Project # 2: Local Feature Extraction, Detection and Matching

Instructor: V. Paúl Pauca

Jianqiu Xu (Tony)

1. Edges and Corners in Video

```
def print_howto():
    print("""
        Control keys to change the video image:
            1. Canny Edge Detection - press 'c'
            2. Harris Corner Detection - press 'h'
            3. SIFT - press 's'
            4. Quit - press 'q'
    """)
```

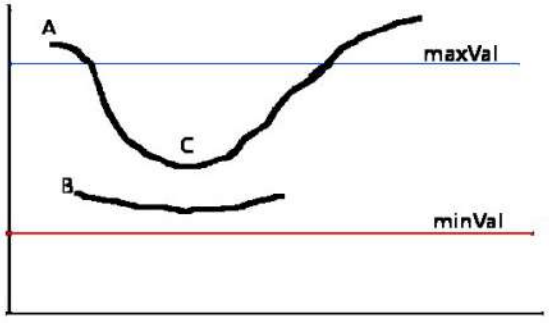
In this part of the program, I first defined several control keys to apply different function to my video image input.

(1) Canny Edge Detection

```
def edge_detect(img):

    # Adjust the high and low threshold
    edge = cv2.Canny(img, 100, 150)
    return edge
```

The canny edge detection function simply used the *cv2.Canny()* code, however, the key to achieve a good edge detection is to adjust the two threshold values to where the algorithm gives meaningful contours of objects regardless of conditions.

The hysteresis thresholding stage of the Canny Edge Detection algorithm decides which edges are edges and which are not, which is defined through the two values in the function, *minVal* and *maxVal*. If intensity gradient of the edge is higher than maxVal, that edge will be shown. Any edge with intensity gradient below the minVal will be considered non-edge, so discarded. For those who lie between the two thresholds, whether they are edges or not is determined by their connectivity to those confirmed edges.



These test below shows the parameter's impact on the edge detection:

I am recording in a indoor, good lighting condition (Min for *minVal*, Max for *maxVal*):

The original image:

The initial parameter setting (Min 100, Max 250)

Decrease Min (Min 10, Max 250)

Increase Max (Min 100, Max 400)

Increase Min (Min 200, Max 250)

Decrease Max (Min 100, Max 150)

It is clear that when I decrease minVal, the Canny detection will include many noises in the image; whereas when I increase maxVal, the detection only detects the most obvious edges and ignores many details in the image. After adjusting the parameters, I increased minVal to eliminate unnecessary noises in the image and decreased the maxVal to include more edges into the detection. When the parameter reached (Min 100, Max 150), the detection image shows a clear edge of my facial features as well as the sofa and window shades behind me. Thus, the appropriate parameter value should be around (Min 100, Max 150).

I repeated the test under poor lighting condition:

(Min 30, Max 150)

(Min 100, Max 100)

(Min 100, Max 300)

(Min 50, Max 150)

Under the poor lighting condition, the edge detection is more difficult to determine the edges. Decreasing the minVal value will result as an increase in noises in facial expression and include unnecessary details, while increasing the maxVal will result in detecting too few edges and a lot of little wiggly edges dude to the edges between minVal and maxVal failed to connect smoothly to the confirmed edges. After trying different combination, the appropriate parameter is around (Min 50, Max 150) in order to provide a clear facial feature and edges for furniture in the back. Since light is limited, we lower the minVal to include more edges into consideration.

When it comes to different scale:

(Min 100, Max 150)

(Min 50, Max 150)

(Min 50, Max 100)

The scaling performance refers to the performance when the same object is in different distance. The same parameter range works very well. Since the lighting is relatively poor, I lowered down the maxVal to include more edges on my face when my face is close to the camera. Thus, the range (Min 50~100, Max 100~150) fits regardless of conditions.

However, we need to use different parameters on blurred image:

I applied a Gaussian blur filter before the video image is used for edge detection. The Gaussian blur filter is set to a kernel of 9 x 9. Since the video image is blurred, the original noises in the image are reduced and edge is less clear.

(Min 50, Max 150)

(Min 10, Max 100)

As I decreased both parameters, the edges can be seen more clearly, since we need to include more edges with smaller intensity gradient. I also experimented on applying gaussian filter with a kernel of 15 x 15 and even larger. Under such condition, we will have to further lower both parameters to detect edge from the blurred image. Thus, we will have to adjust the parameters according to the blur filters and there is not a fixed suitable value.

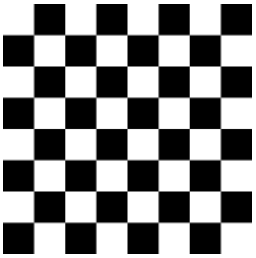
(2) Harris Corner Detection

```
def corner_detect(img, gray_img):

    dst = cv2.cornerHarris(gray_img, blockSize=4, ksize=5, k=0.04)

    # Threshold for an optimal value, it may vary depending on the image.
    img[dst > 0.02 * dst.max()] = [0, 0, 255]
    return img
```

Similar to edge detection, I wrote a function called *corner_detect()* to detect corners in the video image, using OpenCV package code *cv2.cornerHarris()*.

The parameters for this function are:

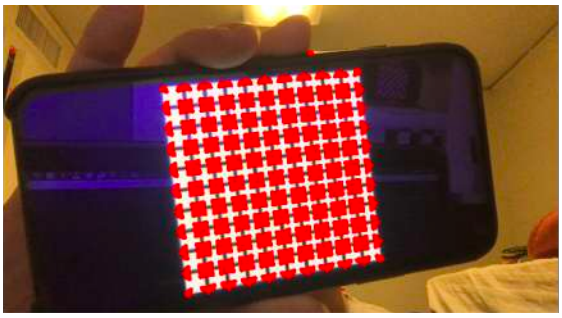
- img - Input image, should be grayscale and float32 type
- blockSize - It is the size of neighborhood considered for corner detection
- ksize - Aperture parameter of Sobel derivative used
- k - Harris detector free parameter in the equation



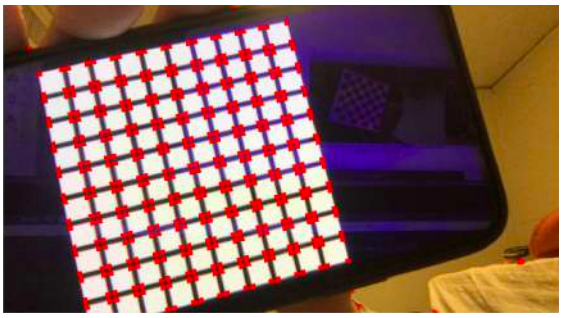
I will use the grids on the right to test my Harris corner detection.

The parameter "ksize" determines the size of the Sobel kernel (3x3, 5x5, …). As the ksize increases, more pixels are part of each convolution process and the edges will get more blurry.

(blockSize = 6, ksize = 31)



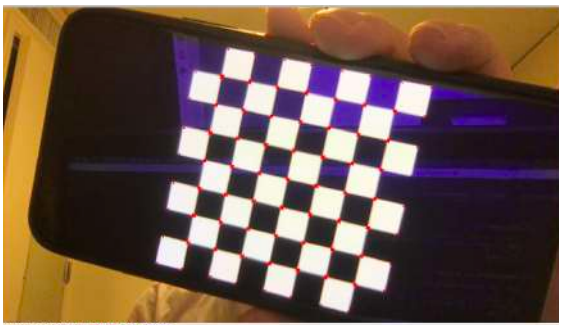
(blockSize = 6, ksize = 3)



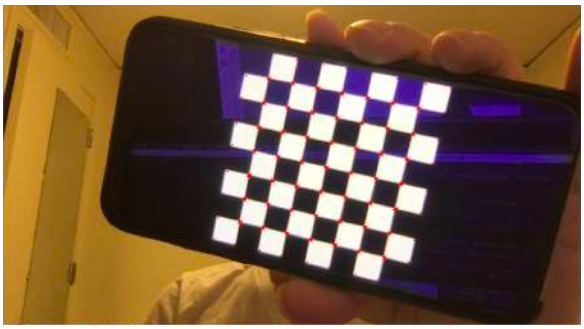
I set ksize to its maximum value 31 and tested it with a blockSize of 8. By comparing it with corner detection with ksize = 3, we can see that higher ksize will end up blurry the corners. So I set ksize to a fix value of 5 so that corners can be recognized easily.

The k parameter here trades off the precision and recall of the corner detection. When k is larger, we can get less false corners but also miss more real corners (high precision), while a smaller k will result in more corners with less precision.

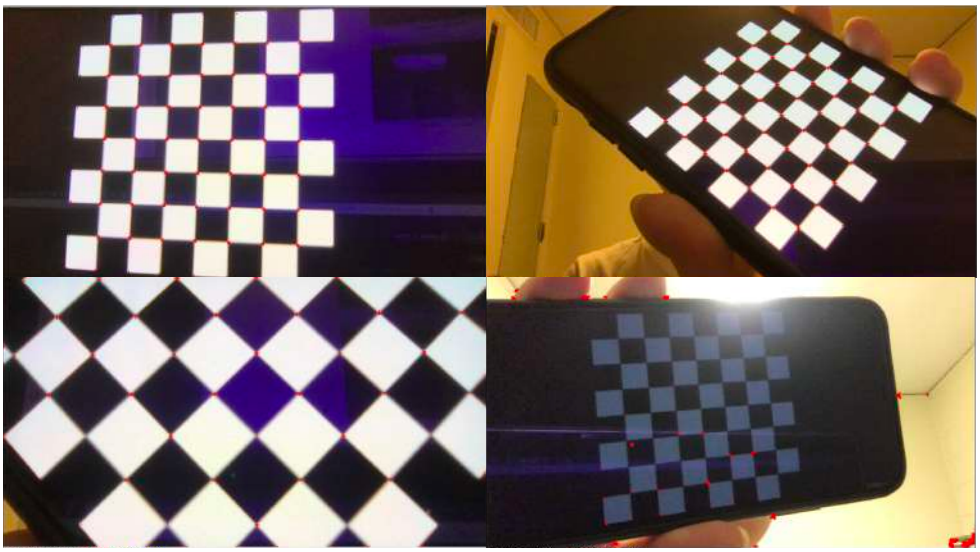
(k = 0.04)



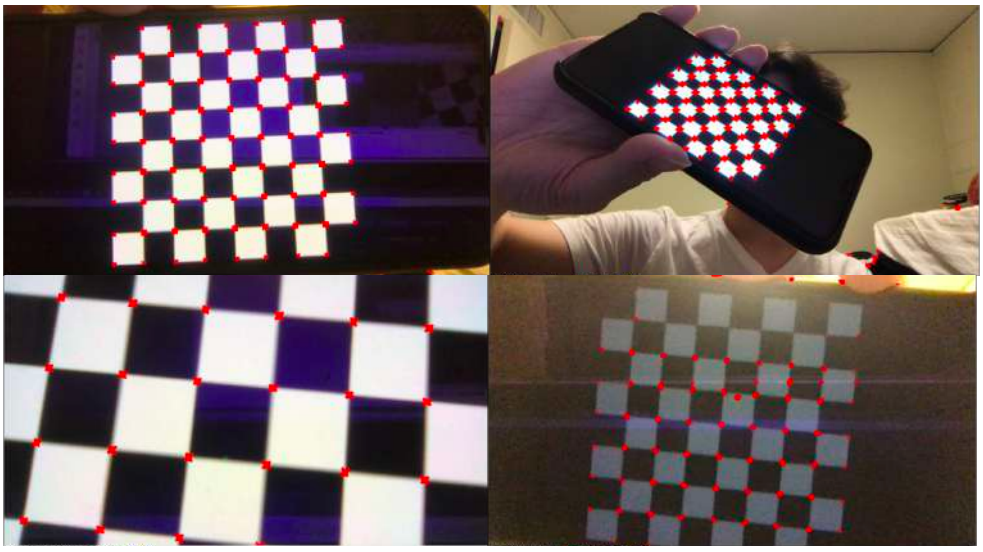
(k = 0.06)



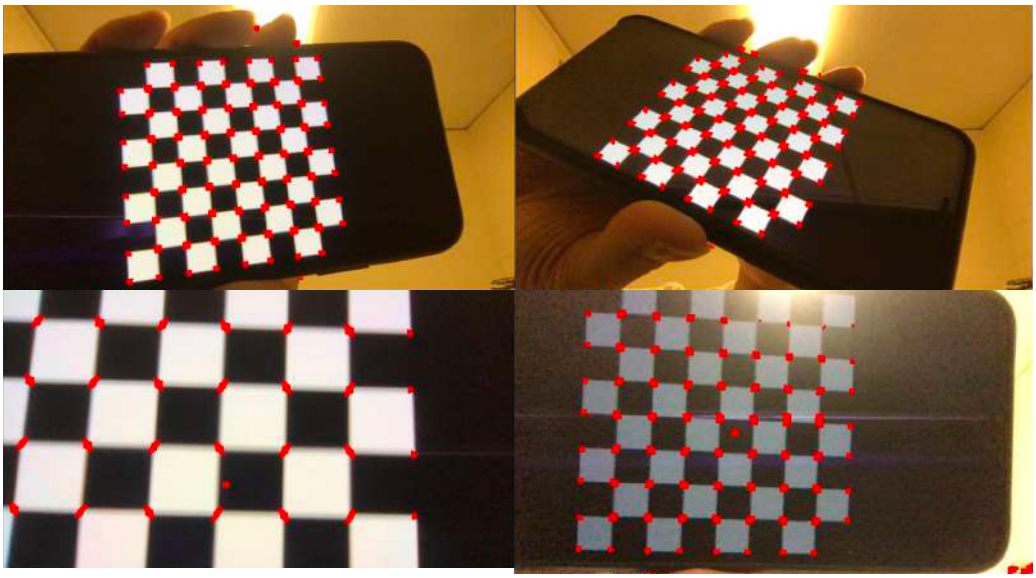
Compare the two images above, one with k value of 0.04 and one with 0.06, while other parameter remains the same. We can see that the image with k of 0.04 detected all the corners on the grids as well as many “corners” on my fingers, whereas image with k of 0.06 are more precise in the corner detection with less corners detected. Thus, I will use k = 0.06 to get more precise result of corners.



(1)



(2)



(3)

The last parameter I experimented is the blockSize for Harris corners. The blockSize is the size of neighborhood considered for corner detection. The three sets of test only changed their blockSize value (Figure 1: blockSize = 2, Figure 2: blockSize = 6, Figure 3: blockSize = 8) to detect the corners of the sample grids in normal, rotation, translation, and poor lighting conditions.

We can see that for blockSize = 2, the corner detection cannot detect some of the corners in the grids when it is placed straight or after rotation. It also has a bad result on detecting corners in poor lighting condition. When the blockSize is changed up to 8, the size of the neighborhood become too big and covered up the corners. Thus, I consider blockSize = 6 as a proper parameter value. The parameters are (blockSize = 6, ksize = 5, k = 0.06).

2. SIFT Descriptors and Scaling

```
def sift_image(img, gray_img):
    sift = cv2.xfeatures2d.SIFT_create(nfeatures=0, nOctaveLayers=3, contrastThreshold=0.04, edgeThreshold=10, sigma=1.6)
    keypoints = sift.detect(gray_img, None)

    cv2.drawKeypoints(img, keypoints, img, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    return img
```

In this part of the code, I used the code in OpenCV package, *sift = cv2.xfeatures2d.SIFT_create(nfeatures=0, nOctaveLayers=3, contrastThreshold=0.04, edgeThreshold=10, sigma=1.6)* in which contains five parameters.

The parameters for this function are:

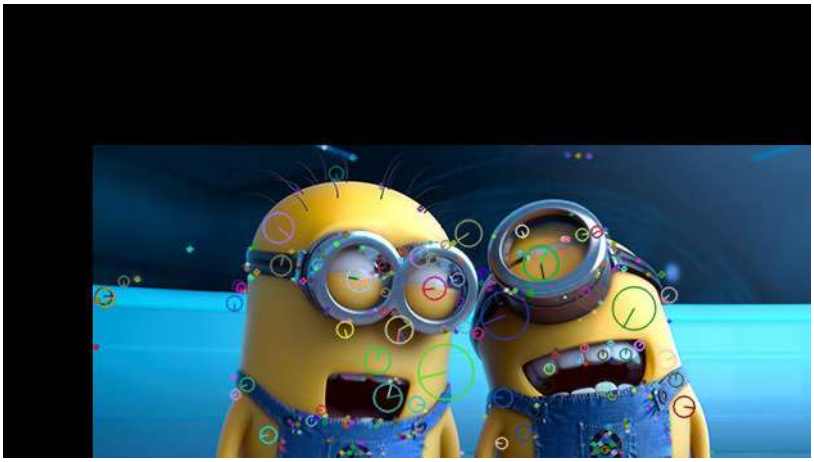
- nfeature = The number of best features to retain.
- nOctaveLayers = The number of layers in each octave.
- contrastThreshold = The contrast threshold used to filter out weak features in semi-uniform regions. The larger the threshold, the less features are produced by the detector.
- edgeThreshold = The threshold used to filter out edge-like features. The larger the edgeThreshold, the less features are filtered out.
- sigma = The sigma of the Gaussian applied to the input image at the octave #0.

I first tried the initial parameter values with the translation, rotation, scale and blur images:

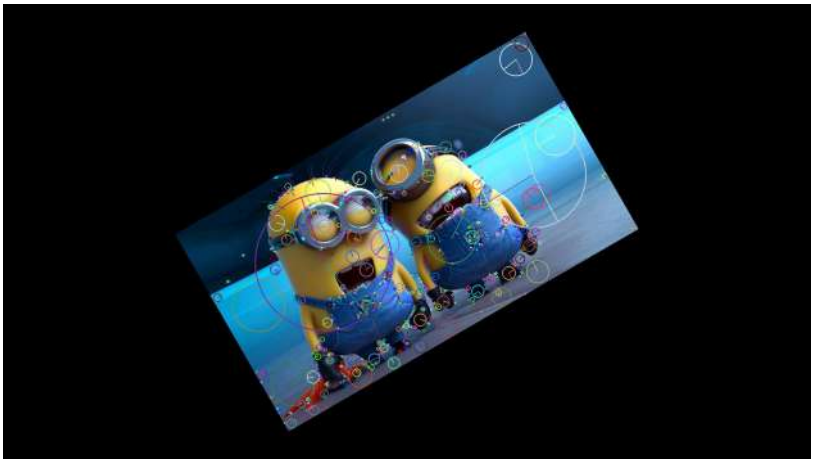
The original image:

This image with SIFT:

The translated image with SIFT:

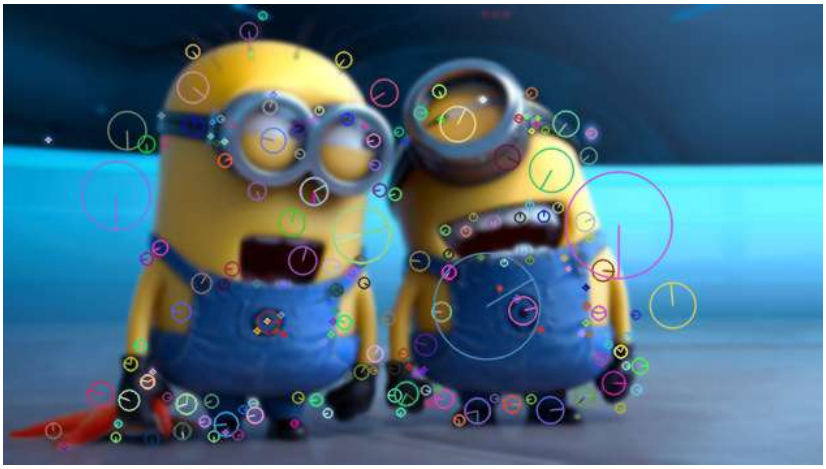


The rotated image with SIFT:

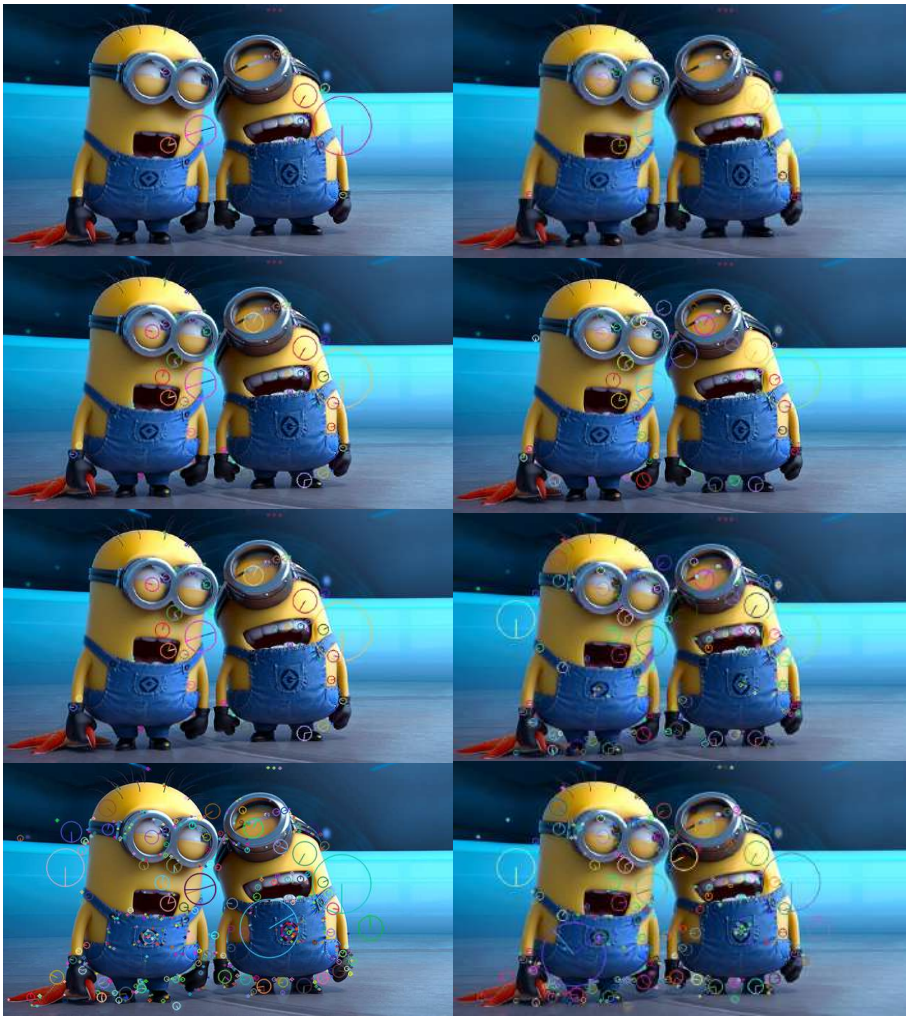


The scaled image with SIFT:

The blurred image with SIFT:



After briefly applying SIFT to the image, I found that the SIFT algorithm preformed very well in identifying the key features of the images in different conditions. Some consistent key features can be found in those result images. Especially for the blurred and scaled images, their key features found by SIFT are almost identical with the original image. The translated image also got the similar result. Although the rotated image has got some different key features compared to those found in original image, they still have a good match even with completely different angles of the objects.

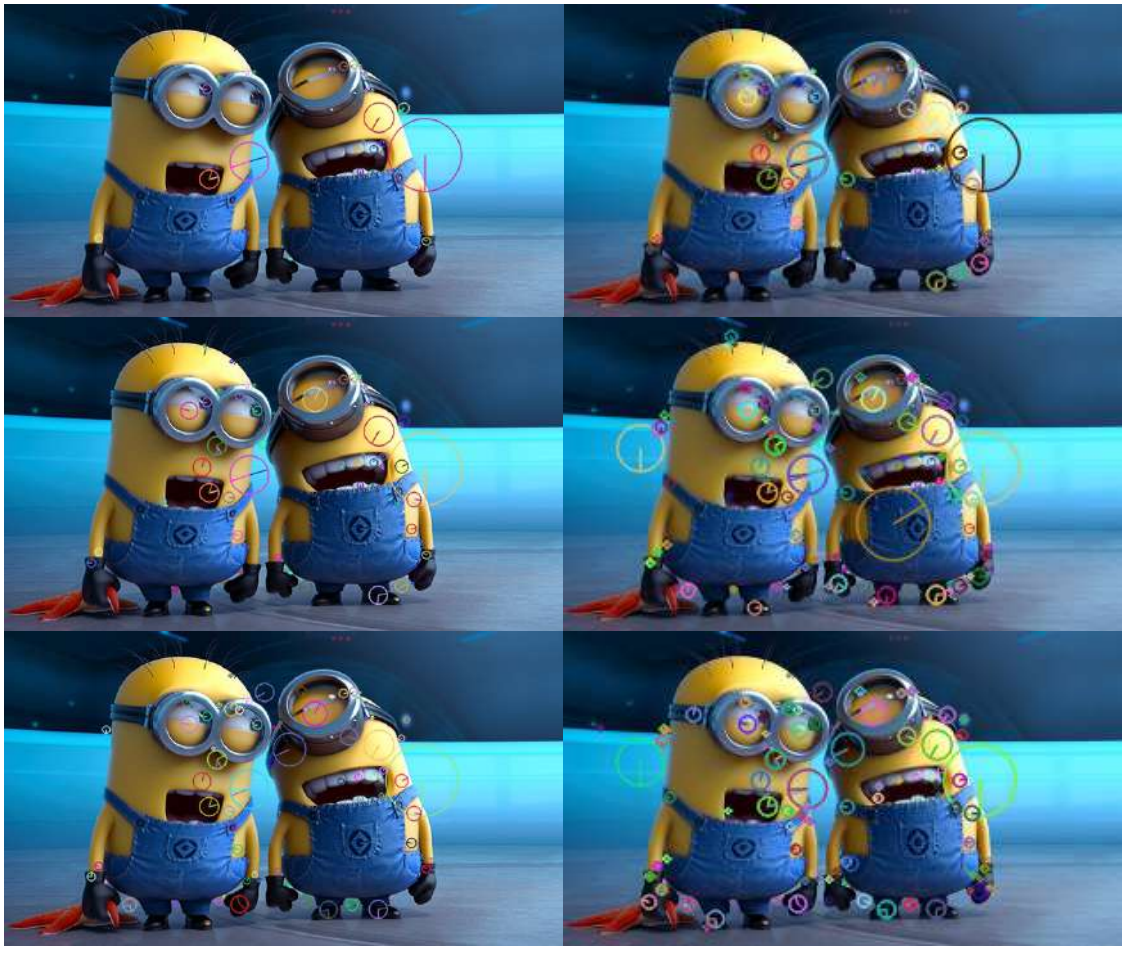


Parameter Value Change	
N (nfeatures), S (Sigma)	
N = 50, S = 1.6	N = 50, S = 2.6
N = 100, S = 1.6	N = 100, S = 2.6
N = 200, S = 1.6	N = 200, S = 2.6
N = 0, S = 1.6	N = 0, S = 2.6

Among the five parameters used in the sift image function, the *nfeatures* and *sigma* are too parameters that causes significant differences on the result. I explored the influence of these two parameters on the SIFT descriptor by adjusting the value of *nfeatures* and *sigma*. The comparison is above.

Since “0” in nfeatures means no restraints in the number of features displayed, which can be consider as the max value of the parameter. As we increase the nfeatures parameter from 50 to 0, the SIFT captures more features and details on the image, while nfeatures = 50 only captures limited major features on the image. When nfeatures value remains the same, we can adjust sigma to see its influence. If the image is captured with soft lenses (soft edges), we might reduce the value of sigma to capture features. However, the images of “minions” have sharp edges, using a higher sigma value can capture more details. Thus, I would choice nfeatures with a value equal or higher than 200, and sigma at around 2.6 to capture more details and features.

Scaled images:



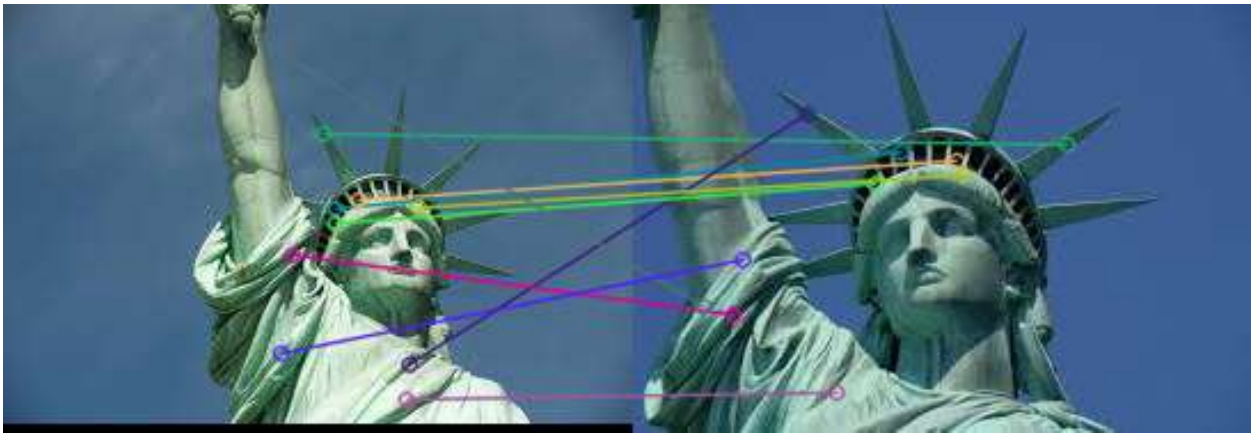
Parameter Value Change	
Original	Scaled
N = 50, S = 1.6	N = 50, S = 1.6
N = 100, S = 1.6	N = 100, S = 1.6
N = 100, S = 2.6	N = 100, S = 2.6

I applied the same SIFT to the scaled images and compared them with the the original images after applying SIFT. We can see that since scaled images have blurry edges, using a smaller sigma is a better idea to detect useful features. Also, as the images are scaled, the there will be more features found on the scaled images come to their original images. Thus, we should we a sift function with smaller values for both nfeatures and sigma on scaled images compared to the original images.

3**. Keypoints and Matching**

(1) Same Object with Different Distances and Angles

SIFT Keypoints Matching:



Harris Corner Keypoints Matching:

I applied both SIFT Keypoints Matching and Harris Corner Keypoints Matching to match the Statue of Liberty with two different distances and angles. The SIFT Keypoints Matching looks at different features of the object, including cloth, hair, crown, and etc. It did a good job in matching the features on the cloth and crown, but there are also a few mismatches. Harris Corner

Matching looks for specific corners on Statue of Liberty's cloth and crown. The matched corners have a very high precision. However, the corners are limited to a small part of the object, and Harris Corner Matching failed to match more general parts of the Statue of Liberty.

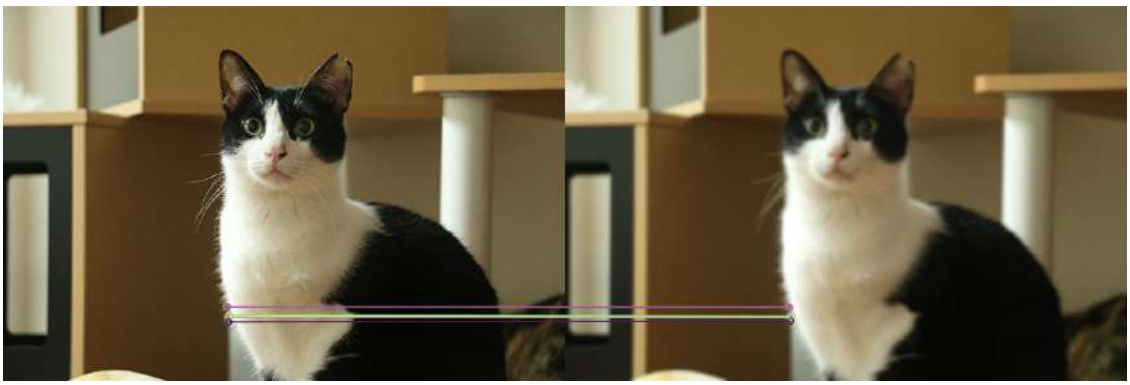
(2) SIFT and Harris

Image of a cat:

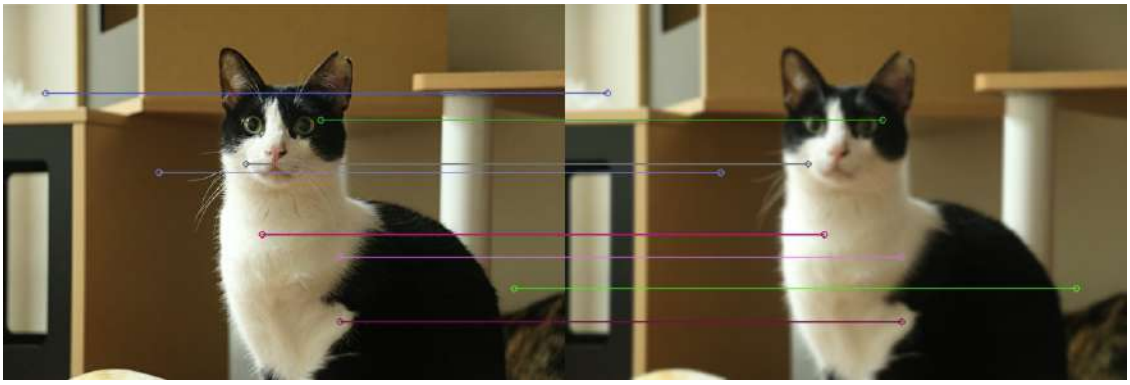
Harris Corner Matching:

SIFT Keypoints Matching:

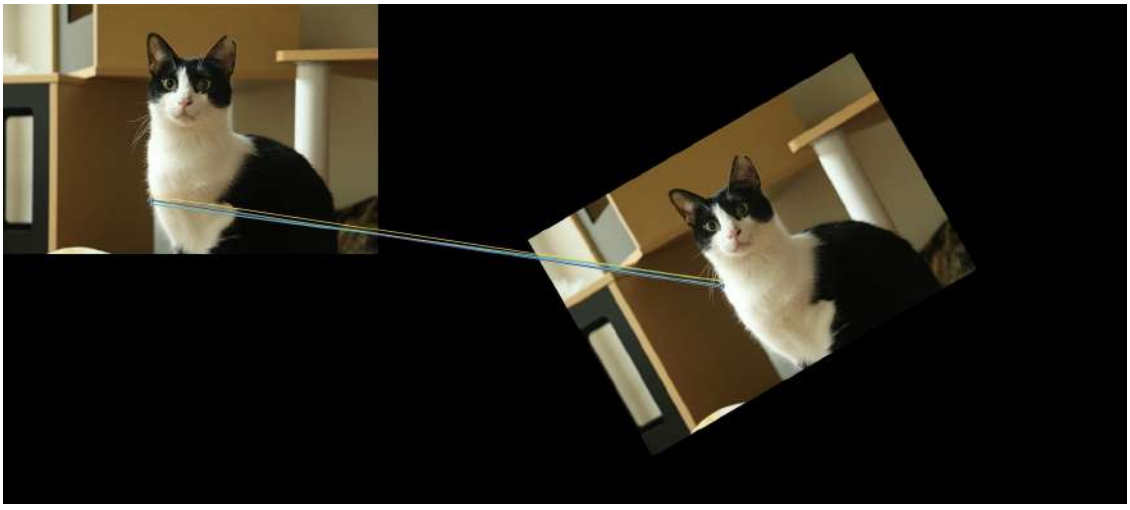
Harris Corner Matching (blur):



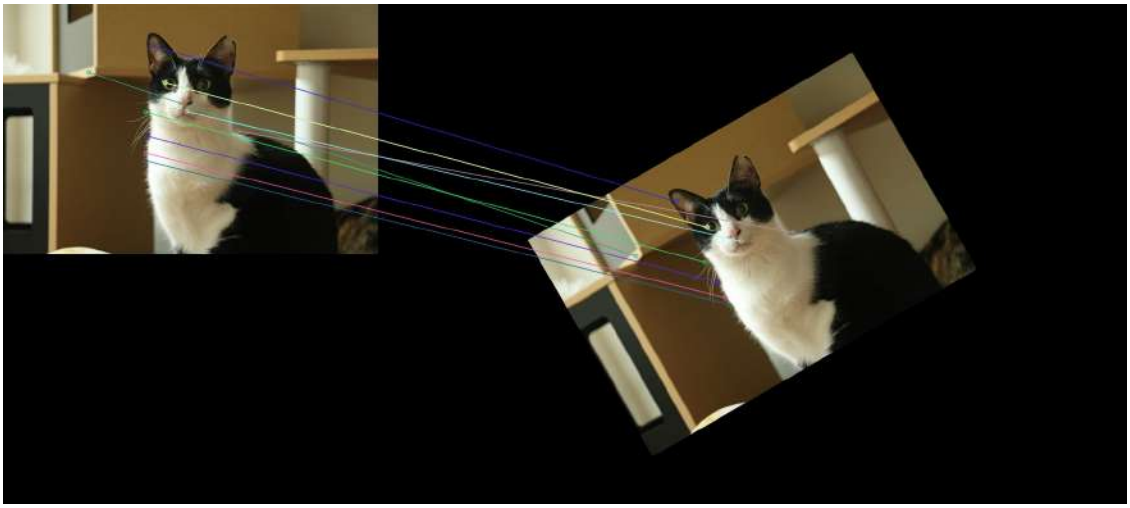
SIFT Keypoints Matching (blur):



Harris Corner Matching (rotation):

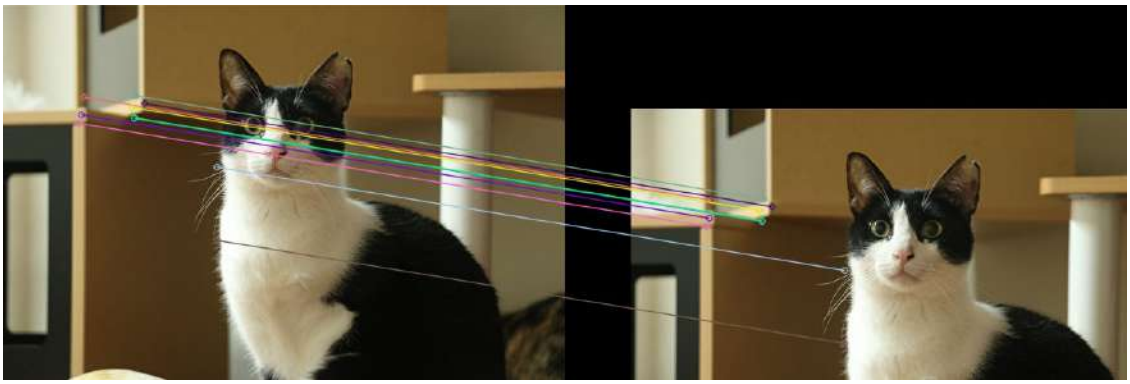


SIFT Keypoints Matching (rotation):



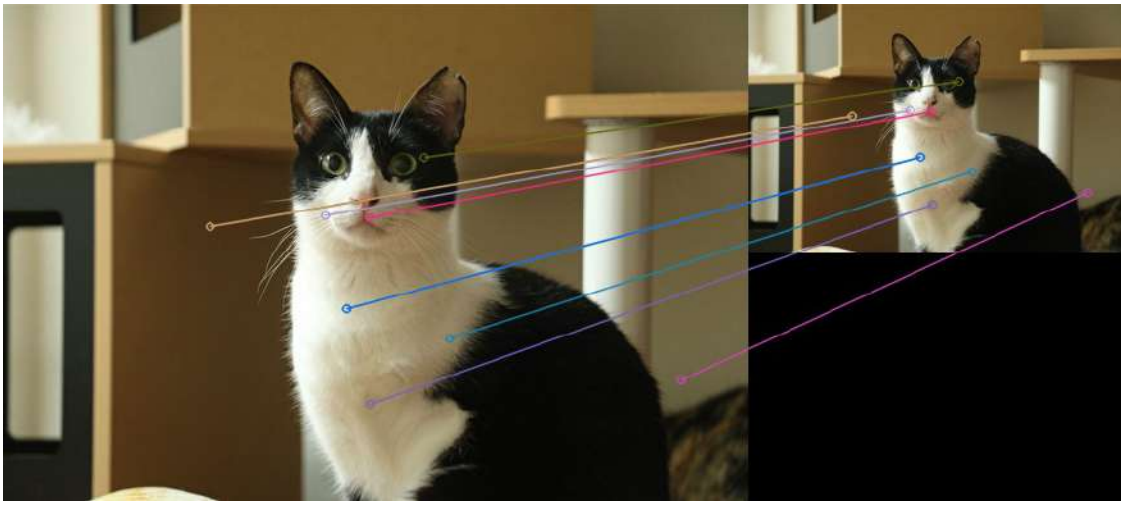
Harris Corner Matching (transition):

SIFT Keypoints Matching (transition):



Harris Corner Matching (scale):

SIFT Keypoints Matching (scale):



From the examples above, we can see that the performance of SIFT Keypoints Matching is definitely better than that of Harris Corner's.

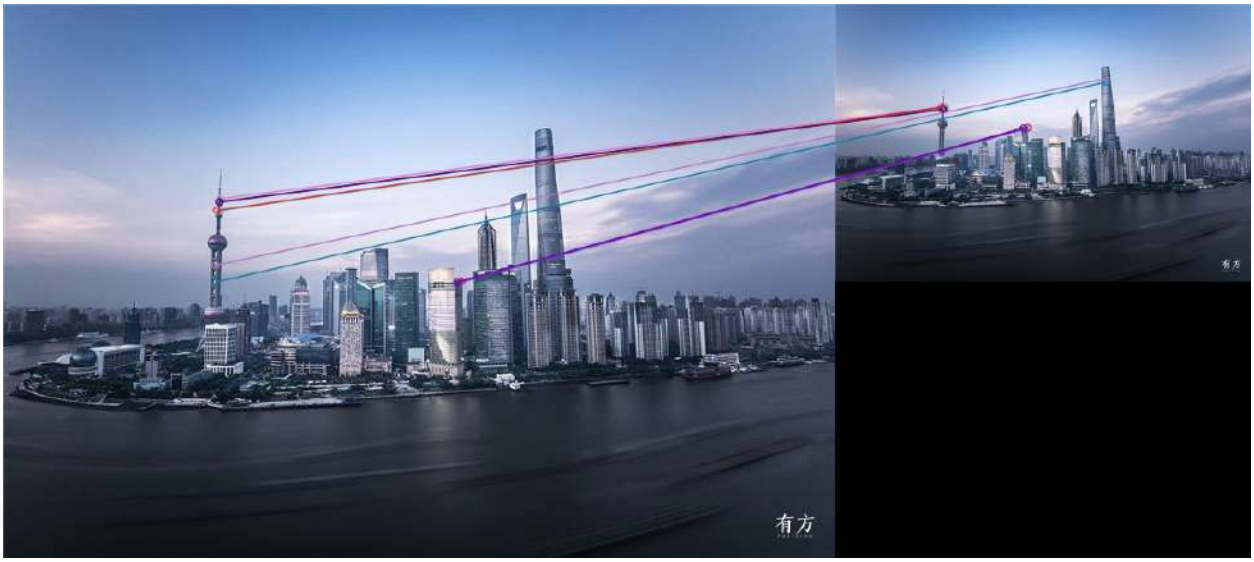
As we look into the accuracy of both matching method, we found that SIFT Keypoints Matching almost perfectly handled all the listed conditions. SIFT can find the exact position in both images and make matches, while Harris Corner Matching is not always reliable.

The cat image is a quite simply structured image with clear edges and objects. However, when Harris is used to compare original image with the rotated image, the corners are clearly mismatched. It might be reasonable since the texture of cats are identical, however, SIFT still does a better job under the same condition. We also have to note that SIFT can match the features

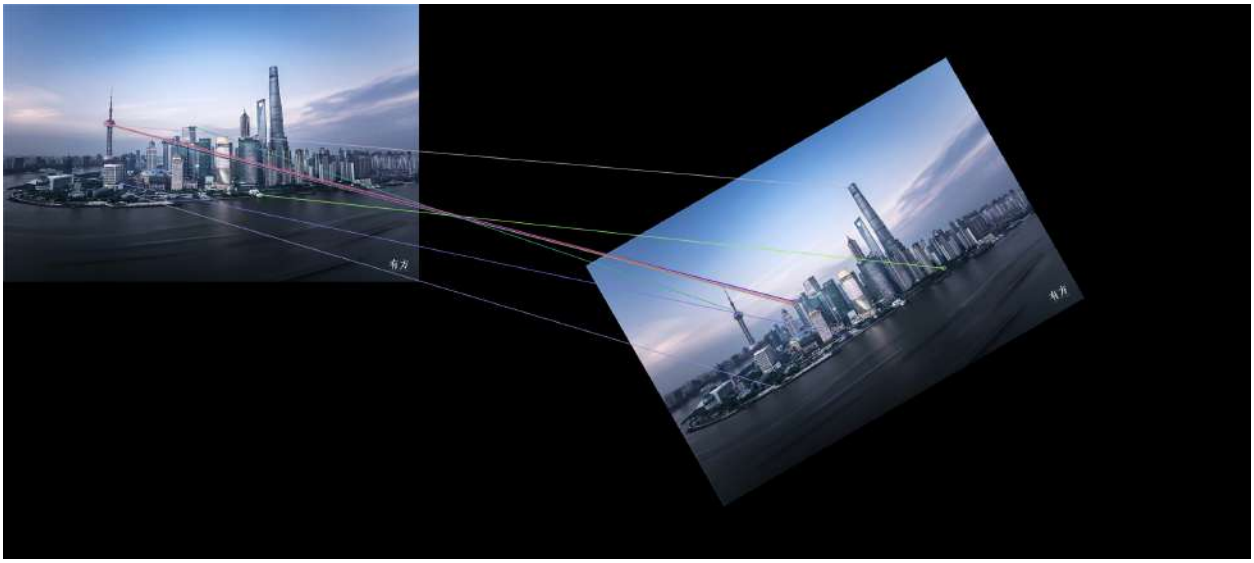
from various parts of the objects and features with different textures, while Harris only matches similar corners within a small part of the images.

I also implemented the both matching to a more complexed images of city views:

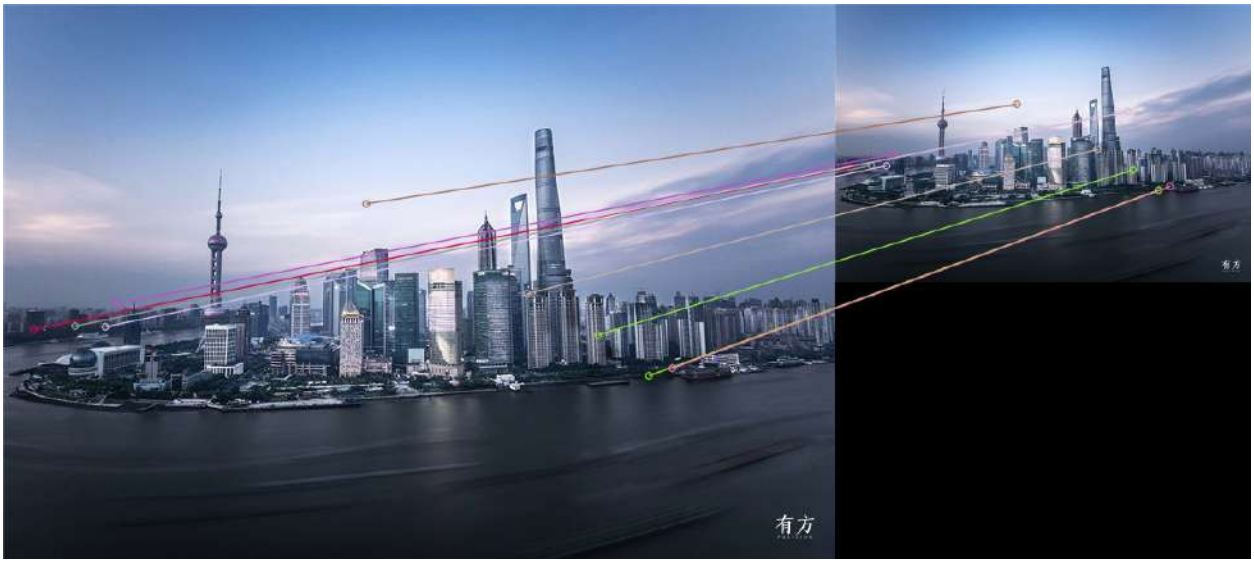
Harris Corner Matching (scale):



Harris Corner Matching (rotation):



SIFT Keypoints Matching (scale):



SIFT Keypoints Matching (rotation):

As what I expected, Harris Corners matching performed poorly on rotated images and scaled images. It performs well in detecting corners of regular images, for example buildings, but more mismatches occur when the images are scaled or rotated.

On the other hand, SIFT Matching can easily match rotated and scaled images with little mismatches, which is proven in multiple examples.