Coursework 2:
Local Feature Detection and matching for object recognition

Rui Xu 10891143

Abstract

This report presents a series of experiments involving various interest point detectors, descriptors, and matchers. These experiments are conducted on identical pairs of images provided by Blackboard, and their outcomes are shown below. Each section encompasses relevant analyses alongside experimental findings, with the implementation code provided in the appendix.

Preparation

For the purpose of this assignment, Python 3.9.6 has been selected as the implementation language, with a majority of functions developed from OpenCV. Functions that implement interest point detection or feature matching are **made from scratch myself**.

Implementation of Harris Interest Point Detector

The primary procedure for feature detection involves several steps: Initially, the gray-scaled image is subjected to blurring; then, the image derivatives in the x and y directions (Ix and Iy) are computed utilizing Sobel operators. Following this, combinations of the image derivatives are calculated, and a Gaussian blur is applied to these combinations to reduce image noise and detail. Reflective Image padding is employed. Subsequently, the determinant and a trace of the matrix M are computed, and the corner strength function $R = \det(M) - \alpha \times \mathrm{trace}(M)^2$ is calculated for each pixel. The Harris Corner Detection algorithm is based on the local variations in the image intensity, which are captured by the corner strength function. Finally, a custom implementation is employed with thresholding to filter the corners. The function identifies local maxima in the corner strength function that are greater than the threshold. These local maxima are potential corners in the image. Key points are subsequently generated from the resultant filtered values.

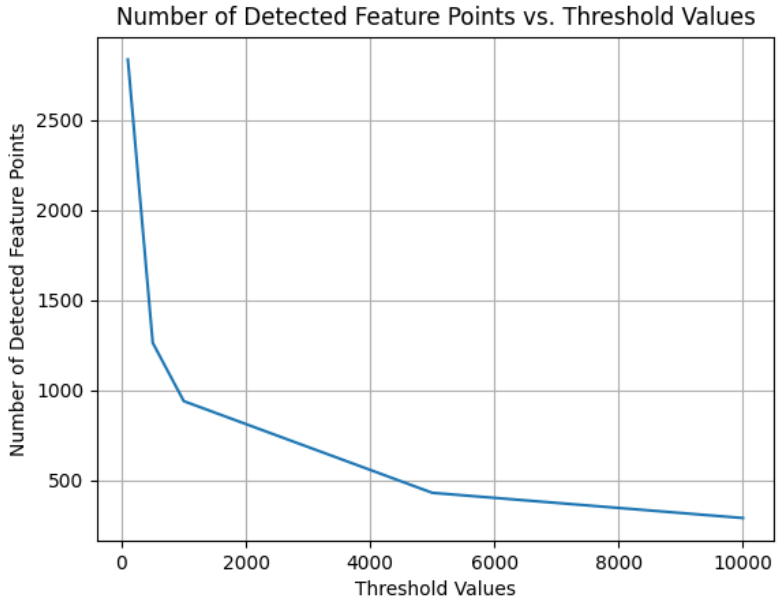
Comparison of the threshold value for Keypoint Detection

The threshold directly affects the quantity of detected keypoints. Lower thresholds result in a higher number of keypoints detected, whereas higher thresholds reduce the number of keypoints. It also influences the quality of detected keypoints. Lower thresholds may lead to the detection of more noise or non-feature points, decreasing the overall quality. A set of threshold values is tested and used for comparison. From the image below, we can see the variation in the number of keypoints as the threshold value is adjusted. We observe a sharp decrease at the beginning. However, as the values increase, they tend to converge, resulting in

the emergence of constant lines towards the end. The optimal threshold chosen is 18000, determined empirically by human observation to minimize the detection of features outside Bernie's shape.

1e2 5e2 1e3

5e3 1e4 5e4



Comparison of different interest point detector/descriptor

In this phase, We opted for the pre-existing ORB framework provided within the OpenCV package. In the figure below, we present comparisons between the keypoints identified by our implementation and those identified by ORB' built-in setting (Harris and FAST interest point detector). In summary, the outcomes produced by OpenCV's FAST and Harris algorithms exhibit notable similarities. Both methodologies effectively identify interest points, particularly corners, within the image. We can observe that the overall performance of the self-designed keypoint detector is similar to that of the ORB built-in detector from OpenCV. However, OpenCV's detection proves to be more precise and robust. Upon visual inspection, it becomes apparent that in my implementation, the identified interest points on the legs and chair appear sparser and somewhat less accurate. Conversely, in the other two images, these interest points are distinct and prominent. However, accurately determining their authenticity solely through visual inspection presents challenges due to the lack of ground truth data.

Self-implemented Harris

Built-in FAST

Built-in Harris

Feature matching

The final stage involves matching the features extracted from the reference image with those from the benchmark images. This process entails computing the **sum of squared differences** between corresponding feature windows. It is executed for all possible combinations of features, and the smallest distance is selected. Given the potential for ambiguous outcomes, the **second smallest distance** is also considered, and a ratio is computed between the first and second distances. For our implementation, we set the threshold value for the ratio test at 0.7.

Comparison of different matching methods

Initially, we will conduct a comparative analysis of two matching methodologies: SSD with Ratio test and SSD without Ratio test. The SSDFeatureMatcher produces a sparse matching

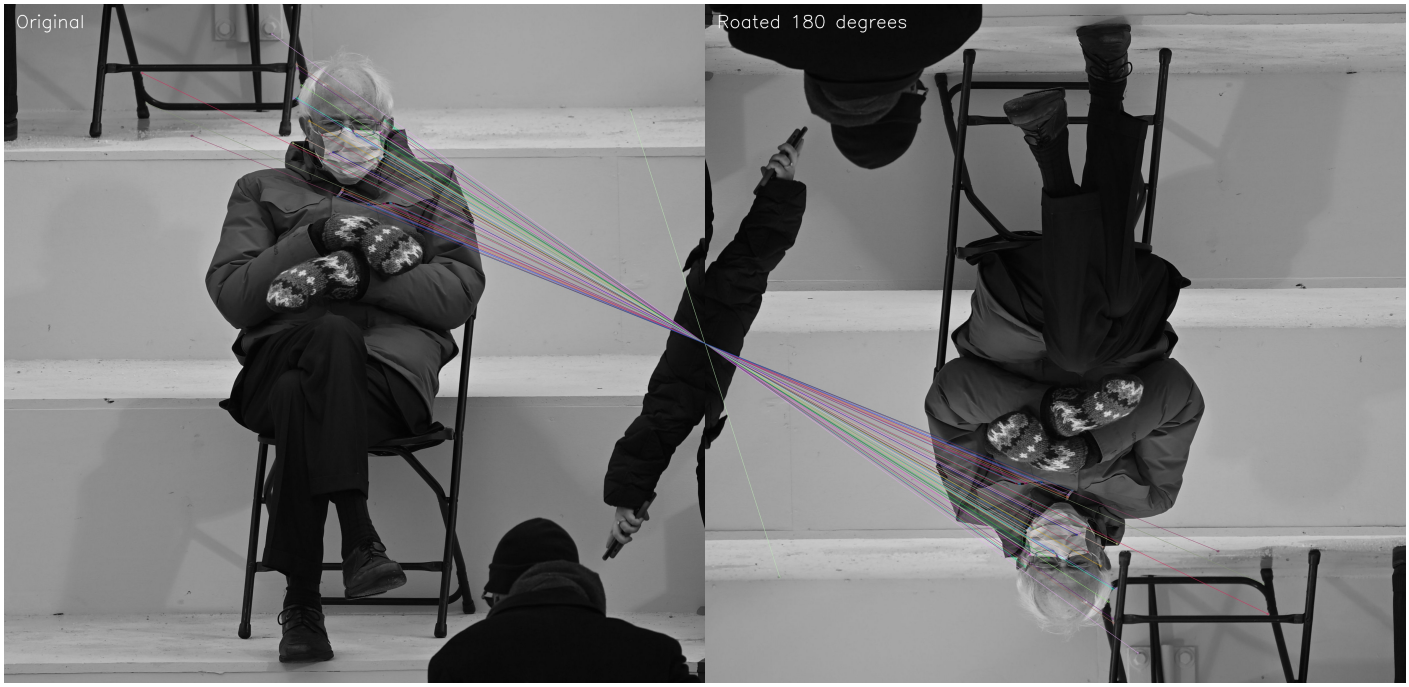
effect. Each feature point in the image is connected to its best matching point in the other image, resulting in a clean and straightforward visualization of the matching pairs.

However, the RatioFeatureMatcher generates a denser matching effect. It considers both the best and the second-best matches for each feature point and establishes connections based on the distance ratio between them. As a result, multiple potential matches can be associated with each feature point, leading to a denser visualization of matched pairs.

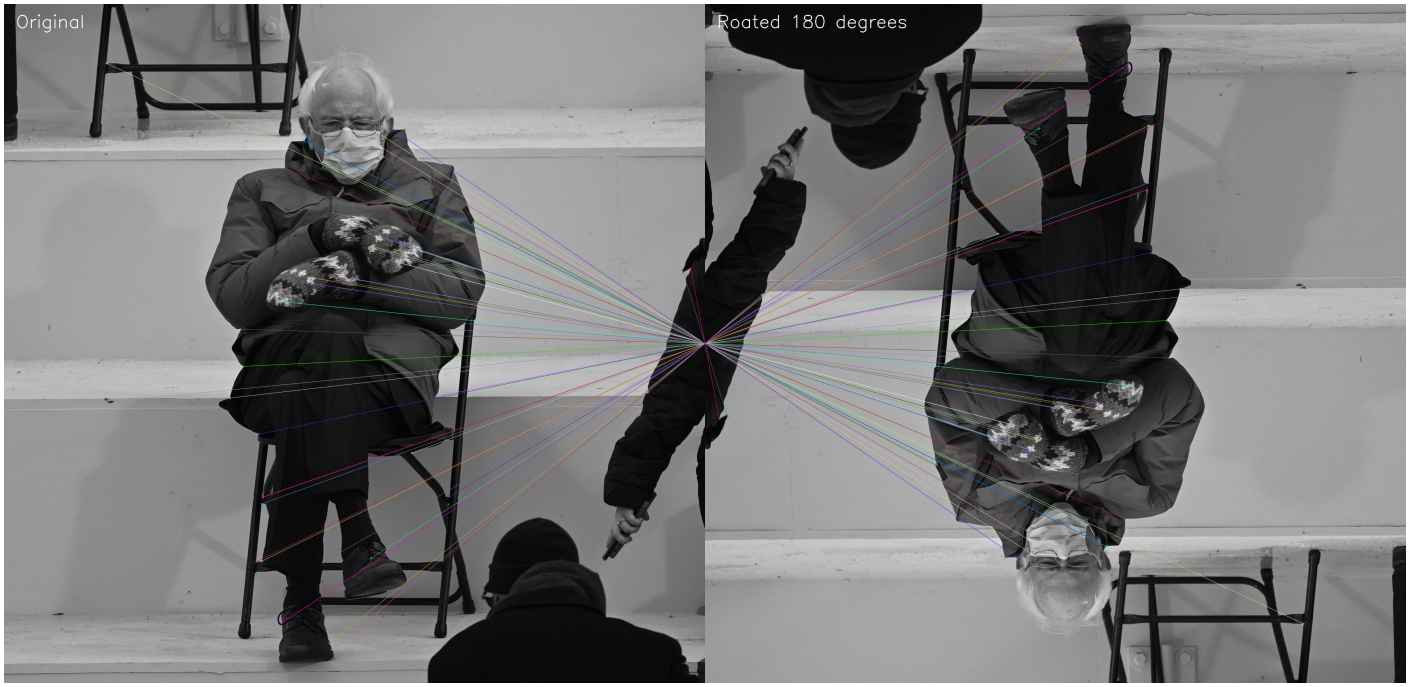
Compare all images in the benchmark set for Feature Matching

180 degree Rotated

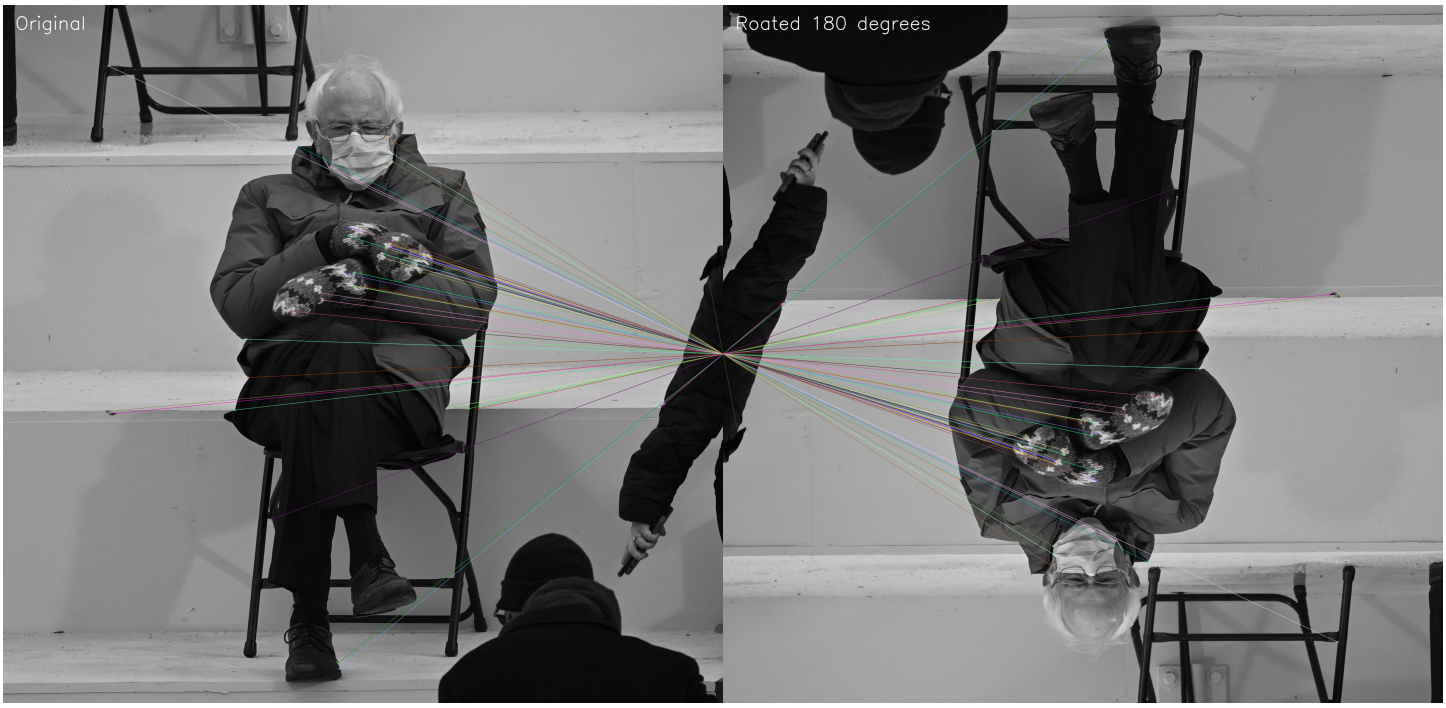
The matcher correctly matches all the features because there is only one variation in rotation.
The effect of the self-implemented detector is very similar to that of the built-in detector.



self-implemented Harris



built-in FAST



built-in Harris

Noisy image

Because of the noise within the image, certain pixels may be misinterpreted, resulting in errors during detection. In this set of comparisons, the effect of the built-in detector is much better than that of the self-implemented detector.

self-implemented Harris

built-in FAST

built-in Harris

Pixelated

This case exhibits similarities to the previously mentioned noisy image, although with a less prominent pixelated pattern, consequently yielding higher accuracy in match detection. But overall, the effect of the built-in detector is slightly better than that of the self-implemented detector.

self-implemented Harris

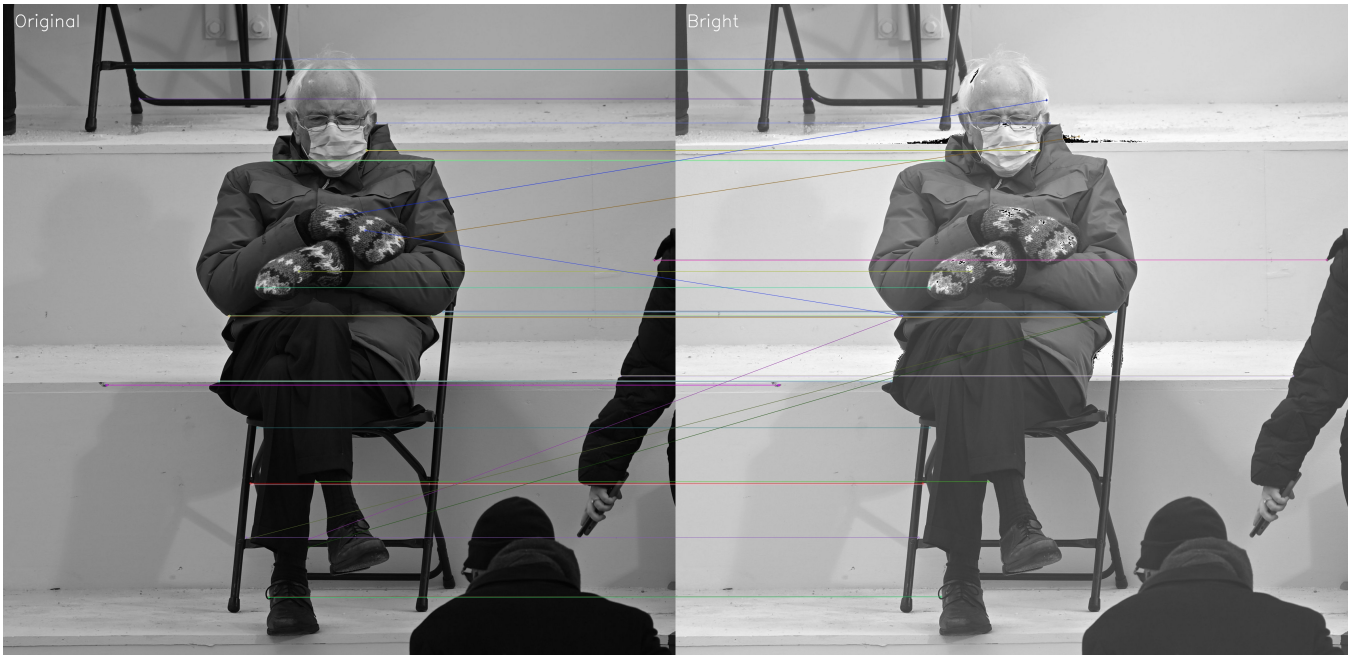
built-in FAST

built-in Harris

Bright

In these images, characterized by intense brightness variations, FAST exhibited limitations in accurately identifying corners compared to Harris. The Harris detector effectively leverages these gradients to accurately identify corners, as it considers both the magnitude and spatial distribution of gradients. For FAST detector, the intensity comparisons may not yield reliable corner detection results when the brightness levels are uniformly high across the image, and subtle intensity variations indicative of corners are not adequately distinguished.

self-implemented Harris



built-in FAST

built-in Harris

Dark

In my observation, when encountering images with strong black-and-white contrast like the images below, We noticed that the Harris corner detector generally performs better than FAST. While FAST is efficient and effective in many scenarios, We found that it may struggle with strong contrast images because it doesn't explicitly consider the gradient information.

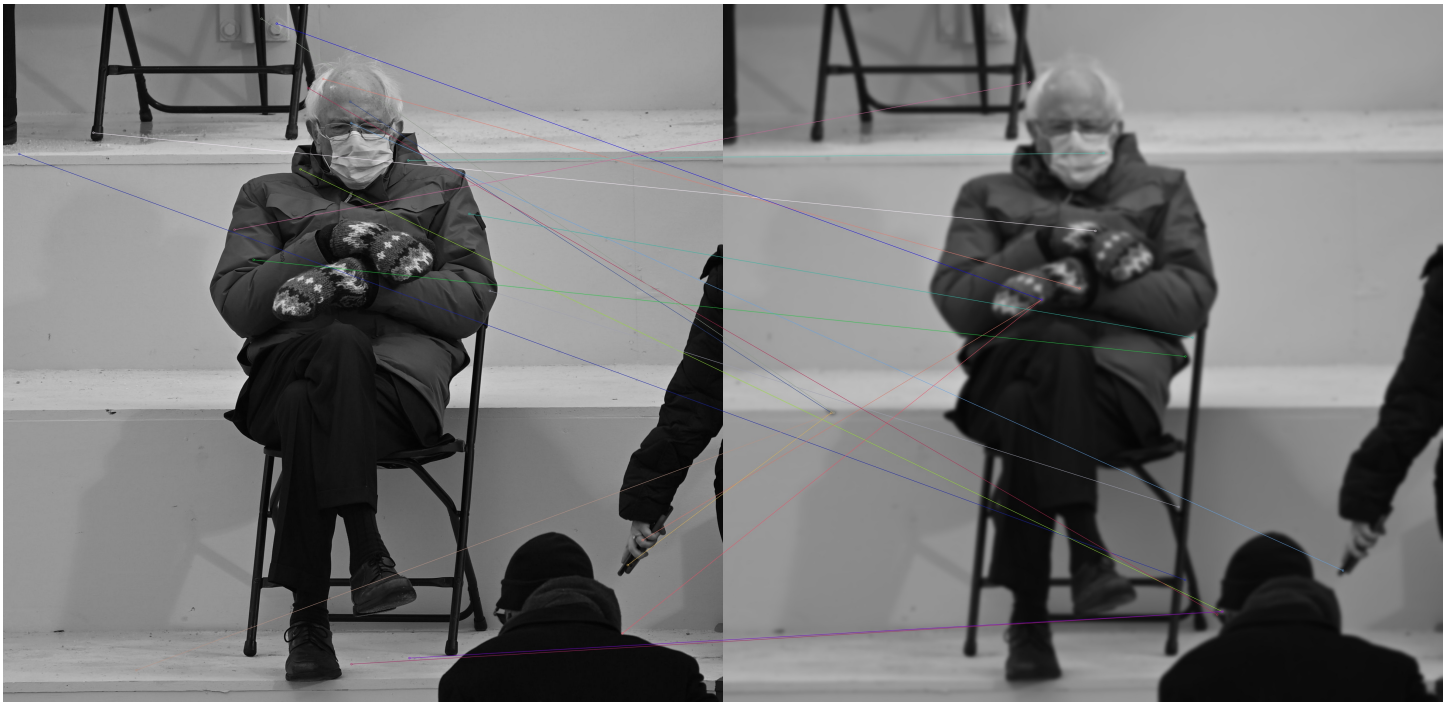
self-implemented Harris

built-in FAST

built-in Harris

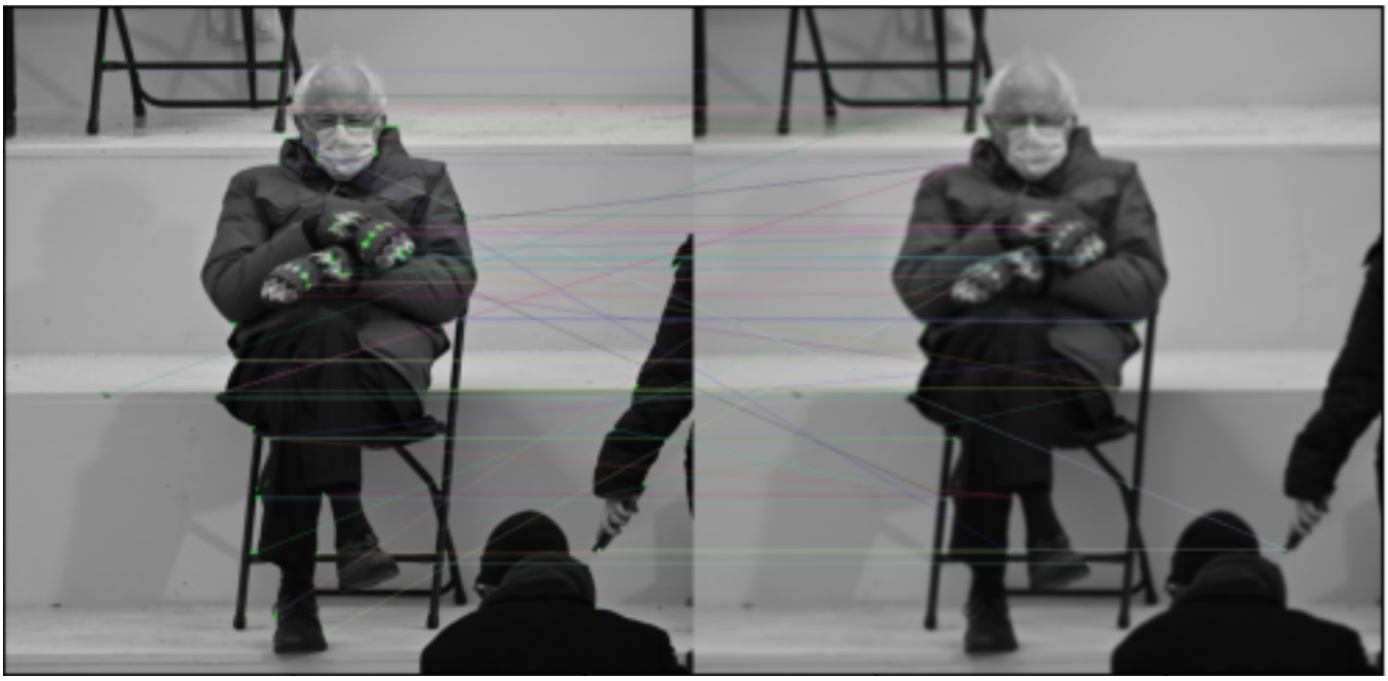
Blur

The blurred image caused serious difficulties in matching, and except for the white mask on the face, which was correctly matched, none of the other key points were found to be correct.



self-implemented Harris

built-in FAST



built-in Harris

Other combined image comparisons

In this set of experiments, most key points can be matched correctly, but in complex images (such as a group photo of multiple people with a dark background color)

Conclusion

In the realm of interest point detection, both Harris and FAST detectors offer distinct advantages and disadvantages across different image scenarios. The Harris interest points detector exhibits robustness in scenarios where images contain prominent gradients and subtle intensity variations, making it particularly effective in scenes with moderate to high contrast. Its ability to leverage gradient information and consider the spatial distribution of gradients enables accurate corner detection even in complex environments. On the other hand, FAST detector performs well in scenarios with minimal noise and relatively uniform intensity distributions. However, it may struggle in scenes with significant noise levels or high brightness, where subtle intensity variations indicative of corners are not distinctly discernible.

Appendix1: Parameters

gaussian mask = 5x5
sobel kernel = 3x3
alpha = 0.05
sigma = 0.5
threshold = 18000
local maxima = 7x7
gaussian size = 5
feature detector limit = 50

Appendix2: Code

HarrisPointsDetector

```python
# 1. Feature detection
def HarrisPointsDetector(mat, gaussian_size=GAUSSIAN_SIZE, threshold=THRESHOLD, alpha=ALPHA, sigma=SIGMA):
    """
    Detect the Harris corner points in the given image
    """
    # Apply Gaussian filter to the image
    mat = cv.GaussianBlur(mat, (0, 0), 3)
    # Calculate the derivatives of x and y
    Ix = cv.Sobel(mat, cv.CV_64F, 1, 0, ksize=3, borderType=cv.BORDER_REFLECT)
    Iy = cv.Sobel(mat, cv.CV_64F, 0, 1, ksize=3, borderType=cv.BORDER_REFLECT)

    # Calculate the products of derivatives
    Ixx = gaussian_filter(np.square(Ix), sigma=sigma,
                          mode='reflect', radius=gaussian_size//2)
    Iyy = gaussian_filter(np.square(Iy), sigma=sigma,
                          mode='reflect', radius=gaussian_size//2)
    Ixy = gaussian_filter(np.multiply(Ix, Iy), sigma=sigma,
                          mode='reflect', radius=gaussian_size//2)

    # orientation of the gradient, in degrees
    orientation = np.arctan2(Iy, Ix) * 180 / np.pi
    detM = (Ixx * Iyy) - (Ixy ** 2)
    traceM = Ixx + Iyy
    R = detM - alpha * (traceM ** 2)  # corner strength function, R

    # Find local maxima in the corner strength matrix
    localMaxima = (R == maximum_filter(R, size=7, mode='reflect'))
    localMaxima = localMaxima * (R > threshold)
    interest_points = np.argwhere(localMaxima)

    keypoints = []
    for (y, x) in interest_points:
        keypoints.append(cv.KeyPoint(x.astype(np.float32), y.astype(
            np.float32), 1, angle=orientation[y, x], response=R[y, x]))
    return keypoints
```

featureDescriptor

```python
# 2. Feature description
def featureDescriptor(image, keypoints, descriptor='orb'):
    """
    Create a descriptor for the given image and keypoints
    """
    if descriptor == 'orb': # ORB with self-designed Harris score
        orb = cv.ORB_create()
        kp, des = orb.compute(image, keypoints)
        return kp, des
    elif descriptor == 'orb_fast':  # ORB with FAST score
        orb_fast = cv.ORB_create(scoreType=cv.ORB_FAST_SCORE)
        kp_fast, des_fast = orb_fast.detectAndCompute(image, None)
        return kp_fast, des_fast
    elif descriptor == 'orb_harris':    # ORB with HARRIS score
        orb_harris_default = cv.ORB_create(scoreType=cv.ORB_HARRIS_SCORE)
        kp_harris, des_harris = orb_harris_default.detectAndCompute(
            image, None)
        return kp_harris, des_harris
```

SSDFeatureMatcher

```python
# 3. Feature matching
# Sum of squared difference
def SSDFeatureMatcher(des1, des2, limit=LIMIT):
    """
    Perform the SSD feature matching
    Returns the best matches(a list of DMatch objects)
    """
    best_matches = []
    # squared Euclidean distance
    distances = cdist(des1, des2, metric='sqeuclidean')
    for queryIdx, distance in enumerate(distances):
        trainIdx = np.argmin(distance)
        best_matches.append(cv.DMatch(queryIdx, trainIdx, distance[trainIdx]))
    best_matches = sorted(best_matches, key=lambda x: x.distance)
    if limit >= len(best_matches):
        return best_matches
    return best_matches[:limit]
```

RatioFeatureMatcher

```python
# Ratio test
def RatioFeatureMatcher(des1, des2, limit=LIMIT, ratio=RATIO):
    """
    Perform the ratio test for feature matching
    """
    best_matches = []
    distances = cdist(des1, des2, metric='sqeuclidean')
    for queryIdx, distance in enumerate(distances):
        trainIdx = np.argmin(distance)
        distance1 = distance[trainIdx]
        distance[trainIdx] = np.inf
        # Find the second closest match to the query descriptor
        second_idx = np.argmin(distance)
        distance2 = distance[second_idx]

        if distance1 / distance2 < ratio:
            best_matches.append(cv.DMatch(queryIdx, trainIdx, distance1))
    best_matches = sorted(best_matches, key=lambda x: x.distance)
    if limit >= len(best_matches):
        return best_matches

    return best_matches[:limit]
```