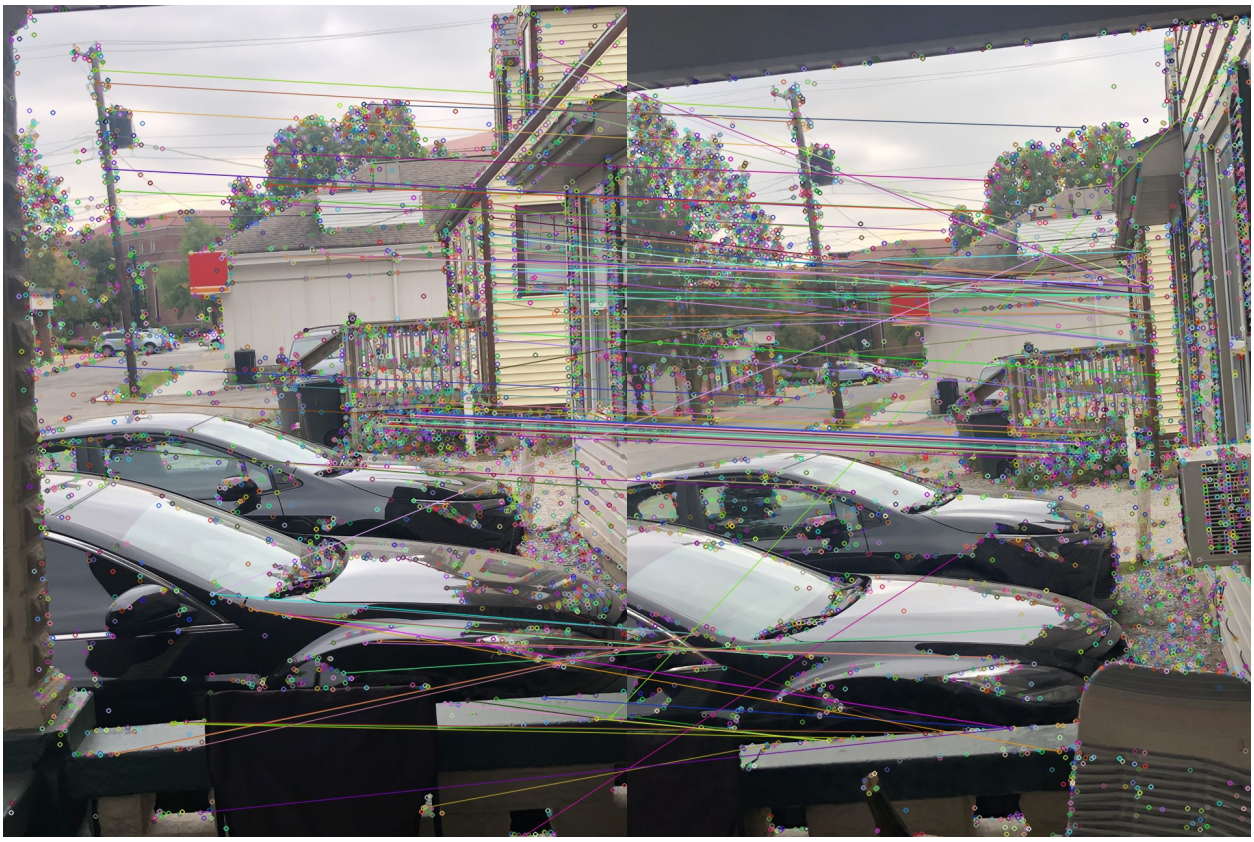


Figure 68: SIFT correspondences for cars

Figure 69: SIFT points for building

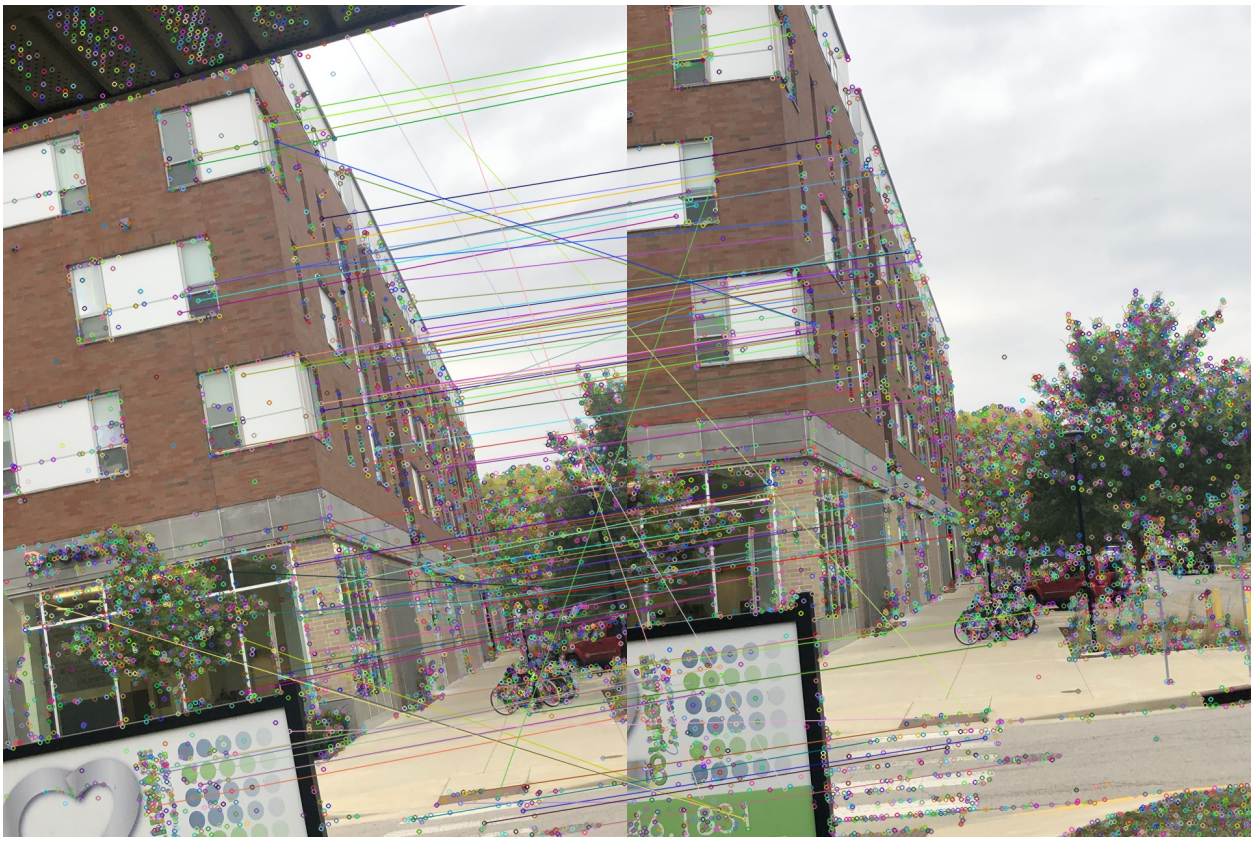


Figure 70: SIFT correspondences for building

3 Discussion

3.1 Effects of scale

Increasing the scale factor caused the Harris detector to identify fewer points as corners. This was an improvement, as most of the identified points were spurious or picking up on noise in the image. By increasing the scale factor, the amount of smoothing that each Haar wavelet does before computing the gradient is increased, which helps to eliminate some of the noise. Furthermore, a larger scale factor means the detector looks at a larger window when determining the singularity of the matrix. The larger the window, the more likely that a pixel identified as a corner is in fact a corner, which also improves the quality of results.

3.2 Effects of distance metrics

Two distance metrics were used here: SSD and NCC. While the results from these two metrics are quite similar, NCC often seemed to fixate on a single point in the second image, assigning it to several corners from the first image. This could perhaps be because it is normalized, and so doesn't distinguish between two separate correspondences that differ only in magnitude.

3.3 Harris Overall

The correspondence results were pretty good overall, mostly mapping regions of the sky to the sky and buildings to buildings. However, the corners themselves as identified by Harris were nowhere near as good

as those detected by SIFT. This is likely because Harris is a much more primitive algorithm that doesn't use the sophisticated thresholding and directional analysis built into SIFT.

3.4 SIFT

SIFT, as a much more sophisticated interest point detection algorithm, performed significantly better than Harris. The key points returned by SIFT contained far fewer spurious corners and matched much better across the two images.

4 Code

```
import cv2
import numpy as np
from math import ceil
from random import choice

def wavelets(sigma):
    size = ceil(sigma * 4)
    size += size % 2

    kernel = np.zeros((size, size))
    kernel[size // 2 :, :] = 1
    kernel[: size // 2, :] = -1

    return kernel, kernel.T

def normalize(img):
    norm = img.astype(np.float64)
    norm -= norm.min()
    norm /= norm.max()
    return norm


def _get_windows(img1, point1, img2, point2, radius=21):
    img1 = normalize(cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY))
    img2 = normalize(cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY))

    x1, y1 = point1
    x2, y2 = point2

    radius = min(
        x1, img1.shape[1] - x1, y1, img1.shape[0] - y1,
        x2, img2.shape[1] - x2, y2, img2.shape[0] - y2,
        radius)

    window1 = img1[y1 - radius : y1 + radius, x1 - radius : x1 + radius]
    window2 = img2[y2 - radius : y2 + radius, x2 - radius : x2 + radius]

    return window1, window2

def ssd(window1, window2):
```

```
    return np.sum((window1 - window2) ** 2)

def ncc(window1, window2):
    m1 = window1.mean()
    m2 = window2.mean()

    corr = np.sum((window1 - m1) * (window2 - m2))
    norm1 = np.sum((window1 - m1) ** 2)
    norm2 = np.sum((window2 - m2) ** 2)
    return corr / (np.sqrt(norm1 * norm2) + 0.001)

def harris(img, sigma):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    xdiff, ydiff = wavelets(sigma)

    Ix = normalize(cv2.filter2D(img, -1, xdiff))
    Iy = normalize(cv2.filter2D(img, -1, ydiff))

    size = 2 * int((5 * sigma) // 2) + 1
    window_sum = np.ones((size, size))

    dx2 = cv2.filter2D(Ix * Ix, -1, window_sum)
    dy2 = cv2.filter2D(Ix * Iy, -1, window_sum)
    dxy = cv2.filter2D(Ix * Iy, -1, window_sum)

    dets = dx2 * dy2 - dxy * dxy
    trace = (dx2 + dy2) ** 2
    k_vals = dets / (trace ** 2 + 0.001)
    k_thresh = np.percentile(k_vals, 70)

    R = dets - k_thresh * trace ** 2
    R[R < 0] = 0

    suppressed = np.zeros(R.shape)

    height, width = img.shape
    radius = size // 2
    for y in range(radius, height - radius):
        for x in range(radius, width - radius):
            window = R[y - radius: y + radius, x - radius: x + radius]
            if R[y, x] == window.max():
                suppressed[y, x] = R[y, x]

    ys, xs = np.where(suppressed > 0)
    return xs, ys


def correspondences(img1, points1, img2, points2, radius=21, metric=ssd):
    total = 1
    corrs = []
    for p1 in points1:
```

```
        bestPt = None
        bestDist = float('inf')
        for p2 in points2:
            dist = metric(*_get_windows(img1, p1, img2, p2, radius))
            if dist < bestDist:
                bestDist = dist
                bestPt = p2
        print(f'Matched {total} out of {len(points1)}')
        total += 1
        corrs.append((p1, bestPt, bestDist))
    return corrs

title = input('title? ')
ext = input('ext? ')

img1_orig = cv2.imread(f'hw4_Task2_Images/{title}1.{ext}')
img2_orig = cv2.imread(f'hw4_Task2_Images/{title}2.{ext}')
img2_orig = cv2.resize(img2_orig, (img1_orig.shape[1], img1_orig.shape[0]))

scales = [9, 11, 13, 15]

for scale in scales:
    img1 = np.copy(img1_orig)
    img2 = np.copy(img2_orig)

    corners1 = list(zip(*harris(img1, scale)))
    corners2 = list(zip(*harris(img2, scale)))

    for x, y in corners1:
        img1 = cv2.circle(img1, (x, y), 3, (255, 0, 0))
    for x, y in corners2:
        img2 = cv2.circle(img2, (x, y), 3, (0, 255, 0))

    stacked_ncc = np.hstack((img1, img2))
    stacked_ssd = np.copy(stacked_ncc)
    cv2.imwrite(f'corners_{title}_{scale}.jpg', stacked_ncc)

    width = img1.shape[1]

    for corr in correspondences(img1, corners1, img2, corners2, 21, metric=ncc):
        color = choice(range(256)), choice(range(256)), choice(range(256))
        cv2.line(stacked_ncc, corr[0], (corr[1][0] + width, corr[1][1]), color, 1)

    cv2.imwrite(f'correspondences_{title}_{scale}_ncc.jpg', stacked_ncc)

    for corr in correspondences(img1, corners1, img2, corners2, 21, metric=ssd):
        color = choice(range(256)), choice(range(256)), choice(range(256))
        cv2.line(stacked_ssd, corr[0], (corr[1][0] + width, corr[1][1]), color, 1)

    cv2.imwrite(f'correspondences_{title}_{scale}_ssd.jpg', stacked_ssd)
```