ECE 66100 Homework #4

by

Adrien Dubois (dubois6@purdue.edu)

October 23, 2025

Contents

1 Theoretical Question:

1.1 What is the theoretical reason we can use DoG?

We can represent an image smoothed by σ Gaussians as follows:

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

$$\text{Where: } g(x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{x^2+y^2}{2\sigma^2}}$$

After applying this Gaussian smoothing step, we can apply the following Laplacian to get the Laplacian of Gaussian (LoG):

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

Next, from the scale space theory for images, we know that:

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

Since

$$\nabla^2 ff(x, y, \sigma)$$

is the LoG of f(x,y) we can say that:

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

Therefore, while the LoG calculation requires an infinitesimally small change in σ, we can estimate this value with a relatively small $\delta\sigma$ instead. This gives a theoretically sound approximate value for the LoG through a difference of Gaussian by using $(\sigma - \delta\sigma)$ for the σ smoothing process.

1.2 Why is DoG computationally more efficient?

This implementation is more efficient since you work with a smaller sized operator for DoG than LoG: we don't need to find the second order partial derivatives by convolving the image with second order Sobel operators. This is due to the fact that you can perform the 2D smoothing for the DoG using a 1D smoothing window since the Gaussian is separable in x and y which makes the number of comparisons proportional to σ instead of σ^2.

2 Harris Corner Detector:

For this homework, we utilized the Harris corner detection algorithm first published in 1988 by Harris & Stephen [1]. My explanation for the algorithm's core logic is described below:

2.1 Harris Detection logic:

We first open the image and convert it to greyscale. We also need to normalize the pixel values to a scale of 0 to 1. This is essential as the Harris Corner Detector only works based on the intensity of light at a given pixel coordinate, not its actual color values.

Next, as long as there does not exist a direction with no intensity changes, a corner detection software should characterize pixels as corners in a manner that is invariant to the rotation of the image. The Harris corner detection algorithm does this by convolving the image with two sets of Haar filters. Examples of such filters for a sigma size of 0.8 is included below:

$$\text{n} = int(4 * sigma) = 4$$

$$\text{Haar size} = \text{smallest even value} \geq n \rightarrow \text{Haar size} = 4]$$

$$dx = \begin{bmatrix} -1. & -1. & 1. & 1. \\ -1. & -1. & 1. & 1. \\ -1. & -1. & 1. & 1. \\ -1. & -1. & 1. & 1. \end{bmatrix} \text{ and } dy = \begin{bmatrix} 1. & 1. & 1. & 1. \\ 1. & 1. & 1. & 1. \\ -1. & -1. & -1. & -1. \\ -1. & -1. & -1. & -1. \end{bmatrix}$$

We can then use this to construct the following matrix to determine if a pixel neighborhood is a corner:

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

If we let λ_1 and λ_2 and we assume that $\lambda_1 \geq \lambda_2$, then we can set a threshold on the ratio $\frac{\lambda_1}{\lambda_2}$ and points at which this ratio is greater than or equal to 0.1 become corner points. However, since finding

the eigenvalues of can be computationally ineficient, we use the equation proposed in Harris & Stephen [1]:

$$R = det(C) - k * Tr(C)^2, \text{ where } k \in [0.04, 0.06]$$

Lastly, we perform non-maximum suppression on the threshold response function R to only keep the best corner in an area of 20 σ where σ is a user chosen value on the scale space that we want to run our corner detection tool.

After getting corner points within a pair of images, we need to determine which corners best match other corner points. To do so, we use two different feature similarity metrics. Since there are no feature vectors to describe the points when using a Harris Corner Detector, we use a neighborhood of points are each corner point as the feature vectors and compute similarity scores between the intensity values at those coordinates. The equations for the two similarity distances utilized in this report are included below:

2.2 Harris implementation:

```
class Harris():
    def __init__(self, sigma):
        self.sigma = sigma
        haar_size = int(np.ceil(4*sigma))
        if haar_size % 2 == 1:
            # Odd -> add 1
            # Else this is already the largest even number > 4 sigma
            haar_size += 1
        self.haar_dx = np.hstack((-1*np.ones((haar_size, haar_size//2)), np.ones((
    haar_size, haar_size//2))))
        self.haar_dy = -1*self.haar_dx.copy().T

        # 5sigma by 5sigma neighborhood for calculating the C matrix
        neigh_size = int(np.ceil(5*sigma))
        self.summation_neighborhood = np.ones((neigh_size, neigh_size))


    def get_matching_interest_points(self, img1_path, img2_path, k, num_matches,
    similarity_measure="NCC"):
        """Takes in two images and returns a list of interest point matches

        Args:
            img1_path (string): string for the path to one image
            img2_path (string): string for the path of another view of the previous
    image
            k (scalar): Constant for corner threshold score k in [0.04, 0.06]
            topk: num of matches to keep in corner matching
        Returns:
            match_list (list): list of matching interest points on both images
        """
        # Open the image pairs
        img1 = np.array(cv2.imread(img1_path))
        img2 = np.array(cv2.imread(img2_path))

        # bring the images two greyscale and normalize pixel values to 0,1
        img1_bw = cv2.cvtColor(img1, cv2.COLOR_RGB2GRAY) / 255
        img2_bw = cv2.cvtColor(img2, cv2.COLOR_RGB2GRAY) / 255

        # Get the threshold response function to find the location of the corners
        threshold_response1 = self.find_corners(img1_bw, k)
        threshold_response2 = self.find_corners(img2_bw, k)

        # Convert threshold response to (x,y) coordinate locations of corners:
        # [(x1, y1), (x2, y2), ..] for non-zero values in the matrix
        corners1 = np.argwhere(threshold_response1)
        corners2 = np.argwhere(threshold_response2)

        print(len(corners1))
        # Apply feature similarity measure
        if similarity_measure == "NCC":
            point_matches = NCC(20, img1_bw, corners1, img2_bw, corners2, num_matches)
        elif similarity_measure == "SSD":
            point_matches = SSD(20, img1_bw, corners1, img2_bw, corners2, num_matches)
        else:
```

```
            raise ValueError("Options for the similarity measure are: NCC and SSD")

        return point_matches

    def find_corners(self, img_bw, k):
        # Apply haar filters through convolution
        dx = cv2.filter2D(img_bw, -1, self.haar_dx)
        dy = cv2.filter2D(img_bw, -1, self.haar_dy)

        # Get second derivatives, these will be applied along a 5sigmax5sigma neighborhood
        dx2, dy2, dxdy = dx*dx, dy*dy, dx*dy

        # Get second degree summations along the neighboorhood area
        sum_dx2 = cv2.filter2D(dx2, -1, self.summation_neighborhood)
        sum_dy2 = cv2.filter2D(dy2, -1, self.summation_neighborhood)
        sum_dxdy = cv2.filter2D(dxdy, -1, self.summation_neighborhood)
        # sum_dydx = cv2.filter2D(dydx, -1, self.summation_neighborhood)

        # Calculate the trace and determinant for the threshold ratio
        trace = sum_dx2 + sum_dy2
        det = sum_dx2*sum_dy2 - (sum_dxdy)**2

        # Threshold ratio following Harris & Stephens 1988:
        R = det - k*(trace**2)

        suppression_map = maximum_filter(R, size=int(20*self.sigma))
        threshold_response = np.where(R == suppression_map, R, 0)

        return threshold_response
```

3 SSD:

The SSD score is essentially a euclidean distance metric between the intensity values within the two corner neighborhoods. We will therefore keep the topk best matches as those with the lowest SSD scores.

$$SSD = \sum_i \sum_j (f_1(i,j) - (f_2(i,j))$$

```
def get_masked_corners(corners, area_bound, img):
    return corners[(corners[:, 0] >= area_bound) & (corners[:, 0] <= img.shape[0] - area_bound) &
                                (corners[:, 1] >= area_bound) & (corners[:, 1] <= img.shape[1] - area_bound)]

def SSD(m, img1, corners1, img2, corners2, num_matches):
    area_bound = m+1
    # Remove boundary points that are outside of the neighborhood search area
    # Make sure to check for the boundaries on all sides of the images
    # Corners is a list of points: [(x1,y1), (x2,y2), (x3,y3), ...]
    masked_corners1 = get_masked_corners(corners1, area_bound, img1)
    masked_corners2 = get_masked_corners(corners2, area_bound, img2)

    matching_points = []
    for coord1 in masked_corners1:
        # Get (m+1) by (m+1) area in the BW image around that corner point
        neighborhood1 = img1[coord1[0]-(area_bound):coord1[0]+(area_bound), coord1[1]-(area_bound):coord1[1]+(area_bound)]
        for coord2 in masked_corners2:
            neighborhood2 = img2[coord2[0]-(area_bound):coord2[0]+(area_bound), coord2[1]-(area_bound):coord2[1]+(area_bound)]

            # Calculate the sum of squared distances and add the result to a list to later get the topk
            ssd = np.sum((neighborhood1 - neighborhood2)**2)
            matching_points.append([ssd, [coord1, coord2]])

    # Get the top num_matches corner matches (with the highest SSD values first)# Convert to array for easier manipulation
    matching_points = np.array(matching_points, dtype=object)
```

```
    # Sort matching points by SSD (smallest first)
    sorted_matches = sorted(matching_points, key=lambda x: x[0])

    # To store the top correspondences without repeats
    topk_matches = []
    used_coords1 = set()
    used_coords2 = set()

    # Iterate over sorted matches and select the top num_matches with unique points
    for match in sorted_matches:
        coord1, coord2 = match[1]

        # Ensure no coordinate from coord1 or coord2 is repeated
        if tuple(coord1) not in used_coords1 and tuple(coord2) not in used_coords2:
            topk_matches.append([coord1, coord2])
            used_coords1.add(tuple(coord1))
            used_coords2.add(tuple(coord2))

        # Stop when we have enough matches
        if len(topk_matches) == num_matches:
            break
    return topk_matches
```

4 NCC:

The NCC scores will have values from 0 to 1 where 1 indicates a perfect match. We will therefore keep the topk best matches and present them on the images in the results section.

$$NCC = \frac{\sum\sum(f_1(i,j) - m_1)(f_2(i,j) - m_2)}{\sqrt{[\sum\sum f_1(i,j) - m_1]\,[\sum\sum f_2(i,j) - m_2]}}$$

```
ef NCC(m, img1, corners1, img2, corners2, num_matches):
    area_bound = m+1
    # Remove boundary points that are outside of the neighborhood search area
    # Make sure to check for the boundaries on all sides of the images
    # Corners is a list of points: [(x1,y1), (x2,y2), (x3,y3), ...]
    masked_corners1 = get_masked_corners(corners1, area_bound, img1)
    masked_corners2 = get_masked_corners(corners2, area_bound, img2)

    matching_points = []
    for coord1 in masked_corners1:
        # Get (m+1) by (m+1) area in the BW image around that corner point
        neighborhood1 = img1[coord1[0]-(area_bound):coord1[0]+(area_bound), coord1[1]-(
    area_bound):coord1[1]+(area_bound)]

        for coord2 in masked_corners2:
            neighborhood2 = img2[coord2[0]-(area_bound):coord2[0]+(area_bound), coord2
    [1]-(area_bound):coord2[1]+(area_bound)]
            # sum(sum((f1 - mu1) * f2 - mu2)
            numerator = np.sum((neighborhood1 - np.mean(neighborhood1)) * (neighborhood2
     - np.mean(neighborhood2)))

            # sum(sum((f1 - mu1)^2)) * sum(sum((f2 - mu2)^2))
            denom1 = np.sum((neighborhood1 - np.mean(neighborhood1))**2)
            denom2 = np.sum((neighborhood2 - np.mean(neighborhood2))**2)

            # sqrt(denom2 * denom2)
            ncc = numerator / ((denom1 * denom2)**(1/2))
            matching_points.append([ncc, [coord1, coord2]])

    # Get the top num_matches corner matches (with the highest SSD values first)#
    Convert to array for easier manipulation
    matching_points = np.array(matching_points, dtype=object)

    # Sort matching points by SSD (smallest first)
    sorted_matches = sorted(matching_points, key=lambda x: x[0], reverse=True)

    # Get the top correspondences, without repeats in both the coord1 and coord2 values
```

```
    topk_matches = []
    used_coords1 = set()
    used_coords2 = set()

    for match in sorted_matches:
        coord1, coord2 = match[1]

        if tuple(coord1) not in used_coords1 and tuple(coord2) not in used_coords2:
            topk_matches.append([coord1, coord2])
            used_coords1.add(tuple(coord1))
            used_coords2.add(tuple(coord2))

        if len(topk_matches) == num_matches:
            break
    return topk_matches
```

4.1 SIFT:

4.2 Gaussian Pyramids

To understand the SIFT algorithm, we must first review Gaussian Pyramids. We can create such pyramids by applying σ-Gaussian smoothing to the image. If we consider the bottom of the pyramid as an (N,N) sized image smoothed by a factor of σ, then smoothing the image by a factor of 2σ creates a down-sampled version of the image by a factor of two: the (N,N) size image will be down-sampled to (N/2, N/2). This process can be repeated with 4σ to make a (N/4, N/4) image etc. These down-sampled versions of the image can then be represented in the pyramidal structure shown below:

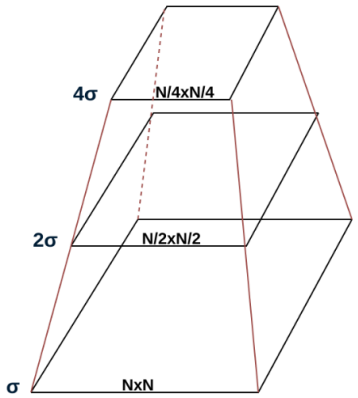

Figure 1: Example of the Gaussian pyramid used for the SIFT algorithm.

We can then apply this concept to the Laplacian of Gaussians. We can therefore create sets of scale pyramids for each octave of σ values (one octave is from $k\sigma$ to $2k\sigma$, $k \in \mathbb{Z}^+$). Finding the scale representations for between small changes in σ values will then allow us to estimate the Difference of Gaussian pyramidal representations for images.

4.3 SIFT Logic

The follow 5 steps explain the process of applying the SIFT keypoint matching algorithm to a pair of images:

1. Find all the local extrema in the DoG pyramid. These will be the maximum or minimum values of the following second-order partial:

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

   For each local minimum or maximum point, we also associate the following values:

   - The 8 points in its immediate 3x3 neighborhood.

- The 9 points in the in the 3x3 neighborhood of the DoG that is in the next level scale space.
- The 9 points in the 3x3 neighborhood of the DoG that is in the level just below in the scale space pyramid.

2. We now require more accurate estimates for the discrete extrema points as you move up and down in the scale space pyramid. These can be calculated with sub-pixel level precision using the second-order partial at sampling points in the pyramid. We get these points by evaluating the Jacobean and Hessian matrices at x_0 and get the true locations through the following formula:

$$\vec{x} = -H^{-1}(\vec{x_0})J(\vec{x_0})$$

3. We can then apply a threshold on the DoG value of the extrema $D(x, y, \sigma)$. We typically rejected extrema if $|D(\vec{x}| < 0.03$.

4. Next, we associate with each valid extrema a dominant local orientation. The goal of this is to make the algorithm invariant to in-plane rotations of the image. To calculate the dominant local direction, we compute the gradient magnitude and orientation:

   - Gradient Magnitude: $m(x, y) = \sqrt{|ff(x+1, y, \sigma) = ff(x, y, \sigma)|^2 + |ff(x, y+1, \sigma) - ff(x, y, \sigma)|^2}$
   - Gradient Orientation: $\theta(x, y) = \arctan \frac{ff(x,y+1,\sigma) - ff(x,y,\sigma)}{ff(x+1,y,\sigma) - ff(x,y,\sigma)}$

   We can combine these two metrics in a histogram to find the dominant orientation by the bin with the largest amount of sample within a 10 deg range.

5. Lastly, we associate a 128-dimensional SIFT descriptor vector with each retained extrema from the DoG pyramid. To do so, we apply the same gradient calculations as Step 4, except we take them in respect to the dominant local orientation. This creates an 8-bin histogram for each of the 16pixels in the 4x4 block around the extrema. We combine these histograms together to create the 128-dimensional embedding vector.

4.4 SURF

SURF works similarly to SIFT except we no longer place interest points at the extrema of the second degree partial, we now place the interest points at the maximum of the determinant of the Hessian matrix:

$$\begin{bmatrix} \frac{\partial^2}{\partial x^2} ff(x, y, \sigma) & \frac{\partial^2}{\partial x \partial y} ff(x, y, \sigma) \\ \frac{\partial^2}{\partial x \partial y} ff(x, y, \sigma) & \frac{\partial^2}{\partial y^2} ff(x, y, \sigma) \end{bmatrix}$$

Additionally, while we no longer down-sample the image in SURF, we can maintain the high computational speed by utilizing integral versions of the images where:

$$I(x, y) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} f(i, j)$$

Using this representation, for any size of the σ operator, we can calculate the derivative with just six additions of the $I(x, y)$ values. This is due to the fact that, while the size of the image remains the same, the size of the box filters used to calculate the Hessian matrix become larger as the operator increases.

After having calculated the Hessian matrix, you need the first-order partials to get the dominant direction of the interest point. We can calculate these using a Haar filter as explained in Section 2.1. We can find the dominant direction by weighting the d_x and d_y values with a 2σ Gaussian centered at the interest point and finding which weighted (d_x, d_y) values in a 60 deg cone yield the largest resulting vectors.

Afterwards, we create a 20σ by 20σ neighborhood around the point that is oriented along the dominant direction. We can divide this neighborhood into 4x4 scares sampled over an array of 5σx5σ points. Lastly, for each output of the two Haar operators, we construct a 4 vector summing over the 5x5 array of points in the square. We can lastly, string together these 4-vectors for each of the 16 squares to get a 64-element descriptor vector associated with each point.

5 Results

5.1 Discussion of results

5.2 Source Images

Figure 2: Source images used for corner detection.

5.3 Harris Results

5.3.1 Harris+NCC point matches

Sigma = 0.6

Figure 3: Matching points as detected using a Harris corner detection algorithm using an NCC feature similarity measure with sigma=0.6.

Sigma = 0.9

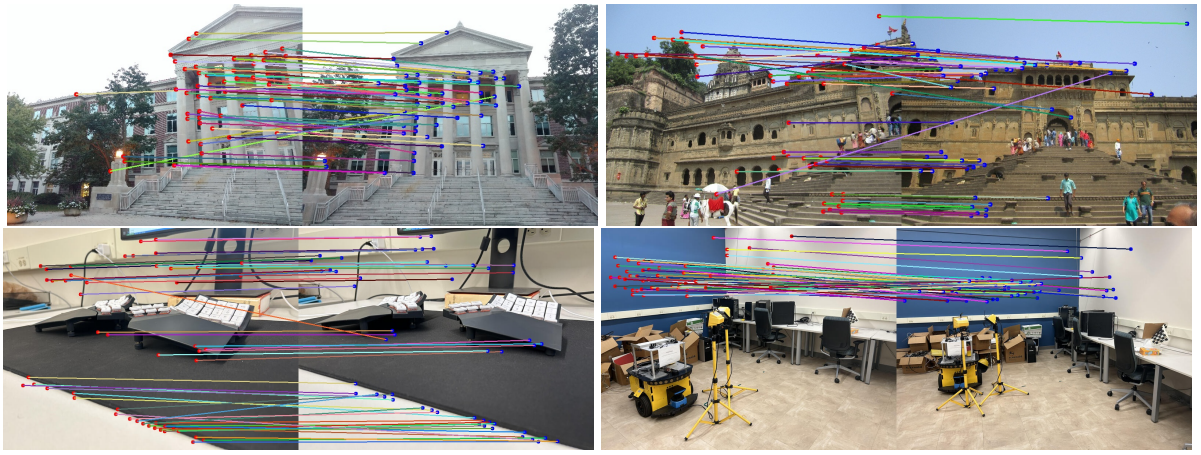

Figure 4: Matching points as detected using a Harris corner detection algorithm using an NCC feature similarity measure with sigma=0.9.

Sigma = 1.2

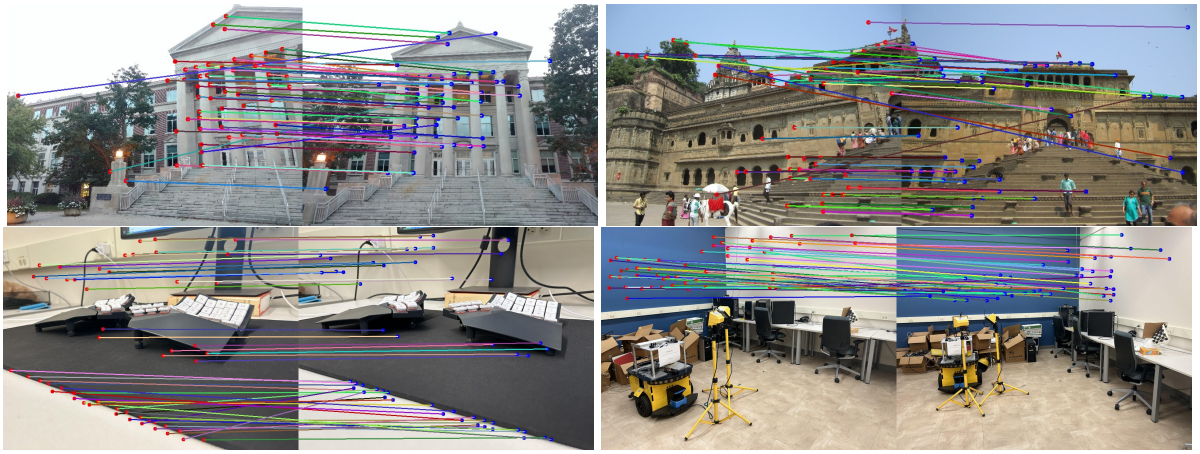

Figure 5: Matching points as detected using a Harris corner detection algorithm using an NCC feature similarity measure with sigma=1.2.

Sigma = 5.0

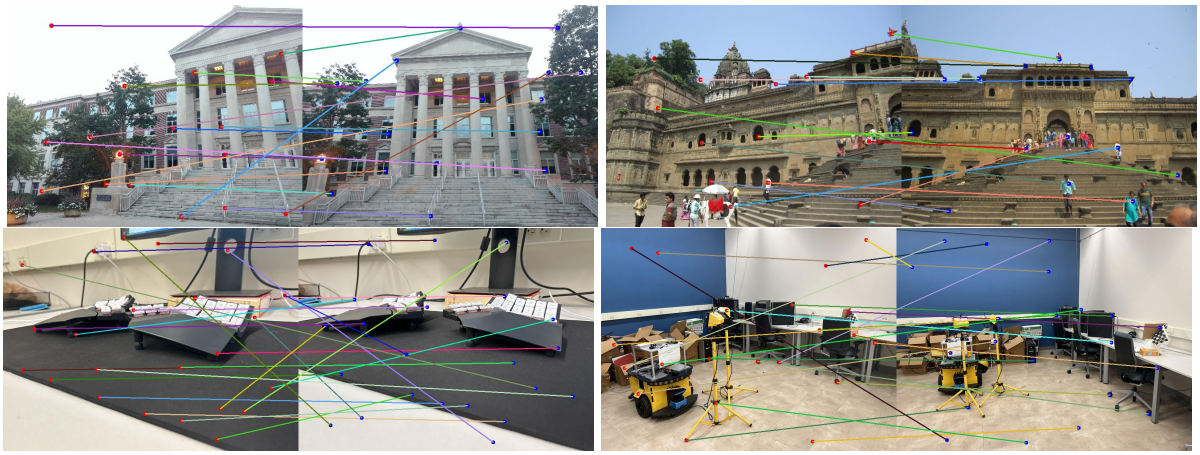

Figure 6: Matching points as detected using a Harris corner detection algorithm using an NCCSSDfeature similarity measure with sigma=5.0.

5.3.2 Harris+SSD point matches

Sigma = 0.6

Figure 7: Matching points as detected using a Harris corner detection algorithm using an SSD feature similarity measure with sigma=0.6.

Sigma = 0.9

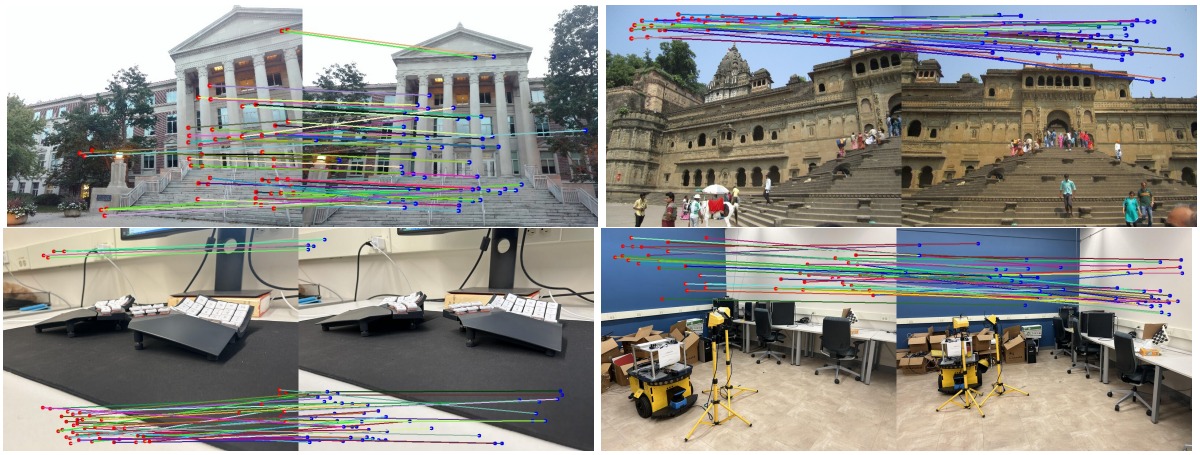

Figure 8: Matching points as detected using a Harris corner detection algorithm using an SSD feature similarity measure with sigma=0.9.

Sigma = 1.2

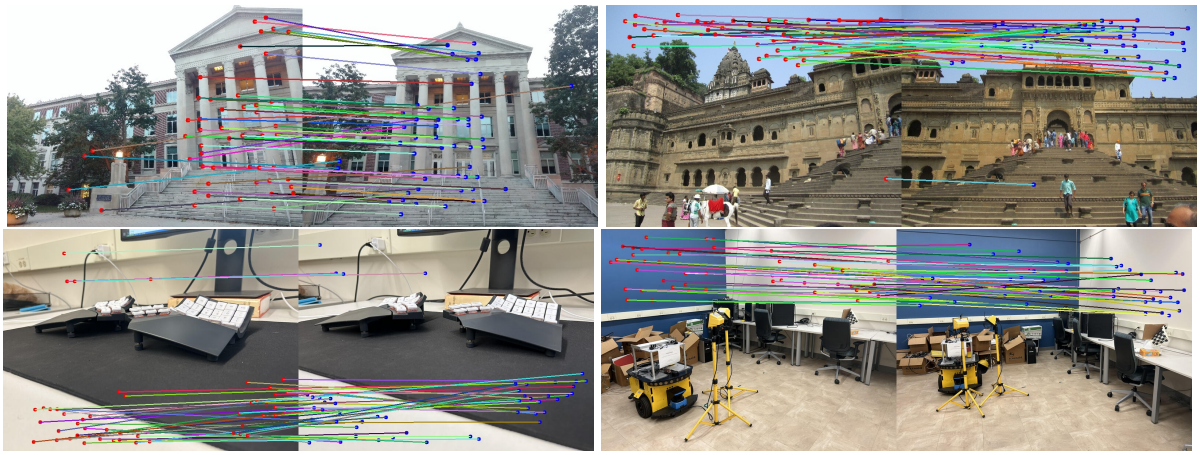

Figure 9: Matching points as detected using a Harris corner detection algorithm using an SSD feature similarity measure with sigma=1.2.

Sigma = 5.0

Figure 10: Matching points as detected using a Harris corner detection algorithm using an SSD feature similarity measure with sigma=5.0.

5.3.3 Harris+NCC Detected Points

Sigma = 0.6

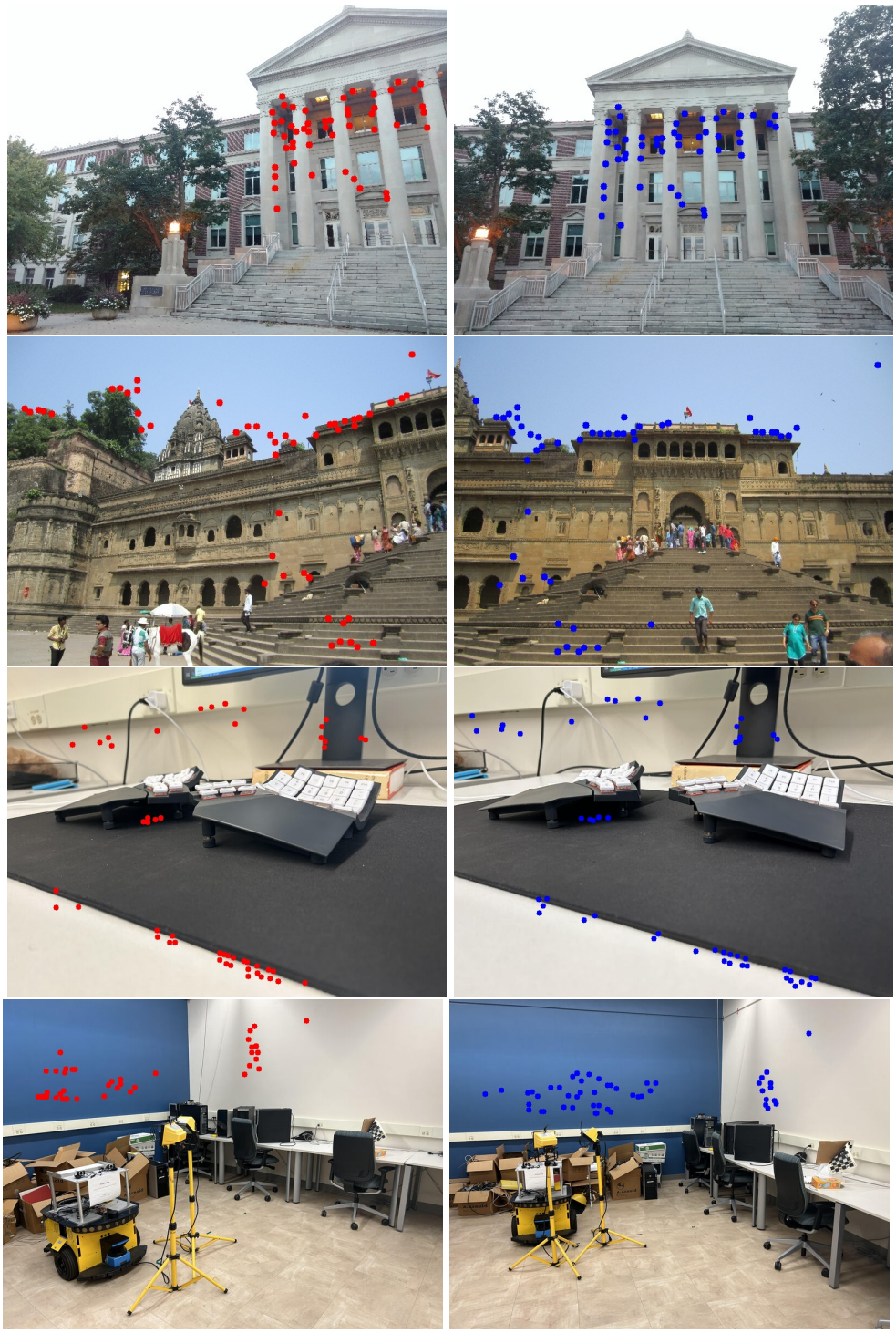

Figure 11: Corner points as detected using a Harris corner detection algorithm using an NCC feature similarity measure with sigma=0.6.

Sigma = 0.9

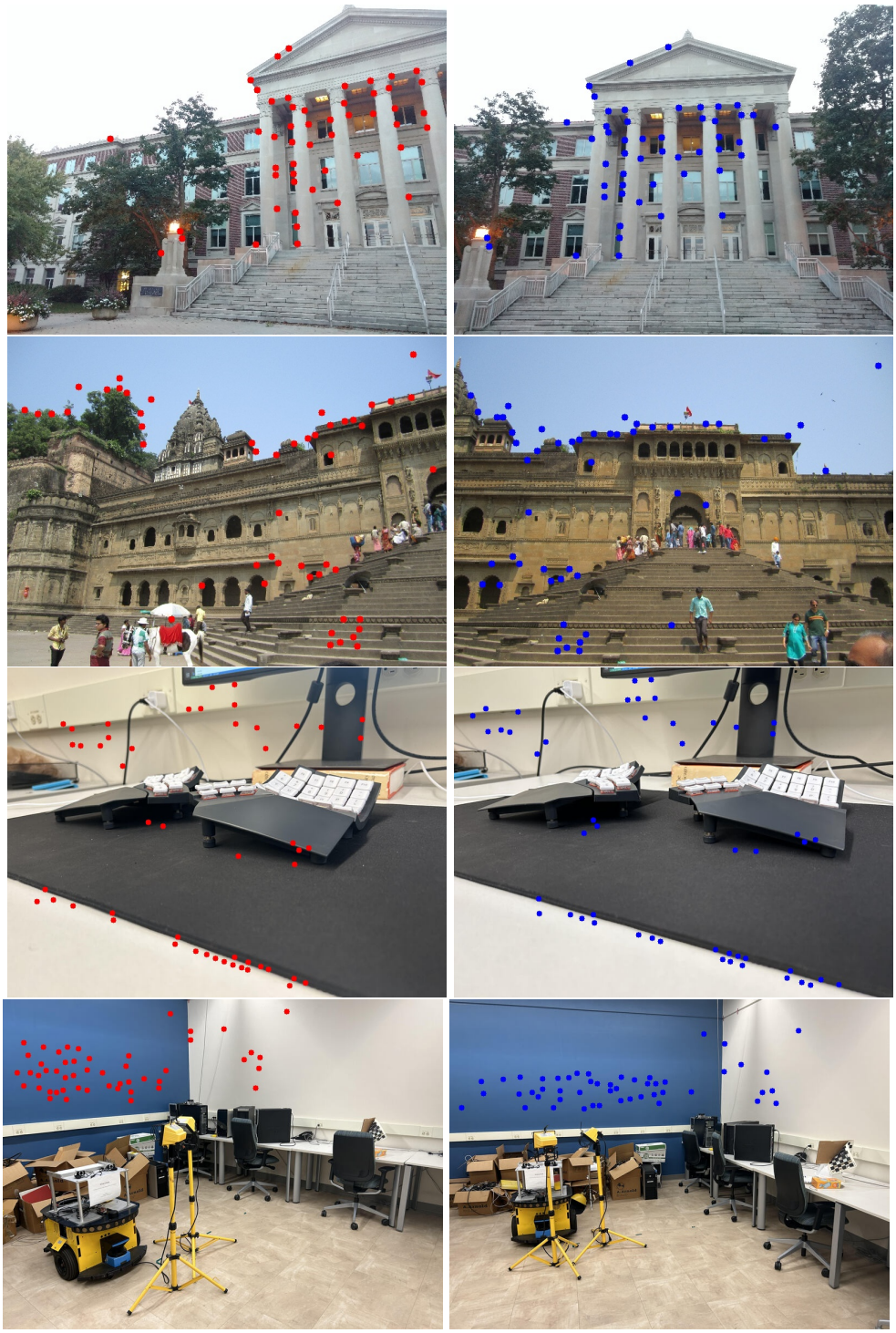

Figure 12: Corner points as detected using a Harris corner detection algorithm using an NCC feature similarity measure with sigma=0.9.

Sigma = 1.2

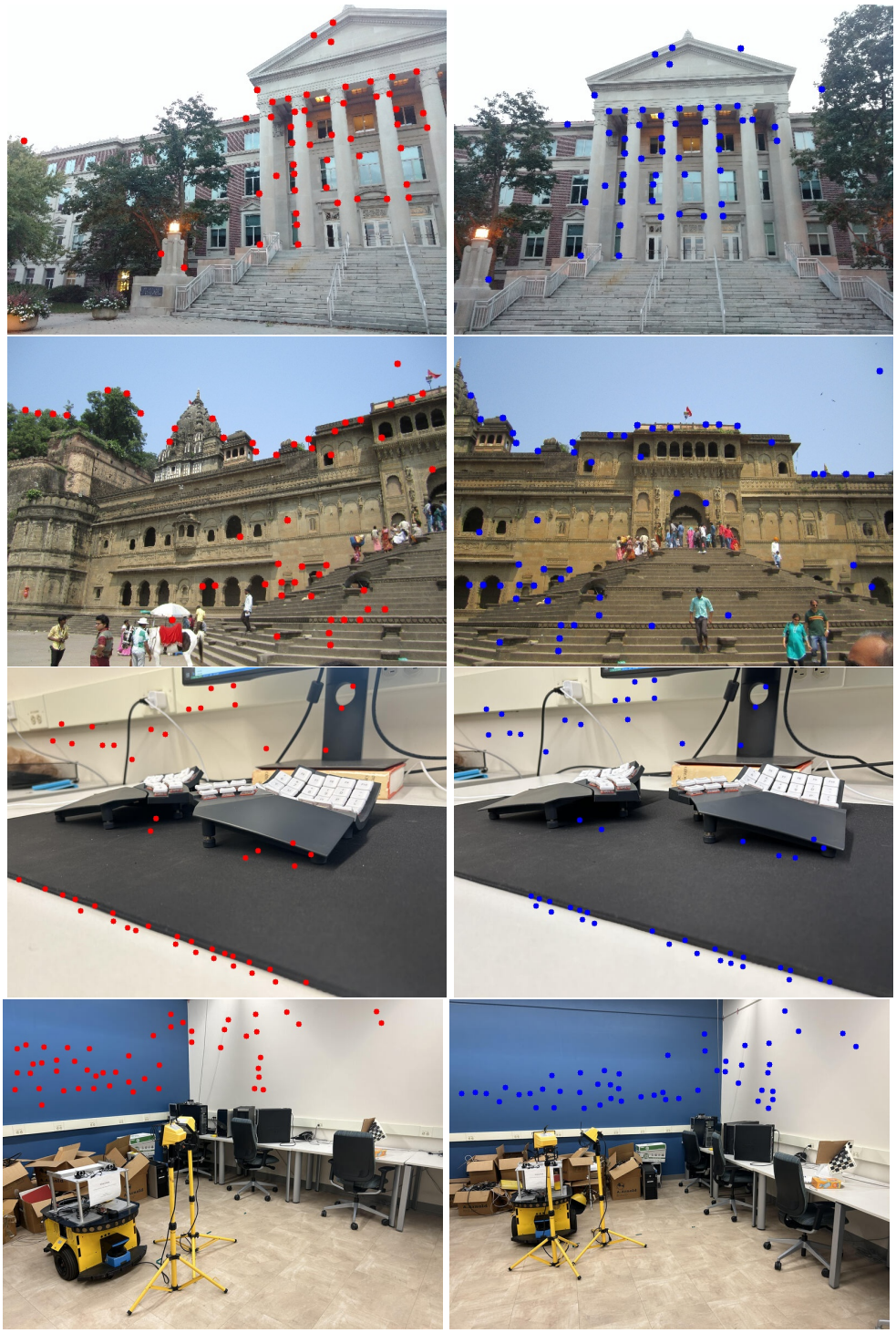

Figure 13: Corner points as detected using a Harris corner detection algorithm using an NCC feature similarity measure with sigma=0.9.

Sigma = 5.0

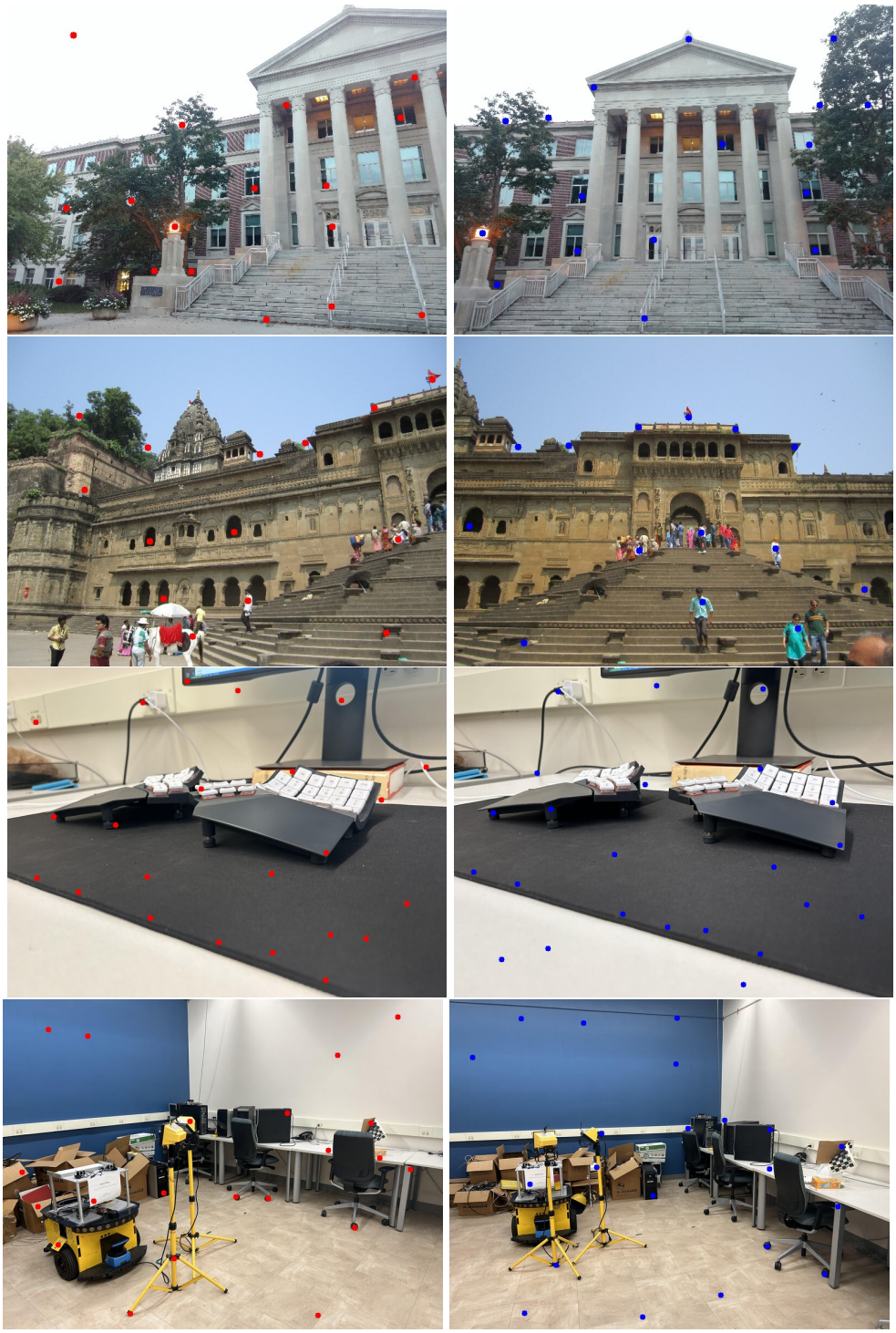

Figure 14: Corner points as detected using a Harris corner detection algorithm using an NCC feature similarity measure with sigma=5.0.

5.3.4 Harris+SSD Detected Points

Sigma = 0.6

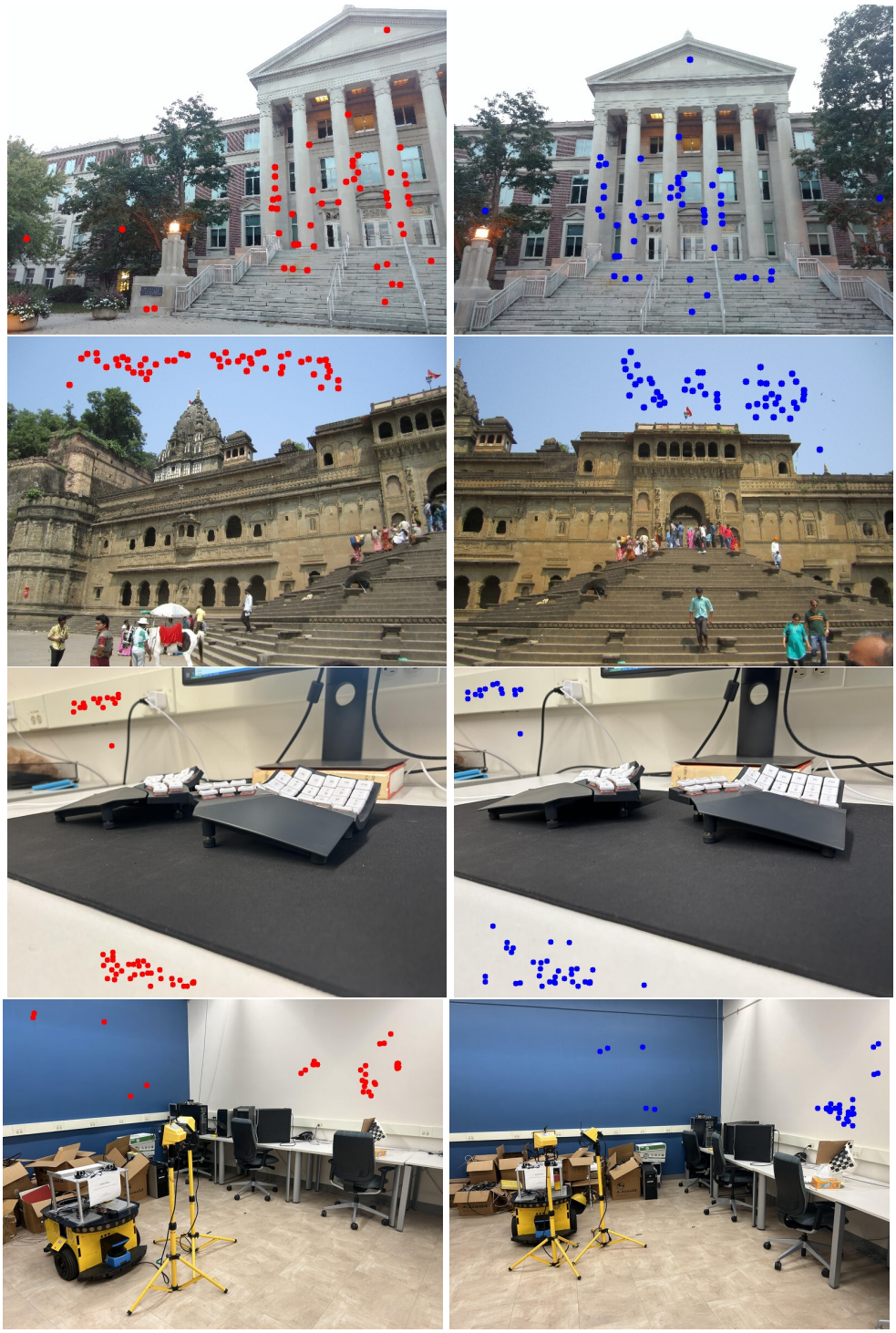

Figure 15: Corner points as detected using a Harris corner detection algorithm using an SSD feature similarity measure with sigma=0.6.

Sigma = 0.9

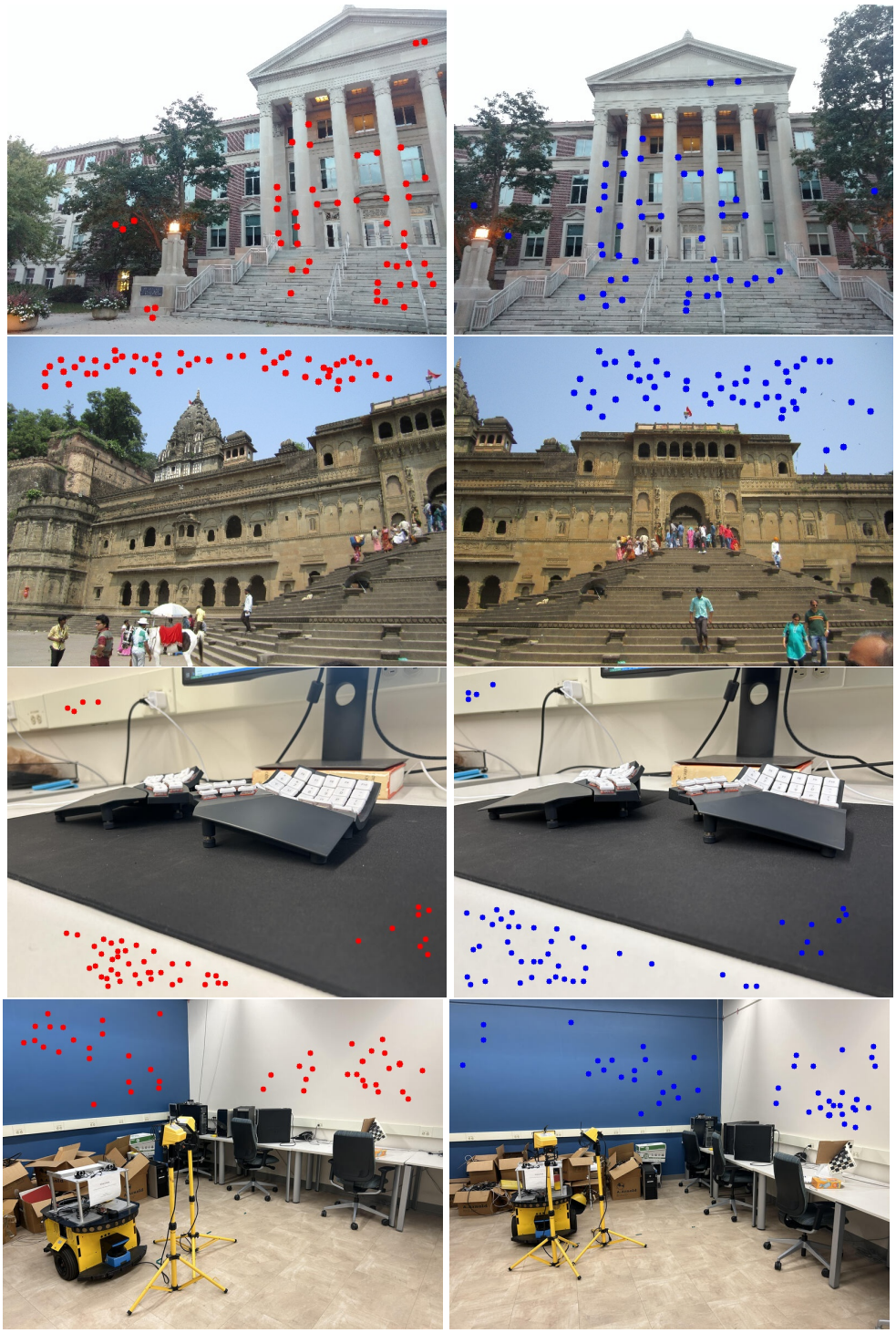

Figure 16: Corner points as detected using a Harris corner detection algorithm using an SSD feature similarity measure with sigma=0.9.

Sigma = 1.2

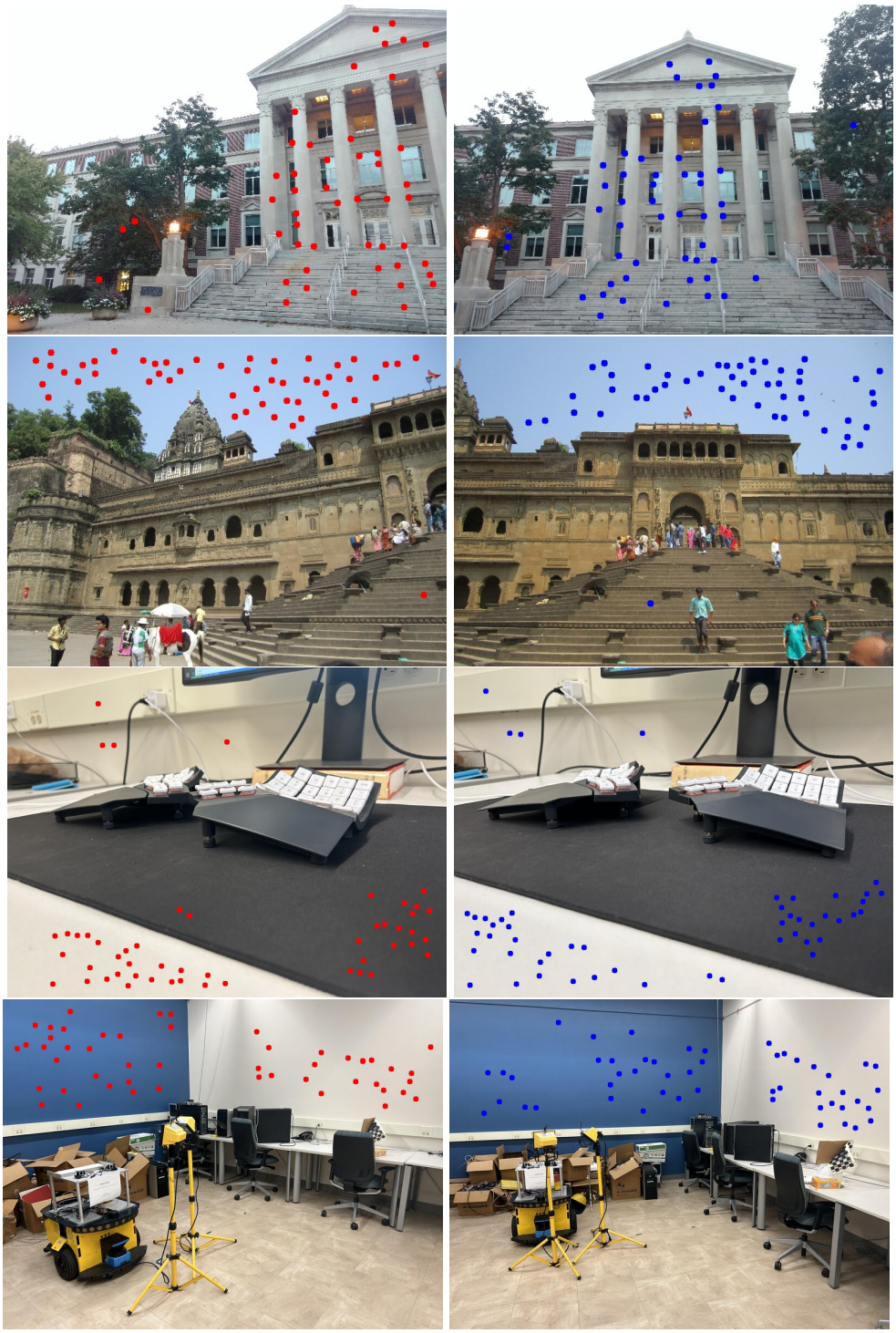

Figure 17: Corner points as detected using a Harris corner detection algorithm using an SSD feature similarity measure with sigma=0.9.

Sigma = 5.0

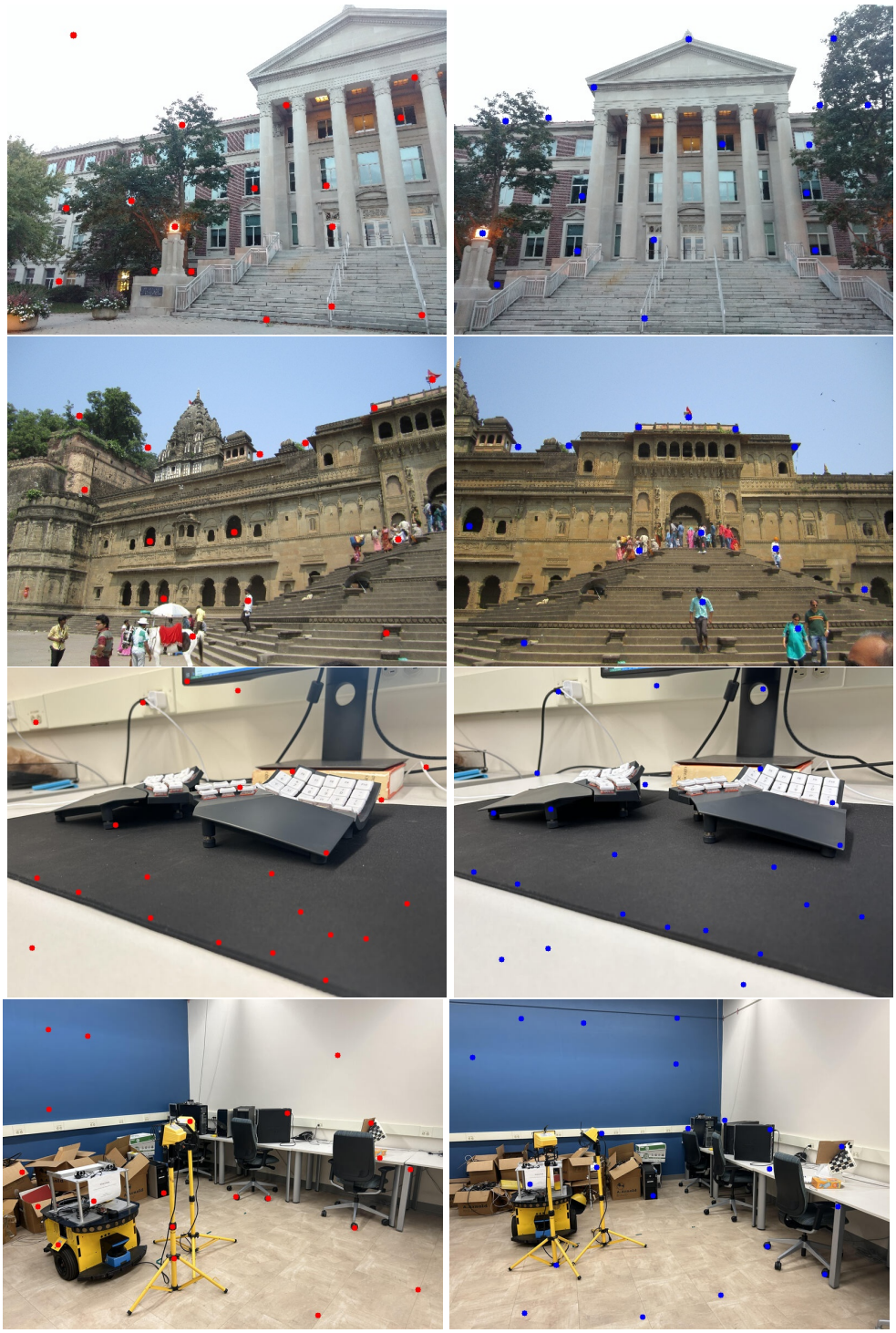

Figure 18: Corner points as detected using a Harris corner detection algorithm using an SSD feature similarity measure with sigma=5.0.

5.4 SIFT Results

Included below are the resulting key-point matches detected through the SIFT software.

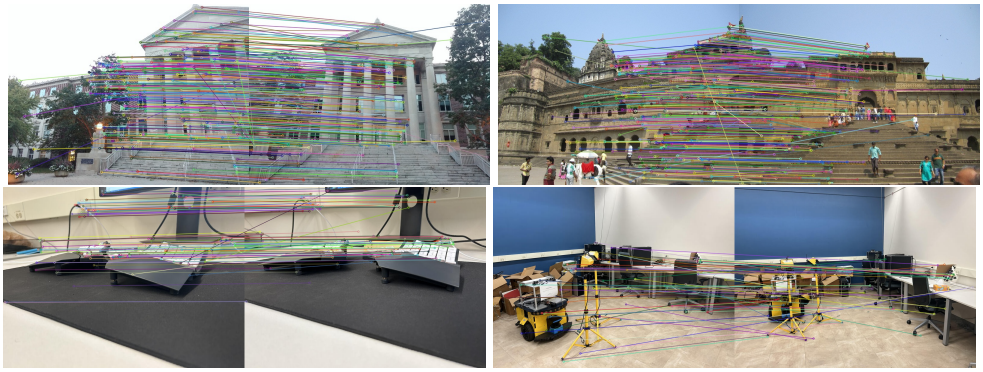

Figure 19: Resulting keypoint matches using the SIFT software for all image pairs used in this report.

5.5 Superglue Results

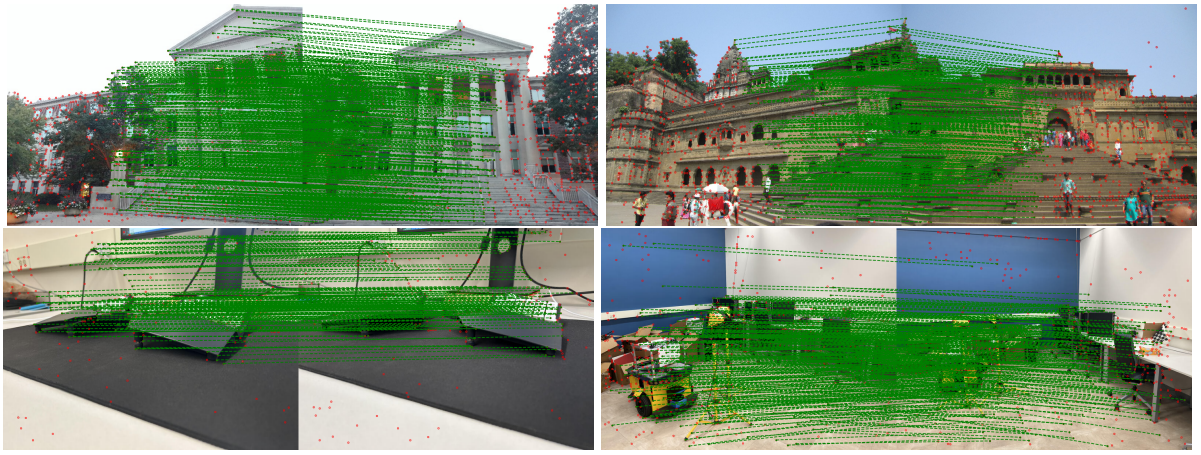

Figure 20: Point matches using the Superglue Neural Network based software.

6 Discussion of results

In conclusion the performance of the key point matching software can be organized as follows:

1. SuperGlue
2. SIFT
3. Harris with NCC for feature similarity
4. Harris with SSD for feature similarity

The SuperPoint portion of SuperGlue detected the highest amount of interest points which means that there was a decent amount of noise; however, it captured what seemed to be all key interest points that were captured by other algorithms as different scales. Also, there was very little noise in its line outputs which means that it can very accurately detect interest point matches. This was not possible for the Harris key-point matching algorithm. While the interest points that are outputted by the algorithm were fairly decent, there was a lot of noise in the output. Using the NCC feature similarity metric seemed to deal with most of these noisy outputs and found mostly matching points with only a few errors; however, the SSD algorithm was not able to find what I assume are the most important interest point matches. Instead it usually found matches within the white-space in the sky, or the light boundaries on the blue and white walls within my fourth image. While it was successful in finding matches between these points, they were much more noisy than those calculated using NCC. Lastly, by using a larger σ value my Harris implementation was only able to detect larger-scale features due to the heavy-smoothing which meant

that smaller keypoints were missed. This produced a less noisy output; however, only the small σ values detected all the interest points.

All in all, outside of the SIFT and SuperGlue based matching the parameters that led to the best performance were running Harris with NCC at a σ value of 1.2.

References

[1] Christopher G. Harris and M. J. Stephens. "A Combined Corner and Edge Detector". In: *Alvey Vision Conference*. 1988. URL: https://api.semanticscholar.org/CorpusID:1694378.