Introduction

The assignment aims to find and match a certain object shown in a main picture across different test pictures. From a number of test images with different variations such as rotation, blurring, scale, the object we're looking for is Bernie Sanders. Using these images, we test how will the object detector performances. First, we implemented the **Harris Points Detector** for feature detection, next we find the feature description using the OpenCV built-in ORB. Lastly, using our implementation of **SSDFeatureMatcher** and **RatioFeatureMatcher**. These help us figure out which features are a good match. SSDFeatureMatcher looks at the distance between features, while RatioFeatureMatcher looks at the ratio of distances between features. We'll compare the results we get and share our thoughts on how well the process works.

Feature Detection

- Gray-scale image
- Blur the grey-scale image with a 5x5 Gaussian Kernel with a sigma value of 1o
- Compute the image derivatives using the Sobel operators, using the image derivatives find the combinations
- Apply the Gaussian Blur on the combinations of the image derivative
- Compute the Harris Matrix, M, and using M, compute the corner strength function, $R = c(M) = det(M) - 0.05trace(M)^2$ for each pixel
- Filter the corners for non-maxima suppression with thresholding

We created a graph showing the relation between the number of keypoints and threshold to study and find which threshold to choose. Since we desire to minimize the number of features detected outside of Bernie's object, we have chosen to use the threshold value of 0.05.

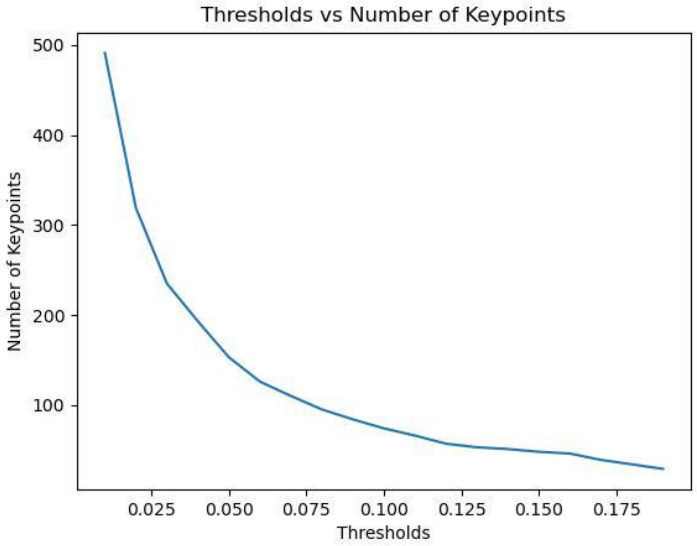

Figure 1: relation between threshold and number of keypoints

```python
"""
def HarrisPointsDetector(
        image, sigma=0.5, alpha=0.05, threshold_value= 0.05, nms=7, graph=False
):
    # Blur gray image
    blur_image = ndimage.gaussian_filter(image, sigma=10, mode="reflect", radius=2)

    # Normalize the image to np.float64
    image_64 = blur_image.astype(np.float64)
    image_64 = blur_image / 255.0

    # Compute image gradients in X and Y directions; axis = 0 for y derivative, axis = 1 for x derivative
    x_gradient_64 = ndimage.sobel(image_64, axis=1, mode="reflect")
    y_gradient_64 = ndimage.sobel(image_64, axis=0, mode="reflect")

    # Compute the combinations between the gradients
    orientations = np.rad2deg(np.arctan2(y_gradient_64, x_gradient_64))

    # Compute the cominations between the gradients for M Matrix: (I_x)^2, (I_y)^2, (I_x)(I_y)
    x2_gradient_64 = np.square(x_gradient_64)
    y2_gradient_64 = np.square(y_gradient_64)
    xy_gradient_64 = np.multiply(x_gradient_64, y_gradient_64)

    # Blur the combinations between the gradients
    blur_x2_gradient_64 = ndimage.gaussian_filter(x2_gradient_64, sigma=sigma, mode="reflect", radius=2)
    blur_y2_gradient_64 = ndimage.gaussian_filter(y2_gradient_64, sigma=sigma, mode="reflect", radius=2)
    blur_xy_gradient_64 = ndimage.gaussian_filter(xy_gradient_64, sigma=sigma, mode="reflect", radius=2)

    # Compute the 2x2 M Matrix Determinant and Trace
    determinant = np.subtract( np.multiply(blur_x2_gradient_64, blur_y2_gradient_64), np.square(blur_xy_gradient_64))
    trace = np.add(x2_gradient_64, y2_gradient_64)

    # Compute the corner strength function, R
    r = determinant - (alpha * np.square(trace))

    maxes = ndimage.maximum_filter(r, size=nms, cval=1e100)
    r_maxes = maxes == r
    rows, cols = image_64.shape

    # Loop through all feature points in r_maxes, where each poiont is filled with content required for descriptor computation
    # requires x, y, and angle
    features = []
    for y in range(rows):
        for x in range(cols):
            if not r_maxes[y, x]:
                continue

            f = cv2.KeyPoint(x, y, 7, orientations[y][x], r[y][x])
            features.append(f)

    """ Graph production when graph flag is true
    graph showing relation between number of keypoints and threshold value
    """
    if graph:
        thresholds = [0.01 * i for i in range(1, 20)]
        no_kps = [0] * len(thresholds)

        for i in range(len(thresholds)):
            threshold = np.max(r) * thresholds[i]
            if features is not None:
                keypoints = [ keypoint for keypoint in features if keypoint.response >= threshold
                ]
                no_kps[i] = len(keypoints)

        plt.plot(thresholds, no_kps)
        plt.xlabel("Thresholds")
        plt.ylabel("Number of Keypoints")
        plt.title("Thresholds vs Number of Keypoints")
        plt.savefig("./threshold_vs_keypoints.jpg")

    # Find the threshold
    # If the response value of the keypoint is greater than threshold, return the keypoint, else don't
    threshold = np.max(r) * threshold_value
    if features is not None:
        keypoints = [ keypoint for keypoint in features if keypoint.response >= threshold
        ]

    return keypoints
```

Figure 2: code for HarrisPointsDetector

Feature Description

To find the feature description, we use the built-in OpenCV ORB. Comparing Figure 3 and 4, we can see that the detection performance between our own implementation (left image) and the ORB detector (right image) is very similar.

Figure 3: Our implementation

Figure 4: ORB Detection

Feature Matching

We implemented two functions **SSDFeatureMatcher** and **RatioFeatureMatcher** to match features from the reference image with features from the sample images. **SSDFeatureMatcher** uses the idea of the squared Euclidean distance between the two feature vectors, and **RatioFeatureMatcher** uses the SSD distance to find the closest and second closest features in ratio.

To avoid ambiguous results when selecting smallest distance for every combination of features, the second smallest distance is also picked then ratio between the first and second is made. The ratio will then be the test ratio distance. We have used 0.75 as the threshold value for the ratio test. If it exceeds the threshold, it is removed.

<u>Rotated Image</u>

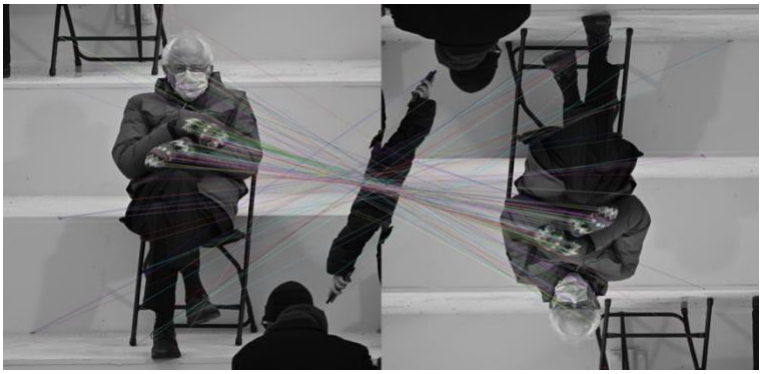

Figure 5: Detection from rotated image using SSDFeatureMatcher

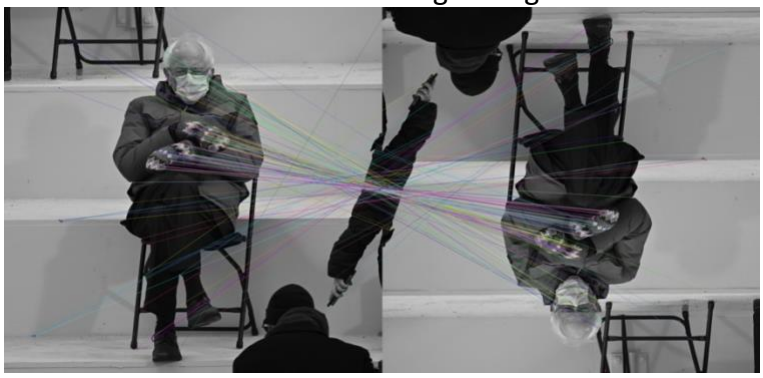

Figure 6: Detection from rotated image using RatioFeatureMatcher

Changes in image brightness

Changes in brightness make it more difficult to accurately detect and find distinctive points in the image especially the edges and corners, where they may become less pronounced. This can result in false positives or false negatives in feature detection, affecting the accuracy.

Figure 7: Detection from brighter image using SSDFeatureMatcher

Figure 8: Detection from brighter image using RatioFeatureMatcher

Figure 9: Detection from darker image using SSDFeatureMatcher

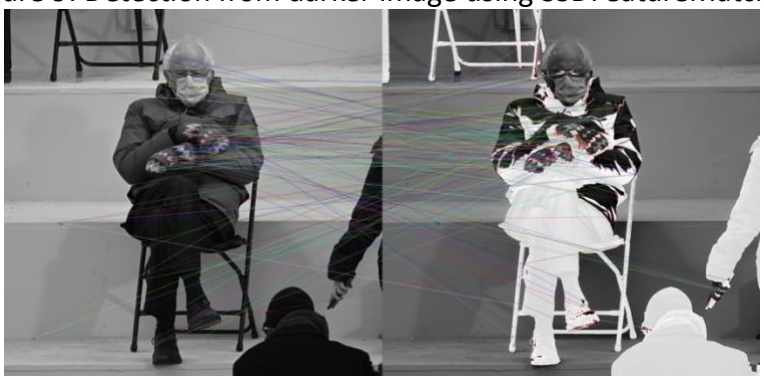

Figure 10: Detection from darker image using RatioFeatureMatcher

Noisy & pixelated images

As the image is noisy, the pixels are misinterpreted. Some matches can be seen but the noise has distorted distinctive points the detector uses to detect, influencing the accuracy of the feature detector. ORB local features use binary feature descriptors where it has high sensitive to intensity changes.

Pixelated images have less impact than noisy images, therefore more matches are correctly found. Due to low resolution and limited colour palette, the edges and corners are not well-defined. ORB Harris implementation meets the expectation in terms of performance, while our implementation has more improvements to be done.

Figure 11: Detection from noisy image using SSDFeatureMatcher

Figure 12: Detection from noisy image using RatioFeatureMatcher

Figure 13: Detection from pixelated image using SSDFeatureMatcher

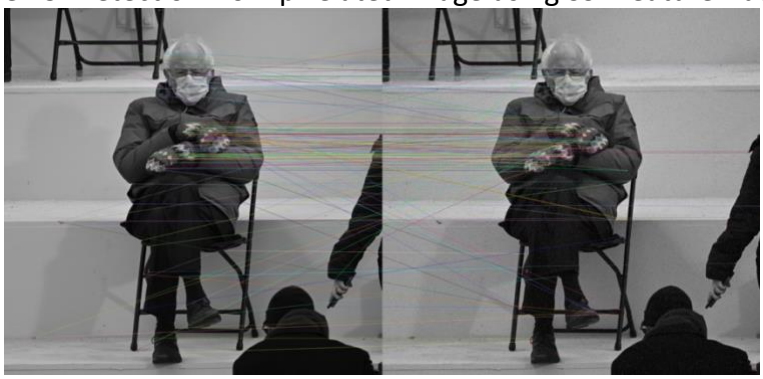

Figure 14: Detection from pixelated image using RatioFeatureMatcher

More blurred images

The image is originally blurred, with more blurring, it reduces the high-frequency information in the image and thus the fine details are lost. The detector managed to take the keypoints from the image, and a few correctly matched.

Figure 15: Detection from more blurred image using SSDFeatureMatcher

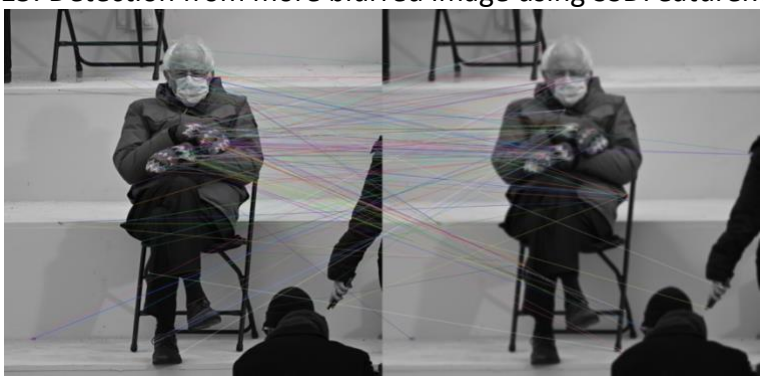

Figure 16: Detection from more blurred image using RatioiFeatureMatcher

Bernie in different locations (RatioFeatureMatcher)

Due to change in scale, only a few matches are correct, while others are misinterpreted. Scaling affect the performance of feature detection and matching

Figure 17: Detection from salon image using RatioiFeatureMatcher

Figure 18: Detection from with friendsimage using RatioiFeatureMatcher

Figure 19: Detection from school lunch image using RatioiFeatureMatcher

```python
"""
Calculate squared Euclidean distance between the two feature vectors
and matches a feature in the first inage with the closest features in the second image

NOTE: There is possibility of having the same multiple features from the first image with the second image
"""
def SSDFeatureMatcher(ref_desc, sample_desc):
    distances = spatial.distance.cdist(ref_desc, sample_desc, 'sqeuclidean')
    matches = []
    for i, dist in enumerate(distances):
        idx1 = i
        idx2 = np.argmin(dist)
        distance = dist[idx2]
        matches.append(cv2.DMatch(idx1, idx2, 0, distance))
    return matches

"""
Uses a ratior of SSD distance of the two best matches and
matches a feature in the first image with the cloeset feature in the second image.

NOTE: There is possibility of having the same multiple features from the first image with the second image
"""
def RatioFeatureMatcher(ref_desc, sample_desc, ratio=0.75):

    distances = spatial.distance.cdist(ref_desc, sample_desc, 'sqeuclidean')
    matches = []
    for i, dist in enumerate(distances):
        idx1 = i
        sorted_indices = np.argsort(dist)
        best_match_dist = dist[sorted_indices[0]]
        second_best_match_dist = dist[sorted_indices[1]]
        if best_match_dist < ratio * second_best_match_dist:
            idx2 = sorted_indices[0]
            matches.append(cv2.DMatch(idx1, idx2, 0, best_match_dist))
    return matches

```

Figure 20: code for SSDFeatureMatcher and RatioFeatureMatcher functions

Conclusion

Our implementation struggles when dealing with different variations, as the detected features don't always match correctly. The level of blurring, determined by the sigma value in Gaussian blurring, affects how well-defined the features appear. A higher sigma value means more blurring, aggregating more information from distant samples when the kernel is applied through convolution. I chose a sigma value of 10 for blurring in our implementation.

Setting a threshold for Sum of Squared Differences (SSD) can lead to many incorrect matches, even with a high ratio threshold, particularly when dealing with benchmarks at different scales. Empirical observations show that most images don't produce low ratios between the first and second matches, even when they are correct.

Instead of selecting the threshold algorithmically by picking the top 10% and choosing the lowest value, we opted for an empirical approach. However, finding optimal parameters for our model's performance requires a significant amount of time due to our current level of experience.

One reason our implementation lacks scale invariance is due to using a 7x7 maximum filter for non-maximum suppression. The reference image is relatively large compared to smaller benchmark images, causing the information extracted by the 7x7 grid in the reference image to be relatively smaller than in the smaller images.

Changes in image intensities in benchmark images can pose challenges for accurate detection and description of distinctive points using the Harris interest points with ORB local features. This can result in inaccuracies in feature detection and matching. When images are noisy, the detector may identify false corners or miss real ones, leading to inaccuracies in feature detection and matching. Blurring the image before passing it to the HarrisPointsDetector function helps remove some high-frequency noise, making it easier for the detector to identify true corners.

Appendix

```python
"""
def HarrisPointsDetector(
        image, sigma=0.5, alpha=0.05, threshold_value= 0.05, nms=7, graph=False
):
    # Blur gray image
    blur_image = ndimage.gaussian_filter(image, sigma=10, mode="reflect", radius=2)

    # Normalize the image to np.float64
    image_64 = blur_image.astype(np.float64)
    image_64 = blur_image / 255.0

    # Compute image gradients in X and Y directions; axis = 0 for y derivative, axis = 1 for x derivative
    x_gradient_64 = ndimage.sobel(image_64, axis=1, mode="reflect")
    y_gradient_64 = ndimage.sobel(image_64, axis=0, mode="reflect")

    # Compute the combinations between the gradients
    orientations = np.rad2deg(np.arctan2(y_gradient_64, x_gradient_64))

    # Compute the cominations between the gradients for M Matrix: (I_x)^2, (I_y)^2, (I_x)(I_y)
    x2_gradient_64 = np.square(x_gradient_64)
    y2_gradient_64 = np.square(y_gradient_64)
    xy_gradient_64 = np.multiply(x_gradient_64, y_gradient_64)

    # Blur the combinations between the gradients
    blur_x2_gradient_64 = ndimage.gaussian_filter(x2_gradient_64, sigma=sigma, mode="reflect", radius=2)
    blur_y2_gradient_64 = ndimage.gaussian_filter(y2_gradient_64, sigma=sigma, mode="reflect", radius=2)
    blur_xy_gradient_64 = ndimage.gaussian_filter(xy_gradient_64, sigma=sigma, mode="reflect", radius=2)

    # Compute the 2x2 M Matrix Determinant and Trace
    determinant = np.subtract( np.multiply(blur_x2_gradient_64, blur_y2_gradient_64), np.square(blur_xy_gradient_64))
    trace = np.add(x2_gradient_64, y2_gradient_64)

    # Compute the corner strength function, R
    r = determinant - (alpha * np.square(trace))

    maxes = ndimage.maximum_filter(r, size=nms, cval=1e100)
    r_maxes = maxes == r
    rows, cols = image_64.shape

    # Loop through all feature points in r_maxes, where each poiont is filled with content required for descriptor computation
    # requires x, y, and angle
    features = []
    for y in range(rows):
        for x in range(cols):
            if not r_maxes[y, x]:
                continue

            f = cv2.KeyPoint(x, y, 7, orientations[y][x], r[y][x])
            features.append(f)

    """ Graph production when graph flag is true
    graph showing relation between number of keypoints and threshold value
    """
    if graph:
        thresholds = [0.01 * i for i in range(1, 20)]
        no_kps = [0] * len(thresholds)

        for i in range(len(thresholds)):
            threshold = np.max(r) * thresholds[i]
            if features is not None:
                keypoints = [ keypoint for keypoint in features if keypoint.response >= threshold
                ]
                no_kps[i] = len(keypoints)

        plt.plot(thresholds, no_kps)
        plt.xlabel("Thresholds")
        plt.ylabel("Number of Keypoints")
        plt.title("Thresholds vs Number of Keypoints")
        plt.savefig("./threshold_vs_keypoints.jpg")

    # Find the threshold
    # If the response value of the keypoint is greater than threshold, return the keypoint, else don't
    threshold = np.max(r) * threshold_value
    if features is not None:
        keypoints = [ keypoint for keypoint in features if keypoint.response >= threshold
        ]

    return keypoints
```

```python
"""
Calculate squared Euclidean distance between the two feature vectors
and matches a feature in the first inage with the closest features in the second image

NOTE: There is possibility of having the same multiple features from the first image with the second image
"""
def SSDFeatureMatcher(ref_desc, sample_desc):
    distances = spatial.distance.cdist(ref_desc, sample_desc, 'sqeuclidean')
    matches = []
    for i, dist in enumerate(distances):
        idx1 = i
        idx2 = np.argmin(dist)
        distance = dist[idx2]
        matches.append(cv2.DMatch(idx1, idx2, 0, distance))
    return matches

"""
Uses a ratior of SSD distance of the two best matches and
matches a feature in the first image with the cloeset feature in the second image.

NOTE: There is possibility of having the same multiple features from the first image with the second image
"""
def RatioFeatureMatcher(ref_desc, sample_desc, ratio=0.75):

    distances = spatial.distance.cdist(ref_desc, sample_desc, 'sqeuclidean')
    matches = []
    for i, dist in enumerate(distances):
        idx1 = i
        sorted_indices = np.argsort(dist)
        best_match_dist = dist[sorted_indices[0]]
        second_best_match_dist = dist[sorted_indices[1]]
        if best_match_dist < ratio * second_best_match_dist:
            idx2 = sorted_indices[0]
            matches.append(cv2.DMatch(idx1, idx2, 0, best_match_dist))
    return matches

```

```python
    def featureDescriptor(image, keypoints):
        _, descriptor = orb.compute(image, keypoints)

        return descriptor
```