Homework 4

Fangda Li
ECE 661 - Computer Vision

October 15, 2016

1 Methodology

1.1 Harris Corner Detector

Corners are certain regions in the image that have significant variation of intensity along at least two direction. They are often used as interest points for establishing point-to-point correspondence between images of the same object in different views. Harris corner detectors use the following steps to identify corners:

1. Find the color intensity gradients along x and y direction for the whole image: d_x and d_y, respectively. We could use Sobel operator to find the gradients if scales were not important. Thus, as the problem asks us to apply the corner detector on different scales, we choose Haar wavelet filter instead. Haar filter is derived from the basic form $(-1, 1)$ for x direction and $(-1, 1)^T$ for y direction. Then, given σ it can be scaled up to $S \times S$, where S is the smallest even number that is greater than $4 \times \sigma$. An example of Haar filter with $\sigma = 1.2$ $(S = 6)$ for d_x and d_y respectively is shown below:

$$Haar filter for\ d_x = \begin{bmatrix} -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \end{bmatrix} \tag{1}$$

$$Haar filter for\ d_y = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}. \tag{2}$$

   We convolve the two filters above with the input image to obtain two gradient images along x and y directions.

2. After we have obtained d_x and d_y, the next step is to construct the co-variance matrix M of the sum of the gradients for all pixels within an image patch. In our case, we chose the size of a patch to be $5\sigma \times 5\sigma$. In order to represent a corner, M of the image patch has to be full rank (i.e. has a rank of 2). On the other hand, if the patch represents an edge, M will have a rank of 1. The co-variance matrix M is constructed as shown:

$$M = \begin{bmatrix} \sum d_x^2 & \sum d_x d_y \\ \sum d_x d_y & \sum d_y^2 \end{bmatrix}. \tag{3}$$

Next, M can be further represented by its eigen decomposition,

$$M = A^{-1} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} A. \tag{4}$$

The eigen values of M, λ_1 and λ_2, are used in the next step to compute the corner response R.

3. Compute the corner response of an image patch. First, the corner response R is defined as,

$$R = det(M) - k(tr(M))^2, \tag{5}$$

where $det(M)$ is the determinant of co-variance matrix M, $tr(M)$ is the trace of M and k is an empirically chosen constant. More specifically, k is set to be 0.04. Moreover, $det(M)$ and $tr(M)$ can be written as,

$$det(M) = \lambda_1 \lambda_2, \tag{6}$$

$$tr(M) = \lambda_1 + \lambda_2. \tag{7}$$

4. Lastly, threshold the corner responses with T and perform non-maximum suppression to obtain final corner pixels. The rational is simple: to eliminate pixels whose corner response is not strong enough by applying a threshold. Then, apply non-maximum suppression to make sure a single corner is not represented repetitively.

1.2 Scale Invariant Feature Transform (SIFT)

In this subsection, we explain how SIFT features are extracted from an image. More specifically, the procedure include the five following steps:

1. **Find all the local extrema in the DoG pyramid.** In order to extract features in the most representative regions of an image, we find points that are either the local maximum or local minimum in the (x, y, σ) 3D space in each octave in the DoG pyramid. For each point, we do 26 comparisons to find the extremum: 9 at the level above, 8 at the current level and 9 at the level below.

2. **Localization of the extremum in the DoG pyramid.** As the σ increases in the DoG pyramid, the size of the level decreases. As a result, the location of a local extremum on a higher level might not directly represent a SIFT point on the original image. To resolve this discrepancy, we use the Taylor series expansion of $D(x, y, \sigma)$

to find its location on the original image with "subpixel" accuracy, as shown in the following equation:

$$D(\overrightarrow{x}) = D(\overrightarrow{x_0}) + J^T(\overrightarrow{x_0})\overrightarrow{x} + \frac{1}{2}\overrightarrow{x}^T H(\overrightarrow{x_0})\overrightarrow{x}, \tag{8}$$

where $\overrightarrow{x_0}$ is the extremum found on a higher level, $\overrightarrow{x}$ is the true location of the extremum, $J(\overrightarrow{x_0})$ is the Jacobian of x_0 and $H(\overrightarrow{x_0})$ is the Hessian of x_0. Next, taking the derivative with respect to $\overrightarrow{x}$ of Equation 8 gives us

$$0 \approx J(\overrightarrow{x_0}) + H(\overrightarrow{x_0})\overrightarrow{x}, \tag{9}$$

which can be further written as

$$\overrightarrow{x} = -H(\overrightarrow{x_0})J(\overrightarrow{x_0}). \tag{10}$$

Now we have obtained the true location of the extremum.

3. **Weak extrema rejection by thresholding.** Here we apply a threshold of 0.03 for $|D(\overrightarrow{x}|$ to eliminate weak extrema. We also reject the extrema which draw their support from edges in the image.

4. **Associate each extremum with with "dominant local orientation".** To find the "dominant local orientation" for an extremum, we first calculate the gradient magnitude $m(x, y)$ and gradient orientation $\theta(x, y)$ of every pixel within a $K \times K$ window around the extremum:

$$m(x,y) = \sqrt{|ff(x+1,y,\sigma) - ff(x,y,\sigma)|^2 + |ff(x,y+1,\sigma) - ff(x,y,\sigma)|^2}, \tag{11}$$

$$\theta(x,y) = arctan\frac{ff(x,y+1,\sigma) - ff(x,y,\sigma)}{ff(x+1,y,\sigma) - ff(x,y,\sigma)}. \tag{12}$$

Next, we construct a histogram with 36 bins of the $m(x, y)$ weighted orientations $\theta(x, y)$ and pick the bin of the peak in the histogram as our "dominant local orientation".

5. **Create the SIFT descriptor for each extremum.** Finally, to construct the SIFT descriptor, a 128-dimensional vector, we divide the 16×16 neighboring window of the extremum into 16 4×4 cells. Then, for each cell, we construct a magnitude-weighted 8 bin histogram of the pixel orientations with respect to the "dominant local direction" found in the previous step. Note that the magnitudes are also weighted by a Gaussian distribution with its σ being half the width of the neighborhood. Now since we have obtained 16 8-bin histograms, we string all of them together and apply normalization to get our final 128-dimensional SIFT descriptor.

1.3 Correspondence Metrics

Now we have detected corners as interest points (features) in the pair of images, we want to establish correspondences between matching points from the two images. Two correspondence metrics are used: SSD and NCC.

1.3.1 Sum of Squared Differences (SSD)

For each interest point (i_1, j_1) from one image, we calculate the SSD of pixel gray levels within a $N \times N$ neighboring window between (i_1, j_1) and every other interest point from the other image, (i_2, j_2). Intuitively, a perfect matching will result in SSD being 0 and we will pick the pair with the minimum SSD to be our matching pair.

$$SSD = \sum_{i=-S/2}^{S/2} \sum_{j=-S/2}^{S/2} |I_1(i,j) - I_2(i,j)|^2 \tag{13}$$

1.3.2 Normalized Cross Correlation (NCC)

Similar to SSD, NCC also compares the gray levels of pixels within a $N \times N$ neighboring window between (i_1, j_1) and every other interest point from the other image, (i_2, j_2). The equation for calculating NCC is shown below:

$$NCC = \frac{\sum_{i=-S/2}^{S/2} \sum_{j=-S/2}^{S/2} (I_1(i,j) - m_1)(I_2(i,j) - m_2)}{\sqrt{\sum_{i=-S/2}^{S/2} \sum_{j=-S/2}^{S/2} (I_1(i,j) - m_1)^2 \sum_{i=-S/2}^{S/2} \sum_{j=-S/2}^{S/2} (I_2(i,j) - m_2)^2}}, \tag{14}$$

where m_1 and m_2 denote the mean gray level in the two neighboring windows.

2 Results and Discussion

2.1 Relevant Parameters

When applying the Harris corner detector with SSD and NCC, the relevant parameters are shown in the following table. Particularly, in this experiment, four different scales,

Table 1: Relevant parameters for Harris, SSD and NCC

Harris Corner Detector	Scale	σ
	Corner response threshold	T_R
SSD	Matching window size	N
	SSD score threshold	T_{SSD}
NCC	Matching window size	N
	NCC score threshold	T_{NCC}

$\sigma = 1.2, 2.4, 3.6, 4.8$ are applied on each image pairs. The corner response threshold for rejecting weak corners is chosen empirically based on each individual pair, as it will be shown later. Furthermore, to make the matching robust to large amount of points to match, the window size N for the matching metrics are chosen to be relatively large, 21 in particular. To reject false matching in SSD, we throw away all the candidates that have higher SSD scores than $T_{SSD} = 10 \times SSD_{min}$, where SSD_{min} is the minimum of all matching scores. Similarly in NCC, we reject the candidates that have lower NCC scores than $T_{NCC} = 0.9 \times NCC_{max}$, where NCC_{max} is the maximum of all matching scores.

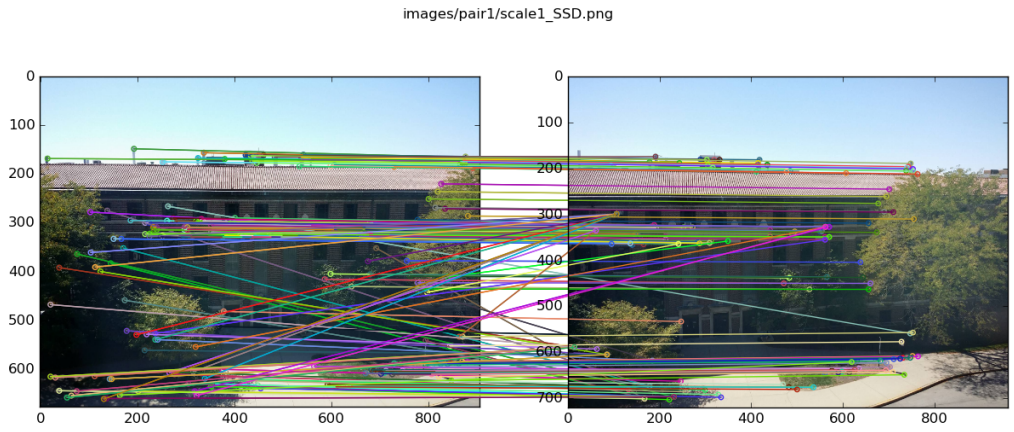
2.2 Image Pair 1

For this pair, the corner response threshold for the four different scales are chosen to be $T_R = 20, 40, 80, 160$. T_R is chosen in a quadratic manner to account for the quadratic increment of the window size M.

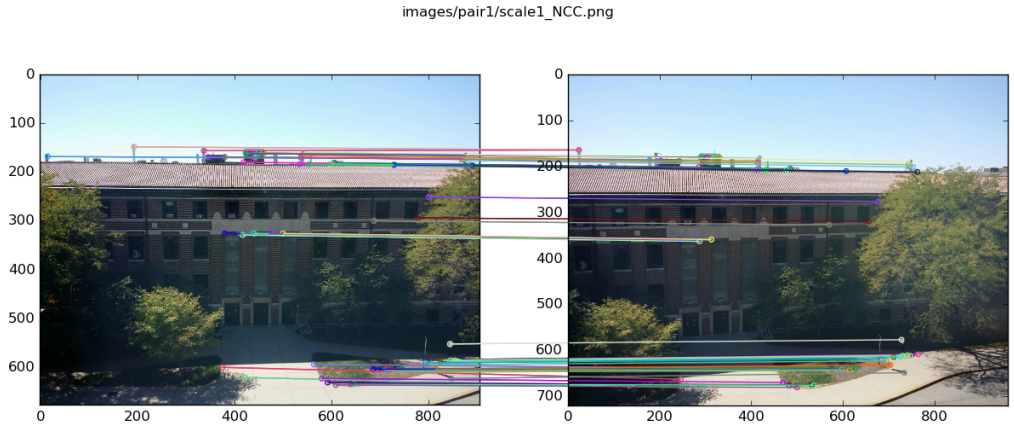
(a) Input image 1

(b) Input image 2

Figure 1: Input images

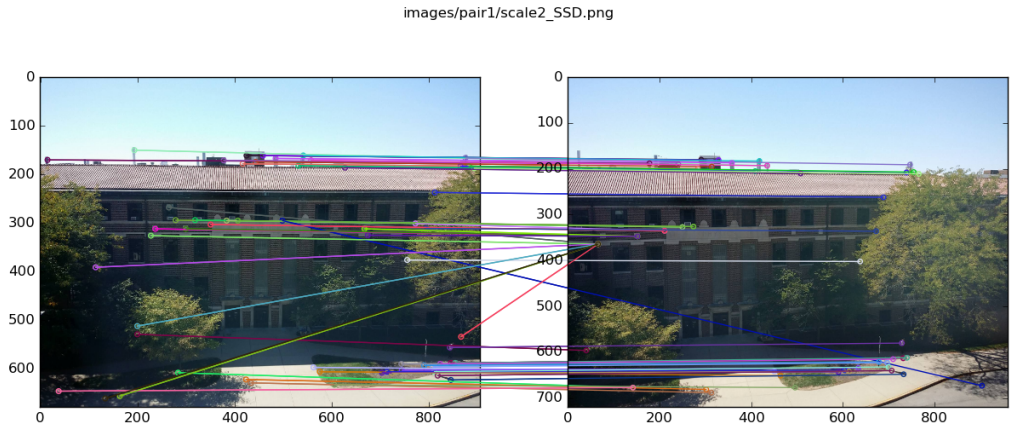


(a) SSD at scale 1

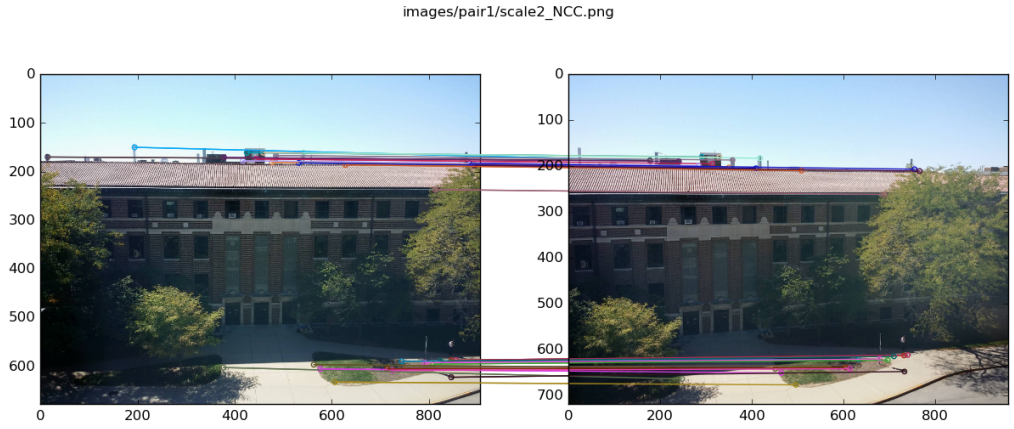


(b) NCC at scale 1

Figure 2: Interest point matching with Harris corner detector at 1

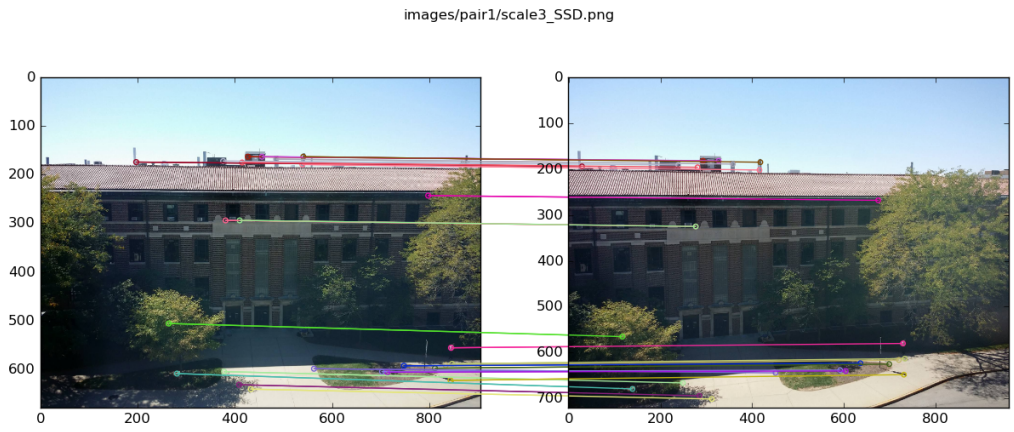


(a) SSD at scale 2

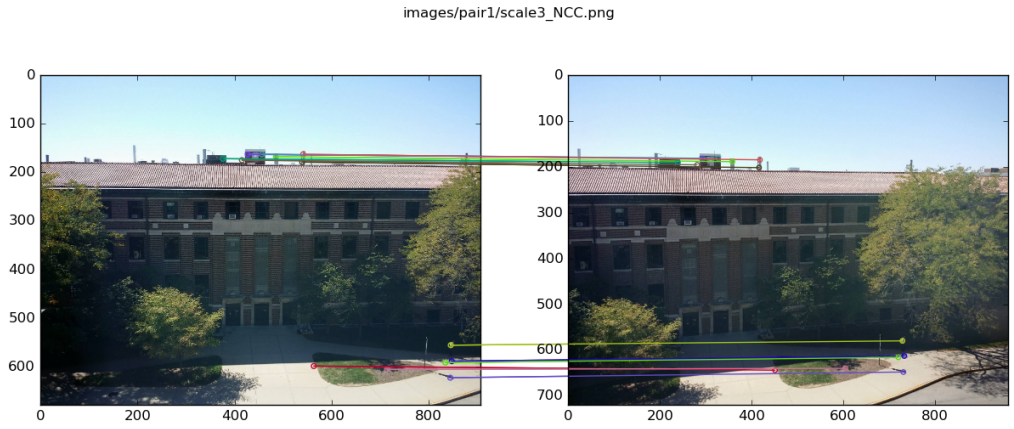


(b) NCC at scale 2

Figure 3: Interest point matching with Harris corner detector at scale 2

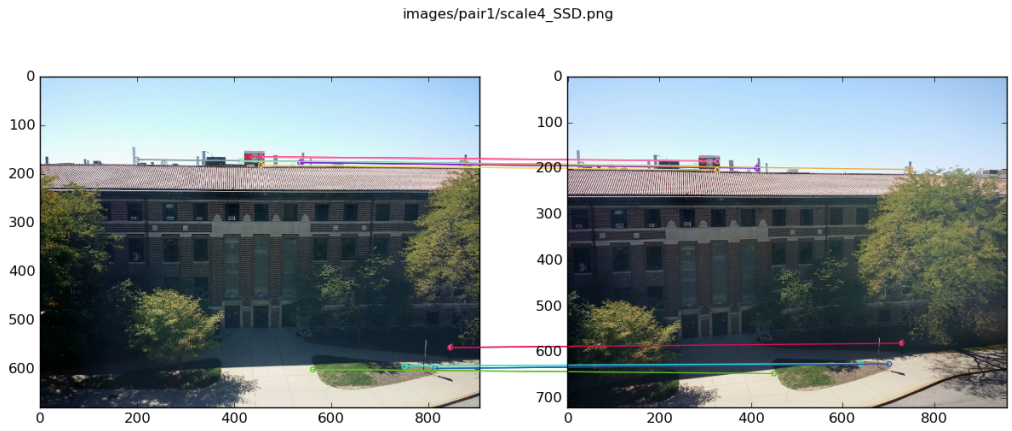


(a) SSD at scale 3

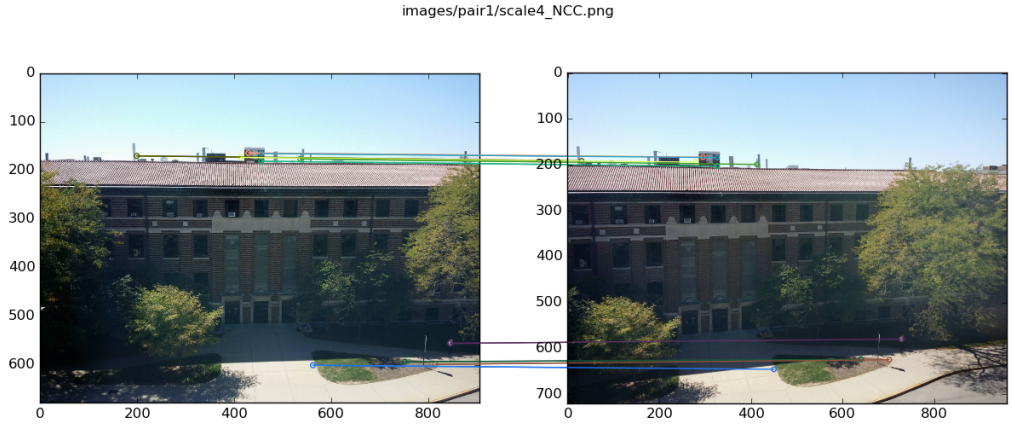


(b) NCC at scale 3

Figure 4: Interest point matching with Harris corner detector at scale 3



(a) SSD at scale 4



(b) NCC at scale 4

Figure 5: Interest point matching with Harris corner detector at scale 4

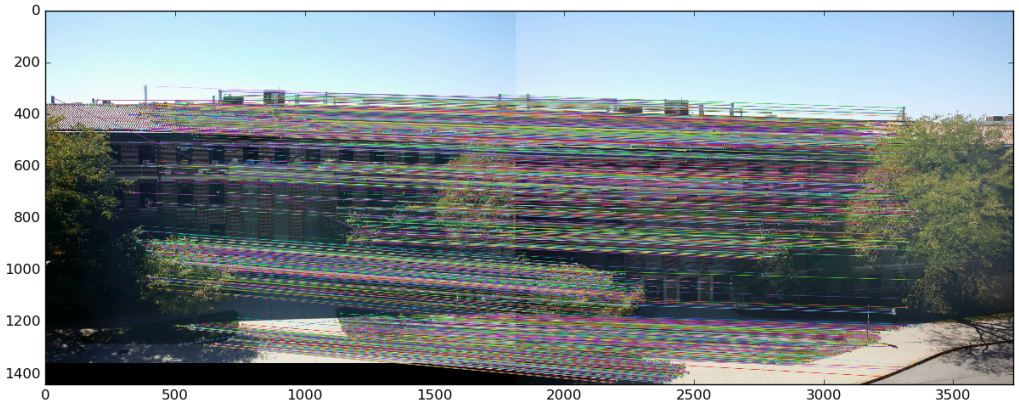
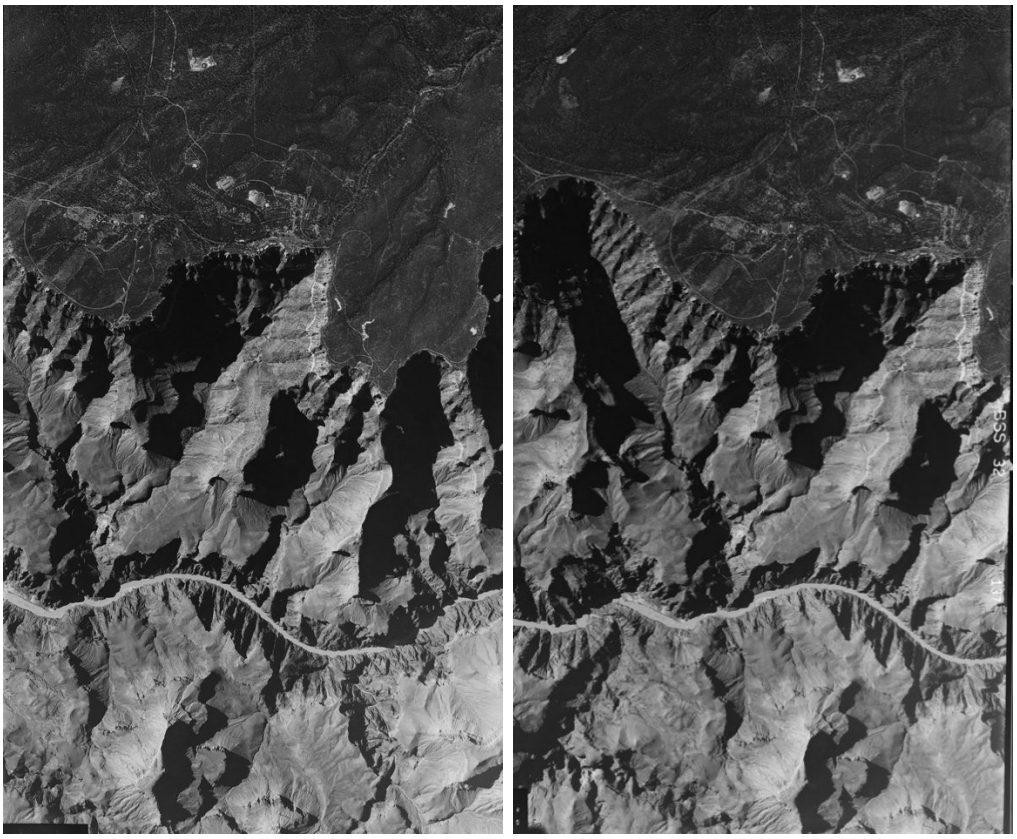


Figure 6: Interest point matching with SIFT

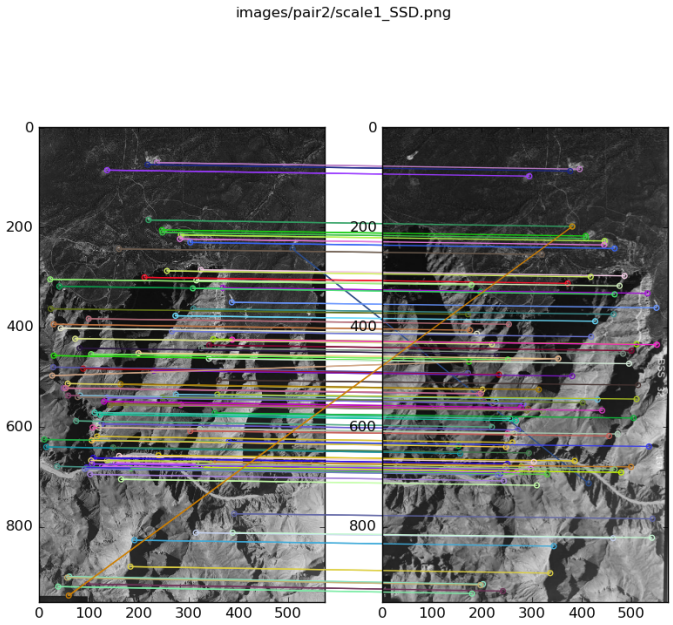
2.3 Image Pair 2

For this pair, the corner response threshold for the four different scales are chosen to be $T_R = 100, 400, 800, 1600$.

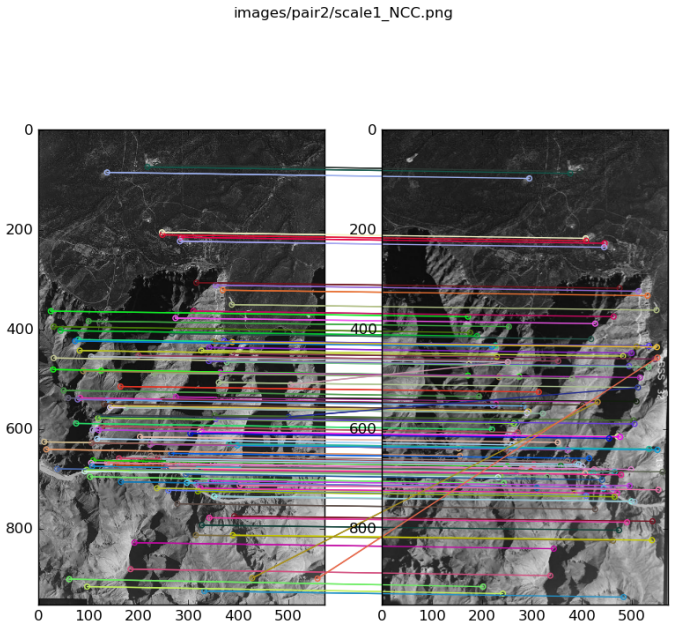


(a) Input image 1 (b) Input image 2

Figure 7: Input images

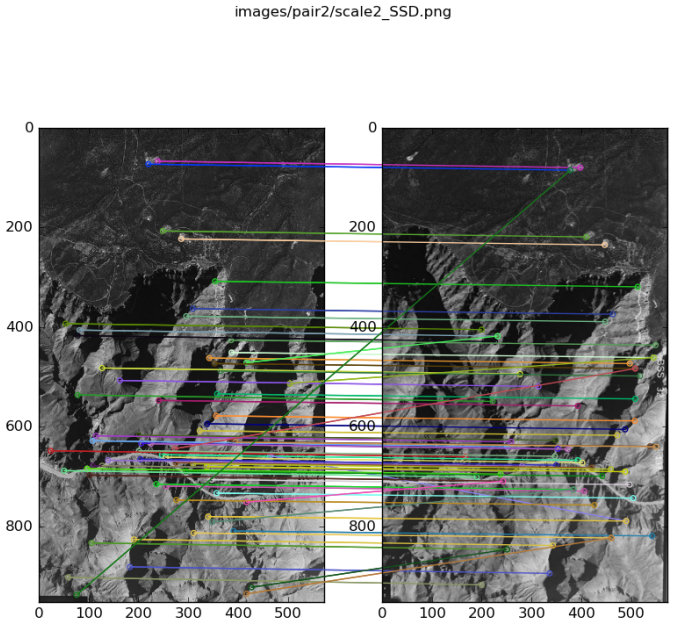


(a) SSD at scale 1

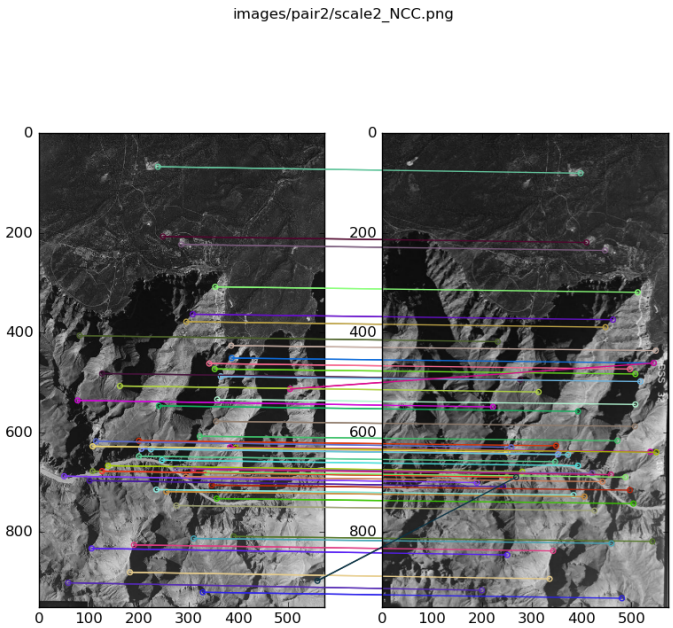


(b) NCC at scale 1

Figure 8: Interest point matching with Harris corner detector at 1

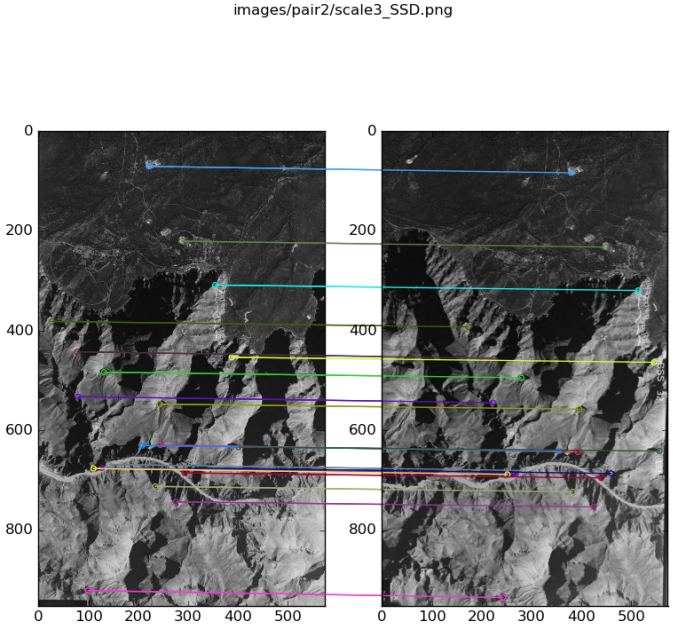


(a) SSD at scale 2

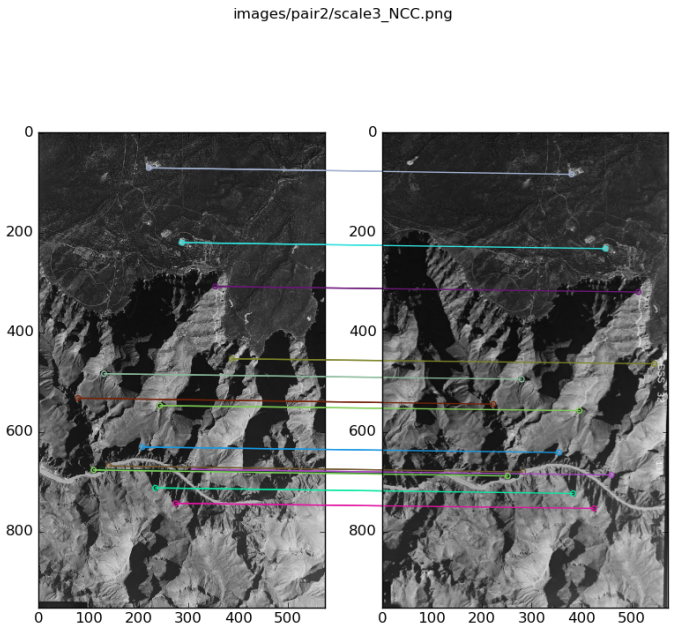


(b) NCC at scale 2

Figure 9: Interest point matching with Harris corner detector at scale 2

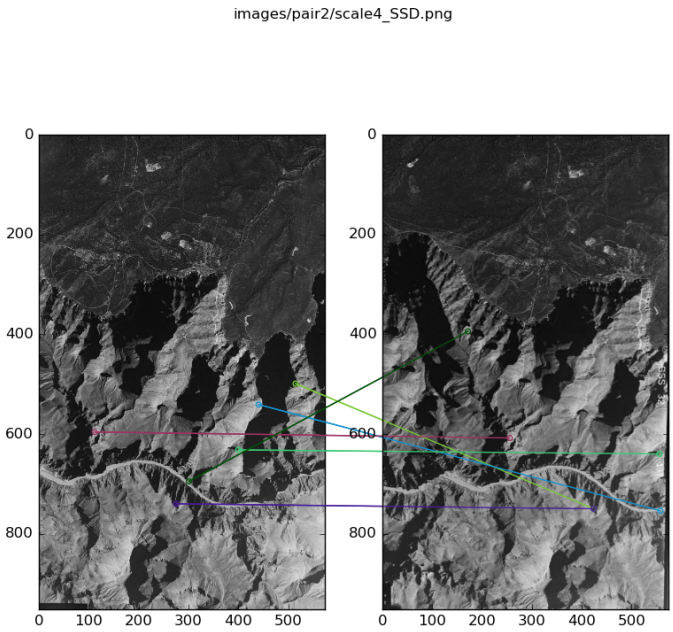


(a) SSD at scale 3

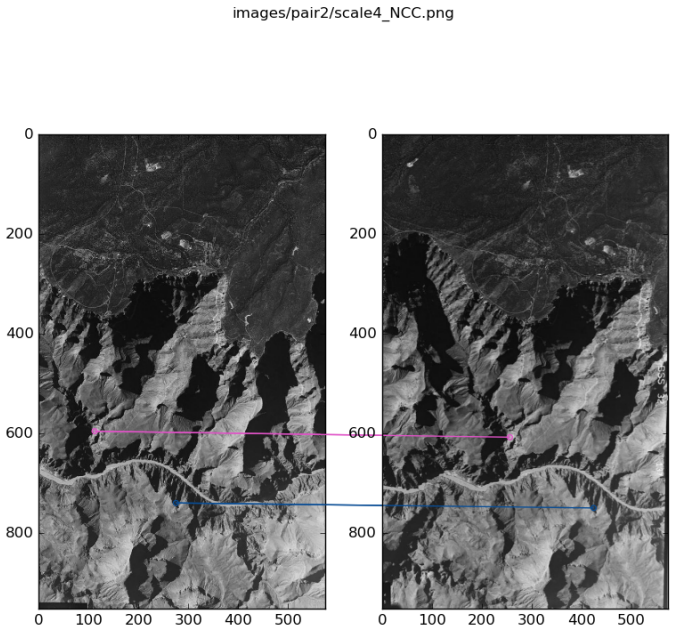


(b) NCC at scale 3

Figure 10: Interest point matching with Harris corner detector at scale 3



(a) SSD at scale 4



(b) NCC at scale 4

Figure 11: Interest point matching with Harris corner detector at scale 4

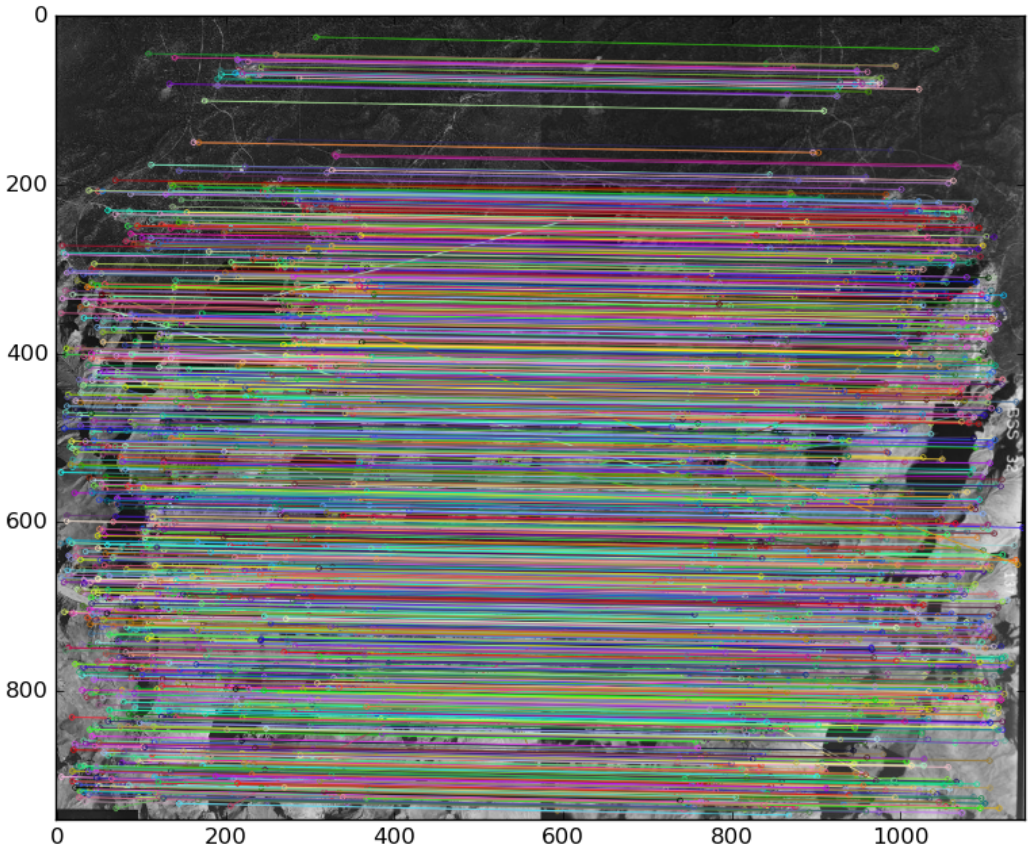


Figure 12: Interest point matching with SIFT

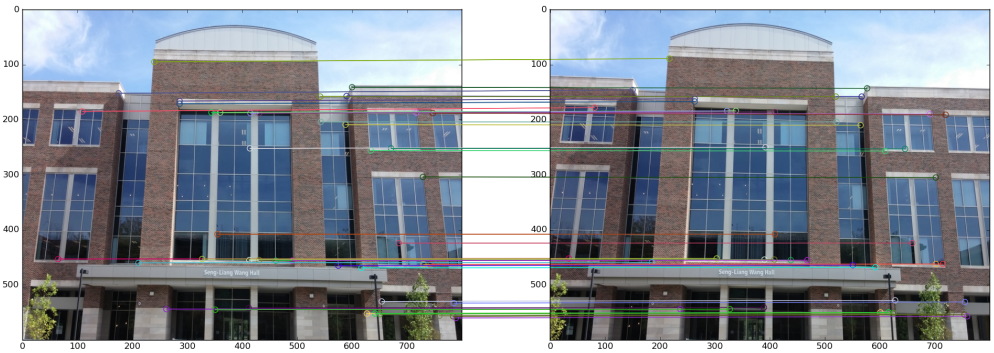
2.4 Image Pair 3

For this pair, the corner response threshold for the four different scales are chosen to be $T_R = 100, 400, 800, 1600$.

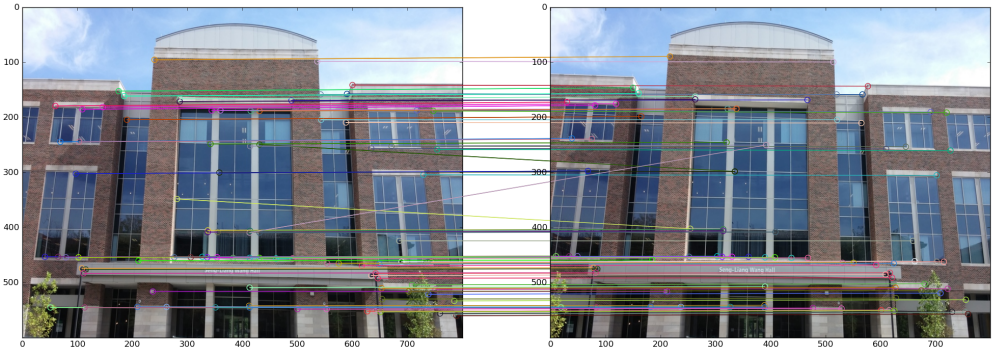
(a) Input image 1

(b) Input image 2

Figure 13: Input images

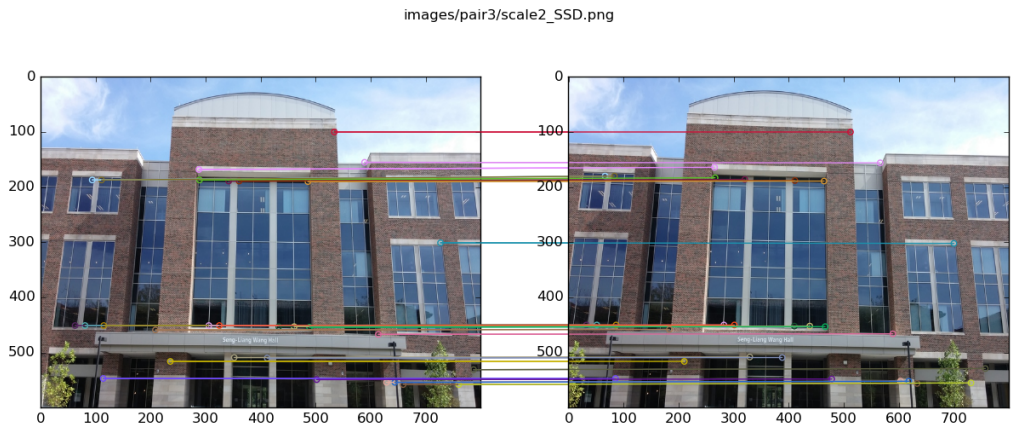


(a) SSD at scale 1

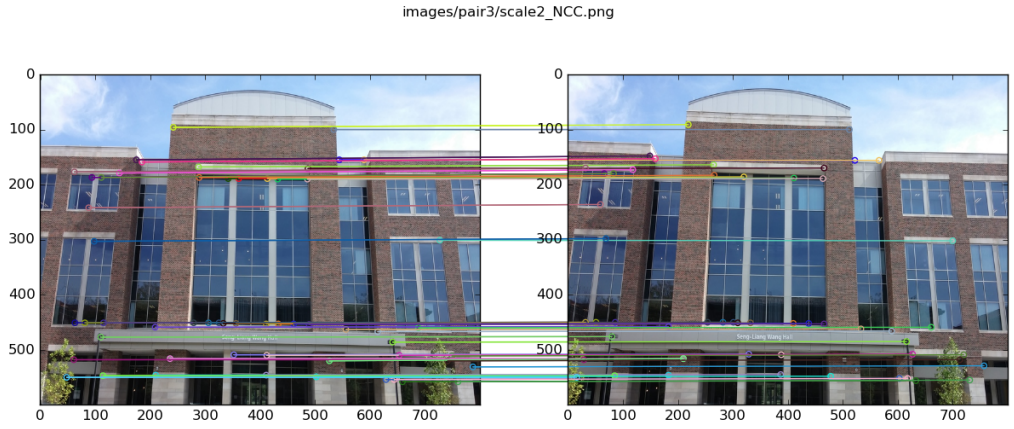


(b) NCC at scale 1

Figure 14: Interest point matching with Harris corner detector at 1

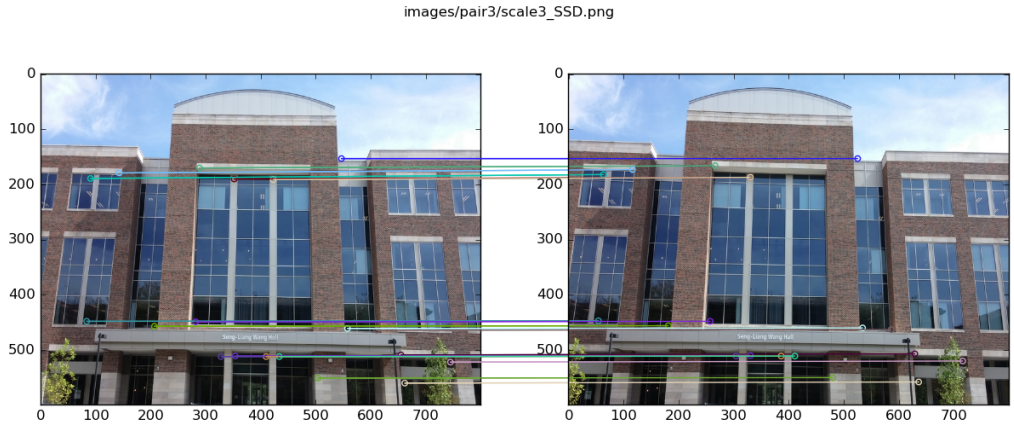


(a) SSD at scale 2

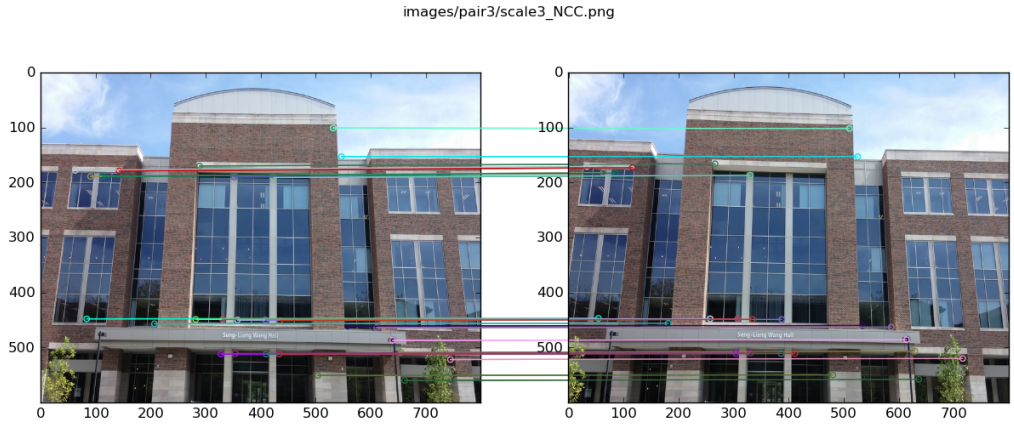


(b) NCC at scale 2

Figure 15: Interest point matching with Harris corner detector at scale 2

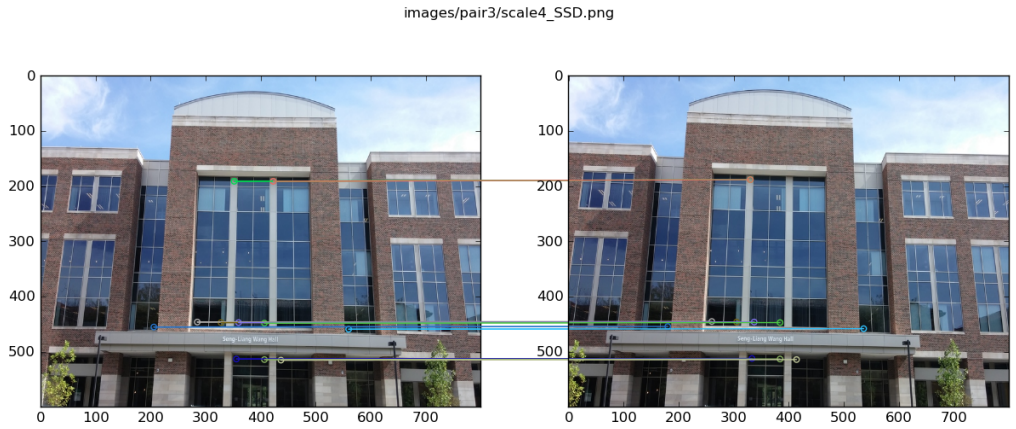


(a) SSD at scale 3

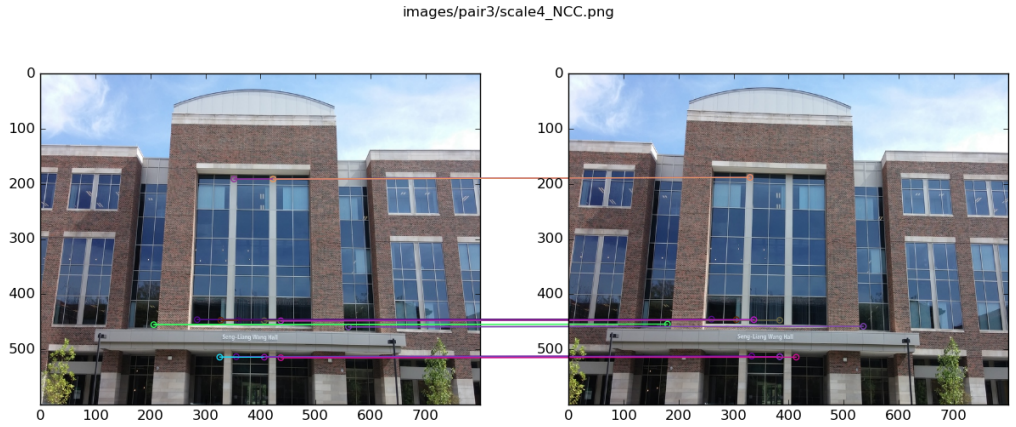


(b) NCC at scale 3

Figure 16: Interest point matching with Harris corner detector at scale 3



(a) SSD at scale 4



(b) NCC at scale 4

Figure 17: Interest point matching with Harris corner detector at scale 4

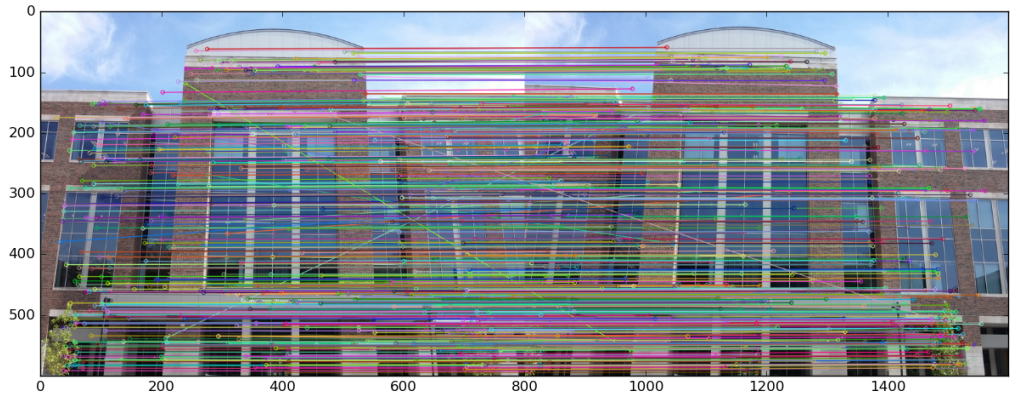


Figure 18: Interest point matching with SIFT

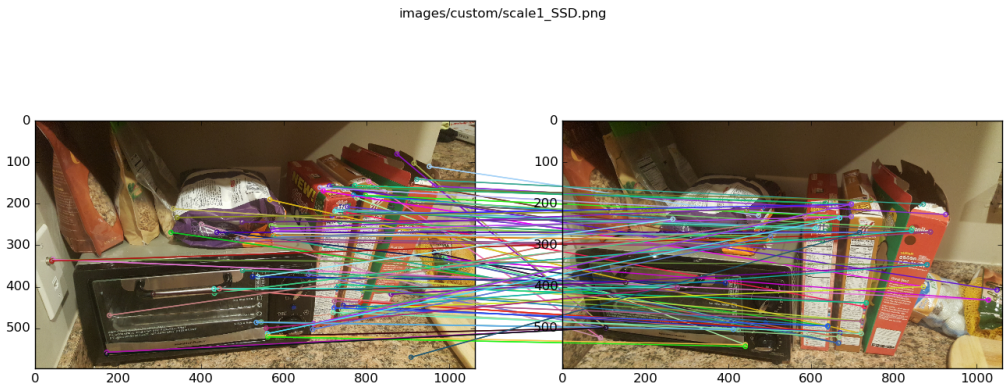
2.5 Custom Image Pair

For this pair, the corner response threshold for the four different scales are chosen to be $T_R = 200, 400, 800, 1600$.

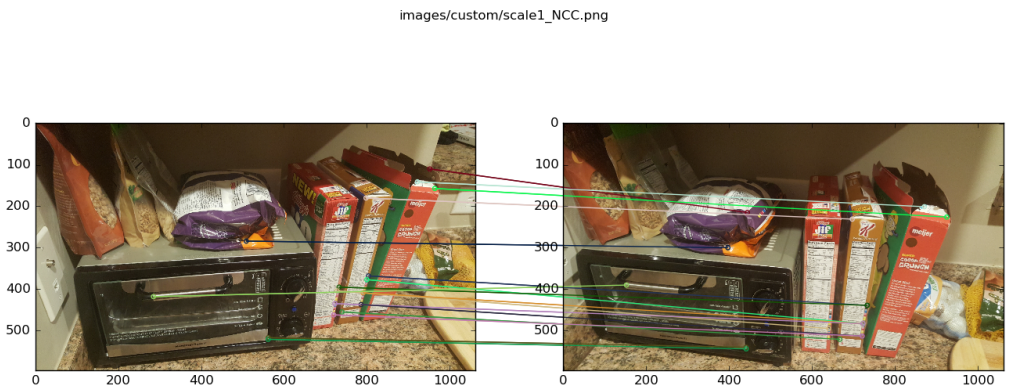
(a) Input image 1

(b) Input image 2

Figure 19: Input images

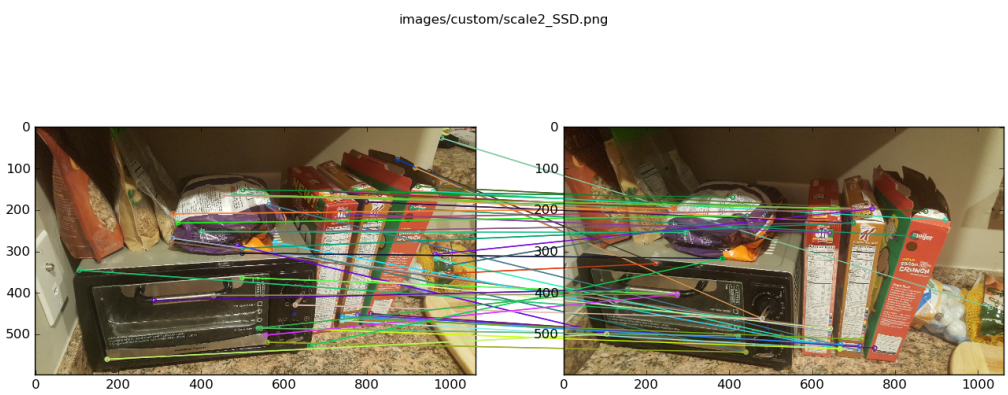


(a) SSD at scale 1

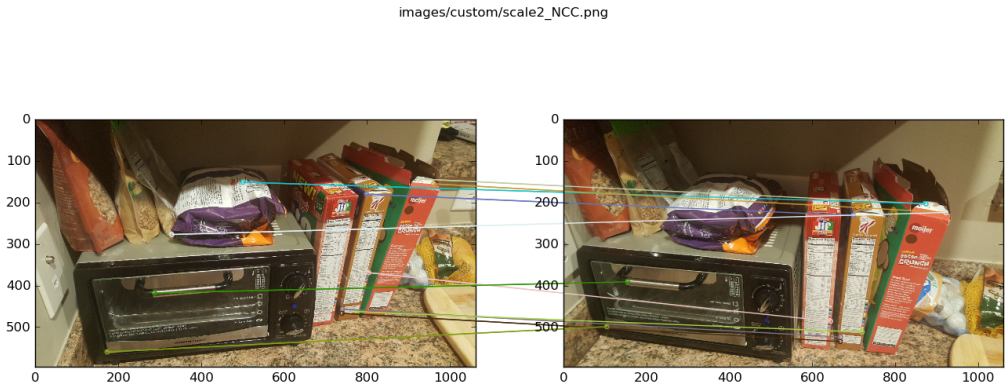


(b) NCC at scale 1

Figure 20: Interest point matching with Harris corner detector at 1

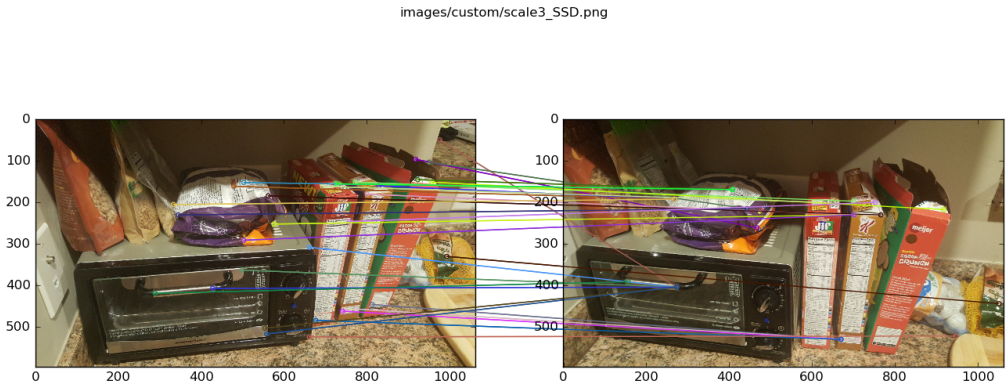


(a) SSD at scale 2

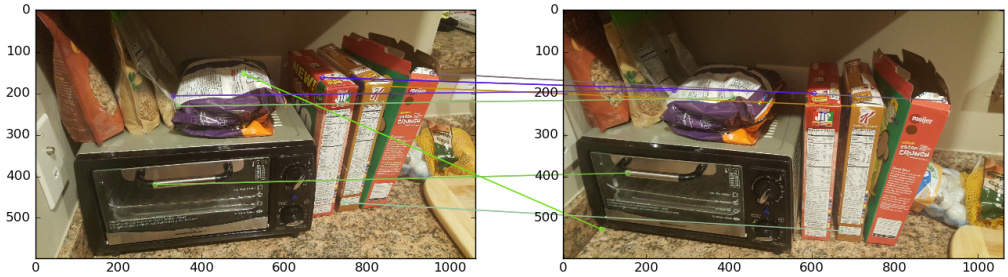


(b) NCC at scale 2

Figure 21: Interest point matching with Harris corner detector at scale 2

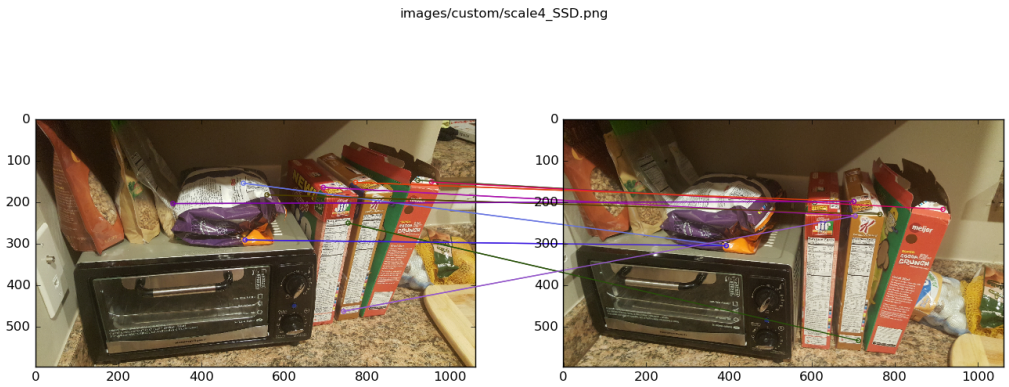


(a) SSD at scale 3

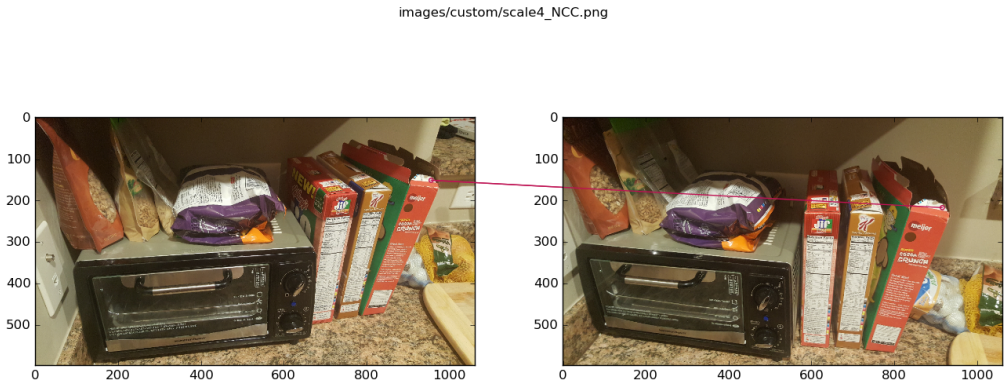


(b) NCC at scale 3

Figure 22: Interest point matching with Harris corner detector at scale 3



(a) SSD at scale 4



(b) NCC at scale 4

Figure 23: Interest point matching with Harris corner detector at scale 4

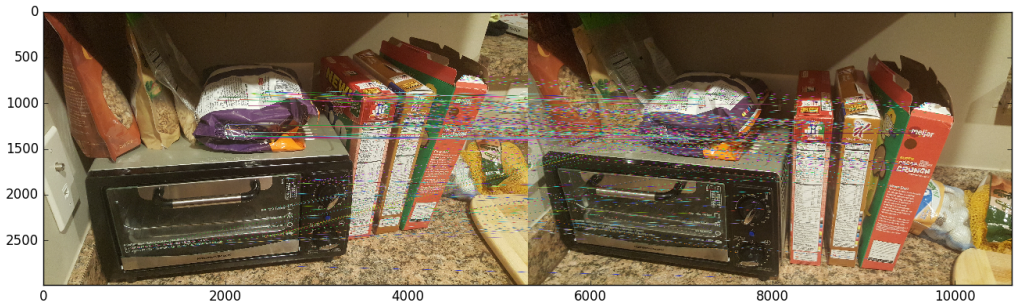


Figure 24: Interest point matching with SIFT

2.6 Discussion

1. As the σ increases in Harris corner detector, less interest points are extracted. In the mean time, the extracted interest points are more visually representative in terms of higher contrast. As seen in Figure 23, at scale 4, the interest points are sparse and have relative higher contrast (white wall, black glass) than the ones from a lower scale (blue sky, brown bricks).

2. Quality control of matching candidates is very critical, as shown in Figure 2a. Compared to NCC, the threshold set for SSD was not able to reject weak matching, which results in the crisscrosses between the two images.

3. Instead of hard-coded T_R, dynamic thresholding could be a possible improvement to account for variant nature of different images, such as illumination, contrast, etc.

4. The implementation of SIFT features with knn matching in OpenCV (using default parameters) gives us consistent high-quality matching in all image pairs. Many SIFT interest points lie on edges with significant contrasts such as the edge between the grass and the sidewalk in Figure 6. On the other hand, Harris corner detector is less sensitive as it tends to ignore edges, as seen in 2a. Both methods are shown to be robust to scale (as shown in pair 2).

3 Source Code

3.1 main.py

```
from harris import get_harris_corners
from metric import get_matching_SSD, get_matching_NCC
from sift import get_sift_kp_des
```

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
from matplotlib.patches import Circle, ConnectionPatch

def resize_image_by_ratio(image, ratio):
        '''
                Resize an image by a given ratio, used for faster debug.
        '''
        return cv2.resize(image, (int(image.shape[1]*ratio),int(image.shape[0]*ratio)))

def find_matching(fpath1, fpath2, feature, metric, resize_ratio, sigma, threshold, match
        '''
                Generic wrapper function for finding matches between interest points i
                @fpath1,fpath2: string of path to the two RGB image files
                @feature: string of feature to use ('Harris', 'SURF', 'SIFT')
                @metric: string of metric to use ('NCC', 'SSD')
                @resize_ratio: double of the actual size of the image to use
                @sigma: double of scale
                @threshold: double of threshold for corners response in Harris
                @match_win_size: int of size of the matching window in metric
        '''
        color1 = cv2.imread(fpath1)
        color1 = resize_image_by_ratio(color1, resize_ratio)
        gray1 = cv2.cvtColor(color1, cv2.COLOR_BGR2GRAY)
        color2 = cv2.imread(fpath2)
        color2 = resize_image_by_ratio(color2, resize_ratio)
        gray2 = cv2.cvtColor(color2, cv2.COLOR_BGR2GRAY)
        if feature == 'Harris':
                # We need double precision
                gray1 = np.double(gray1) / 255.
                gray2 = np.double(gray2) / 255.
                # Find corners in the first image
                features1 = get_harris_corners(gray1, sigma, threshold)
                # Find corners in the second image
                features2 = get_harris_corners(gray2, sigma, threshold)
                # Get the matchings
                if metric == 'SSD':
                        features1, features2 = get_matching_SSD(gray1, features1, gray2,
                elif metric == 'NCC':
                        features1, features2 = get_matching_NCC(gray1, features1, gray2,
                else:
                        print "ERROR: Unknown metric..."
                        return
                # Don't forget to convert r,c format to x,y
```

```
                features1 = [(c[1], c[0]) for c in features1]
                features2 = [(c[1], c[0]) for c in features2]
                # Plot image and mark the corners
                fig, axes = plt.subplots(1,2)
                axes[0].set_aspect('equal')
                axes[0].imshow(cv2.cvtColor(color1, cv2.COLOR_BGR2RGB), cmap='jet')
                axes[1].set_aspect('equal')
                axes[1].imshow(cv2.cvtColor(color2, cv2.COLOR_BGR2RGB), cmap='jet')
                for i in range(0, len(features1)):
                        color = np.random.rand(3,1)
                        pt1 = features1[i]
                        pt2 = features2[i]
                        axes[0].add_patch( Circle(pt1, 5, fill=False, color=color, clip_
                        axes[1].add_patch( Circle(pt2, 5, fill=False, color=color, clip_
                        # Draw lines for matching pairs
                        line1 = ConnectionPatch(xyA=pt1, xyB=pt2, coordsA='data', coords
                        line2 = ConnectionPatch(xyA=pt2, xyB=pt1, coordsA='data', coords
                        axes[0].add_patch(line1)
                        axes[1].add_patch(line2)
                # plt.savefig(fig_name, dpi=100)
                plt.suptitle(fig_name)
                plt.show()
        elif feature == 'SIFT':
                # Find the keypoints and descriptor in one go
                kp1, des1 = get_sift_kp_des(gray1)
                kp2, des2 = get_sift_kp_des(gray2)
                # Find the matchings
                bf = cv2.BFMatcher()
                matches = bf.knnMatch(des1,des2, k=2)
                 # Apply ratio test
                   good = []
                   for m,n in matches:
                       if m.distance < 0.75*n.distance:
                           good.append([m])
                   # Weird fix for cv2.drawMatchesKnn error
                   img3 = np.zeros((1,1))
                   # cv2.drawMatchesKnn expects list of lists as matches.
                   img3 = cv2.drawMatchesKnn(cv2.cvtColor(color1, cv2.COLOR_BGR2RGB),kp1
                                                                   cv2.cvtColor(color2,
                                                                   good,img3,flags=2)

                   plt.imshow(img3),plt.show()
                   return

def pair1():
        # Scale 1
```

```
        find_matching('images/pair1/1.jpg', 'images/pair1/2.jpg', 'Harris', 'SSD',
                                        0.5, 1.2, 10, 20, 'images/pair1/scale1_SSD.png')
        find_matching('images/pair1/1.jpg', 'images/pair1/2.jpg', 'Harris', 'NCC',
                                        0.5, 1.2, 10, 20, 'images/pair1/scale1_NCC.png')

        # Scale 2
        find_matching('images/pair1/1.jpg', 'images/pair1/2.jpg', 'Harris', 'SSD',
                                        0.5, 2.4, 40, 20, 'images/pair1/scale2_SSD.png')
        find_matching('images/pair1/1.jpg', 'images/pair1/2.jpg', 'Harris', 'NCC',
                                        0.5, 2.4, 40, 20, 'images/pair1/scale2_NCC.png')

        # Scale 3
        find_matching('images/pair1/1.jpg', 'images/pair1/2.jpg', 'Harris', 'SSD',
                                        0.5, 3.6, 90, 20, 'images/pair1/scale3_SSD.png')
        find_matching('images/pair1/1.jpg', 'images/pair1/2.jpg', 'Harris', 'NCC',
                                        0.5, 3.6, 90, 20, 'images/pair1/scale3_NCC.png')

        # Scale 4
        find_matching('images/pair1/1.jpg', 'images/pair1/2.jpg', 'Harris', 'SSD',
                                        0.5, 4.8, 160, 20, 'images/pair1/scale4_SSD.png'
        find_matching('images/pair1/1.jpg', 'images/pair1/2.jpg', 'Harris', 'NCC',
                                        0.5, 4.8, 160, 20, 'images/pair1/scale4_NCC.png'

        find_matching('images/pair1/1.jpg', 'images/pair1/2.jpg', 'SIFT', None,
                                    1.0, None, None, None, 'images/pair1/SIFT.png')

def pair2():
        # Scale 1
        find_matching('images/pair2/1.jpg', 'images/pair2/2.jpg', 'Harris', 'SSD',
                                        1.0, 1.2, 100, 20, 'images/pair2/scale1_SSD.png'
        find_matching('images/pair2/1.jpg', 'images/pair2/2.jpg', 'Harris', 'NCC',
                                        1.0, 1.2, 100, 20, 'images/pair2/scale1_NCC.png'

        # Scale 2
        find_matching('images/pair2/1.jpg', 'images/pair2/2.jpg', 'Harris', 'SSD',
                                        1.0, 2.4, 400, 20, 'images/pair2/scale2_SSD.png'
        find_matching('images/pair2/1.jpg', 'images/pair2/2.jpg', 'Harris', 'NCC',
                                        1.0, 2.4, 400, 20, 'images/pair2/scale2_NCC.png'

        # Scale 3
        find_matching('images/pair2/1.jpg', 'images/pair2/2.jpg', 'Harris', 'SSD',
                                        1.0, 3.6, 900, 20, 'images/pair2/scale3_SSD.png'
        find_matching('images/pair2/1.jpg', 'images/pair2/2.jpg', 'Harris', 'NCC',
                                        1.0, 3.6, 900, 20, 'images/pair2/scale3_NCC.png'
```

```
        # Scale 4
        find_matching('images/pair2/1.jpg', 'images/pair2/2.jpg', 'Harris', 'SSD',
                                        1.0, 4.8, 1600, 20, 'images/pair2/scale4_SSD.png
        find_matching('images/pair2/1.jpg', 'images/pair2/2.jpg', 'Harris', 'NCC',
                                        1.0, 4.8, 1600, 20, 'images/pair2/scale4_NCC.png

        find_matching('images/pair2/1.jpg', 'images/pair2/2.jpg', 'SIFT', None,
                                1.0, None, None, None, 'images/pair2/SIFT.png')

def pair3():
        # Scale 1
        find_matching('images/pair3/1.jpg', 'images/pair3/2.jpg', 'Harris', 'SSD',
                                        1.0, 1.2, 100, 20, 'images/pair1/scale1_SSD.png'
        find_matching('images/pair3/1.jpg', 'images/pair3/2.jpg', 'Harris', 'NCC',
                                        1.0, 1.2, 100, 20, 'images/pair1/scale1_NCC.png'

        # Scale 2
        find_matching('images/pair3/1.jpg', 'images/pair3/2.jpg', 'Harris', 'SSD',
                                        1.0, 2.4, 400, 20, 'images/pair3/scale2_SSD.png'
        find_matching('images/pair3/1.jpg', 'images/pair3/2.jpg', 'Harris', 'NCC',
                                        1.0, 2.4, 400, 20, 'images/pair3/scale2_NCC.png'

        # Scale 3
        find_matching('images/pair3/1.jpg', 'images/pair3/2.jpg', 'Harris', 'SSD',
                                        1.0, 3.6, 900, 20, 'images/pair3/scale3_SSD.png'
        find_matching('images/pair3/1.jpg', 'images/pair3/2.jpg', 'Harris', 'NCC',
                                        1.0, 3.6, 900, 20, 'images/pair3/scale3_NCC.png'

        # Scale 4
        find_matching('images/pair3/1.jpg', 'images/pair3/2.jpg', 'Harris', 'SSD',
                                        1.0, 4.8, 1600, 20, 'images/pair3/scale4_SSD.png
        find_matching('images/pair3/1.jpg', 'images/pair3/2.jpg', 'Harris', 'NCC',
                                        1.0, 4.8, 1600, 20, 'images/pair3/scale4_NCC.png

        find_matching('images/pair3/1.jpg', 'images/pair3/2.jpg', 'SIFT', None,
                                1.0, None, None, None, 'images/pair3/SIFT.png')

def custom():
        # Scale 1
        find_matching('images/custom/1.jpg', 'images/custom/2.jpg', 'Harris', 'SSD',
                                        0.2, 1.2, 200, 20, 'images/custom/scale1_SSD.png
        find_matching('images/custom/1.jpg', 'images/custom/2.jpg', 'Harris', 'NCC',
                                        0.2, 1.2, 200, 20, 'images/custom/scale1_NCC.png

        # Scale 2
```

```
        find_matching('images/custom/1.jpg', 'images/custom/2.jpg', 'Harris', 'SSD',
                                        0.2, 2.4, 400, 20, 'images/custom/scale2_SSD.png
        find_matching('images/custom/1.jpg', 'images/custom/2.jpg', 'Harris', 'NCC',
                                        0.2, 2.4, 400, 20, 'images/custom/scale2_NCC.png

        # Scale 3
        find_matching('images/custom/1.jpg', 'images/custom/2.jpg', 'Harris', 'SSD',
                                        0.2, 3.6, 800, 20, 'images/custom/scale3_SSD.png
        find_matching('images/custom/1.jpg', 'images/custom/2.jpg', 'Harris', 'NCC',
                                        0.2, 3.6, 800, 20, 'images/custom/scale3_NCC.png

        # Scale 4
        find_matching('images/custom/1.jpg', 'images/custom/2.jpg', 'Harris', 'SSD',
                                        0.2, 4.8, 1600, 20, 'images/custom/scale4_SSD.pn
        find_matching('images/custom/1.jpg', 'images/custom/2.jpg', 'Harris', 'NCC',
                                        0.2, 4.8, 1600, 20, 'images/custom/scale4_NCC.pn

        find_matching('images/custom/1.jpg', 'images/custom/2.jpg', 'SIFT', None,
                                1.0, None, None, None, 'images/custom/SIFT.png')

if __name__ == '__main__':
        custom()
```

3.2 harris.py

```
#!/usr/bin/python
import numpy as np
import cv2
import time
from matplotlib import pyplot as plt
from matplotlib.patches import Circle

def resize_image_by_ratio(image, ratio):
        '''
                Resize an image by a given ratio, used for faster debug.
        '''
        return cv2.resize(image, (int(image.shape[1]*ratio),int(image.shape[0]*ratio)))

def apply_filter(image, kernel):
        '''
                Convolve the image with filter.
                @image: np.ndarray of input image
                @kernel: np.ndarray of filter
                @return: np.ndarray of filtered image (new storage)
        '''
```

```
        return cv2.filter2D(image, -1, kernel)

def get_integral_image(image):
        '''
                Compute the integral image of the input image.
                @image: np.ndarray of input image
                @return: np.ndarray of integral image
        '''
        return cv2.integral(image)

def apply_haar_filter(image, sigma):
        '''
                Construct two Haar filter based on sigma using convolution,
                one along x direction, another along y direction.
                Then, apply the Haar filters on input image.
                @image: np.ndarray of input image
                @sigma: sigma of kernel
                @return: two np.ndarray images of gradients along row and column direc
        '''
        # Smallest even integer that is greater than 4*sigma
        s = int(4*sigma) + int(4*sigma)%2
        kernel_row = np.ones((s, s))
        kernel_col = np.ones((s, s))
        kernel_row[s/2:,:] = -kernel_row[s/2:,:]
        kernel_col[:,:s/2] = -kernel_col[:,:s/2]
        return apply_filter(image, kernel_row), apply_filter(image, kernel_col)

def apply_haar_filter_integral(int_img, sigma):
        '''
                Construct two Haar filter based on sigma using integral image,
                one along x direction, another along y direction.
                Then, apply the Haar filters on input image.
                @image: np.ndarray of input image
                @sigma: sigma of kernel
                @return: two np.ndarray images of gradients along row and column direc
        '''
        # Smallest even integer that is greater than 4*sigma
        s = int(4*sigma) + int(4*sigma)%2

        return apply_filter(image, kernel_row), apply_filter(image, kernel_col)

def get_covar_matrix(drow2, dcol2, drowcol):
        '''
                Compute covariance matrix of an image patch.
                @drow2, dcol2, drowcol: np.ndarray of precomputed images
```

```
                @return: the covariance matrix M
        '''
        return np.array([
                [np.sum( np.sum(dcol2) ), np.sum( np.sum(drowcol) ) ],
                [np.sum( np.sum(drowcol) ), np.sum( np.sum(drow2) ) ]
                ])

def get_corner_response(covar, k=0.04):
        '''
                Compute the corner reponse using eigen values of the covariance matrix
                @covar: np.ndarray of input covariance matrix
                @return: double of the corner response
        '''
        # Perform eigen decomposition to obtain eigen values
        eigens = np.linalg.eigvals(covar)
        lambda1, lambda2 = eigens[0], eigens[1]
        det = lambda1 * lambda2
        tr = lambda1 + lambda2
        return det - k*tr*tr

def apply_nms(image, corners, win_size):
        '''
                Perform non-maximum compression on input corner image.
                @image: np.ndarray of corner image
                @corners: list of corners to be suppressed
                @win_size: int of the size of local window
                @return: list of suppressed corners
        '''
        (h, w) = image.shape
        hs = int(win_size/2)
        sup_corners = []
        for (r,c) in corners:
                if image[(r,c)] == np.max( image[r-hs:r+hs,c-hs:c+hs] ):
                        sup_corners.append((r,c))
        return sup_corners

def get_harris_corners(image, sigma, threshold):
        '''
                Find the corners in the input image using Harris corner detector.
                @image: np.ndarray of input image, double type
                @sigma: double of scale
                @threshold: double of threshold for corner response
                @return: list of corners
        '''
        start = time.time()
```

```
        corners = []
        (h, w) = image.shape
        # 5*sigma by 5*sigma neighboring window
        # Size should be always odd
        s = int(5*sigma) + (1 - int(5*sigma)%2)
        hs = s/2
        # Blur the image to remove noise
        image = cv2.GaussianBlur(image, (s,s), 0)
        drow_img, dcol_img = apply_haar_filter(image, 1.2)
        # Preprocess the necessary images forms for computing covariance matrix
        drow2_img = np.multiply(drow_img, drow_img)
        dcol2_img = np.multiply(dcol_img, dcol_img)
        drowcol_img = np.multiply(drow_img, dcol_img)
        corner_img = np.zeros((h,w))
        print 'Starting corner detection...'
        # Row means y, col means x
        for r in xrange(hs, h-hs):
                for c in xrange(hs, w-hs):
                        M = get_covar_matrix(drow2_img[r-hs:r+hs,c-hs:c+hs],
                                                        dcol2_img[r-hs:r+hs,c-h
                                                        drowcol_img[r-hs:r+hs,c
                        R = get_corner_response(M)
                        corner_img[r,c] = R
                        if R > threshold:
                                corners.append((r,c))
        corners = apply_nms(corner_img, corners, s)
        print 'Corner detection took', time.time()-start, 'seconds'
        # plt.figure()
        # plt.imshow(corner_img, cmap='gray')
        # plt.title('Corner Response')
        # plt.show()
        return corners

def main():
        ori = cv2.imread('images/pair3/1.jpg')
        ori = resize_image_by_ratio(ori, 1.0)
        image = cv2.cvtColor(ori, cv2.COLOR_BGR2GRAY)
        image = np.double(image) / 255.
        corners = get_harris_corners(image, 1.2, 100)
        fig, ax = plt.subplots(1)
        ax.set_aspect('equal')
        ax.imshow(cv2.cvtColor(ori, cv2.COLOR_BGR2RGB), cmap='jet')
        for (y,x) in corners:
                ax.add_patch( Circle((x,y), 5, fill=False, color=np.random.rand(3,1), cl
        plt.show()
```

```
        # drow_img, dcol_img = apply_haar_filter(image, 1.2)
        # plt.subplot(1,3,1), plt.imshow(cv2.cvtColor(ori, cv2.COLOR_BGR2RGB), cmap='j
        # plt.subplot(1,3,2), plt.imshow(drow_img, cmap='gray')
        # plt.subplot(1,3,3), plt.imsho1w(dcol_img, cmap='gray')
        # plt.show()

if __name__ == '__main__':
        main()
```

3.3 metric.py

```
#!/usr/bin/python
import numpy as np
import cv2
import time
from matplotlib import pyplot as plt
from matplotlib.patches import Circle

def resize_image_by_ratio(image, ratio):
        '''
                Resize an image by a given ratio, used for faster debug.
        '''
        return cv2.resize(image, (int(image.shape[1]*ratio),int(image.shape[0]*ratio)))

def get_patch(image, point, win_size):
        '''
                Convenient function to return the patch centered at point of a certain
        '''
        hs = int(win_size/2)
        r = point[0]
        c = point[1]
        return image[r-hs:r+hs,c-hs:c+hs]

def check_boundary(image, points, win_size):
        '''
                Given the size of window, return the points whose window does not cros
        '''
        hs = int(win_size/2)
        return [pt for pt in points if hs <= pt[0] < image.shape[0]-hs
                                                        and hs <= pt[1] < image.


def SSD(p1, p2):
        '''
                Calculate the SSD between two input image patches.
```

```
                @p1,p2: np.ndarray of two input patches
                @return: double of SSD
        '''
        return np.sum( np.sum( np.square(p1 - p2) ) )

def NCC(p1, p2):
        '''
                Calculate the NCC between two input image patches.
                @p1,p2: np.ndarray of two input patches
                @return: double of NCC
        '''
        m1 = np.mean(p1)
        m2 = np.mean(p2)
        enum = np.sum( np.sum( np.multiply( (p1-m1), (p2-m2) ) ) )
        denom = np.sum( np.sum( np.square(p1-m1) ) ) * np.sum( np.sum( np.square(p2-m2)
        return enum / np.sqrt(denom)

def get_matching_SSD(image1, points1, image2, points2, win_size):
        '''
                Find the matching pairs using SSD metric.
                @image1,image2: np.ndarray of two input gray level images
                @points1,points2: list of interest points
                @win_size: int of size of the neighboring window
                @return: two lists of corresponding points (same index)
        '''
        minInd = []
        minVal = []
        # Perform boundary check
        points1 = check_boundary(image1, points1, win_size)
        points2 = check_boundary(image2, points2, win_size)
        for pt1 in points1:
                p1 = get_patch(image1, pt1, win_size)
                SSDs = [SSD( p1 , get_patch(image2, pt2, win_size) ) for pt2 in points2]
                idx = np.argmin(SSDs)
                minInd.append( idx )
                minVal.append( SSDs[idx] )
        # Threshold based on minimum SSD
        absmin = np.min(minVal)
        threshold = absmin * 5.
        l1 = []
        l2 = []
        for i in range(0, len(points1)):
                if minVal[i] < threshold:
                        l1.append( points1[ i ] )
                        l2.append( points2[ minInd[i] ] )
```

```
        return l1, l2

def get_matching_NCC(image1, points1, image2, points2, win_size):
        '''
                Find the matching pairs using NCC metric.
                @image1,image2: np.ndarray of two input gray level images
                @points1,points2: list of interest points
                @image1,image2: np.ndarray of two input gray level images
                @return: two lists of corresponding points (same index)
        '''
        maxInd = []
        maxVal = []
        # Perform boundary check
        points1 = check_boundary(image1, points1, win_size)
        points2 = check_boundary(image2, points2, win_size)
        for pt1 in points1:
                p1 = get_patch(image1, pt1, win_size)
                NCCs = [NCC( p1 , get_patch(image2, pt2, win_size) ) for pt2 in points2]
                idx = np.argmax(NCCs)
                maxInd.append( idx )
                maxVal.append( NCCs[idx] )
        # Threshold based on maximum NCC
        absmax = np.max(maxVal)
        threshold = absmax * 0.9
        l1 = []
        l2 = []
        for i in range(0, len(points1)):
                if maxVal[i] > threshold:
                        l1.append( points1[ i ] )
                        l2.append( points2[ maxInd[i] ] )
        return l1, l2

if __name__ == '__main__':
        main()
```

3.4 sift.py

```
import cv2

def get_sift_kp_des(image, nfeatures=0):
        '''
                Extract SIFT key points and descriptors from image.
                @image: np.ndarray of input image, double type, gray scale
                @return: tuple of key points and corresponding descriptors
        '''
```

```
sift = cv2.xfeatures2d.SIFT_create(nfeatures=nfeatures)
kps, descs = sift.detectAndCompute(image, None)
print("# kps: {}, descriptors: {}".format(len(kps), descs.shape))
return kps, descs
```