<u>Coursework 2 Report – Usman Ali</u>

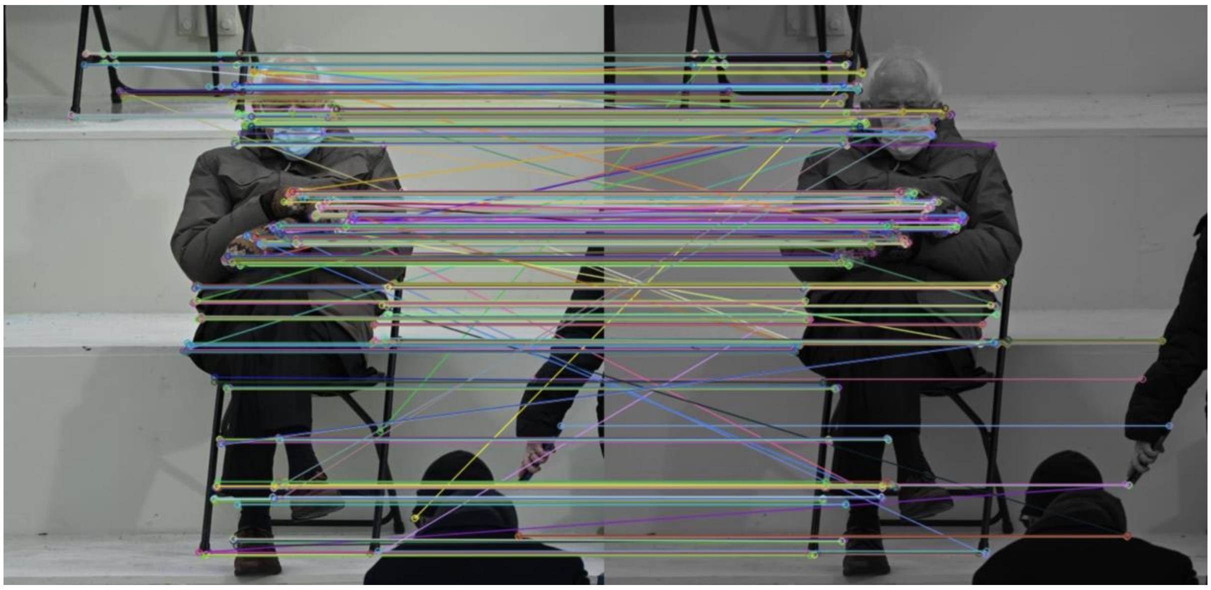

Figure 1

Figure 1 shows the SSD matches drawn between the default Bernie image and the darker Bernie image. The SSD Matching function:

- Computes the sum of squared differences (SSD) between descriptors from two images.
- Finds the best match for each descriptor in the first image by identifying the descriptor in the second image with the smallest SSD.
- The function matches each descriptor in the first image with exactly one descriptor in the second image, even if that match is not a good one, potentially including noise.

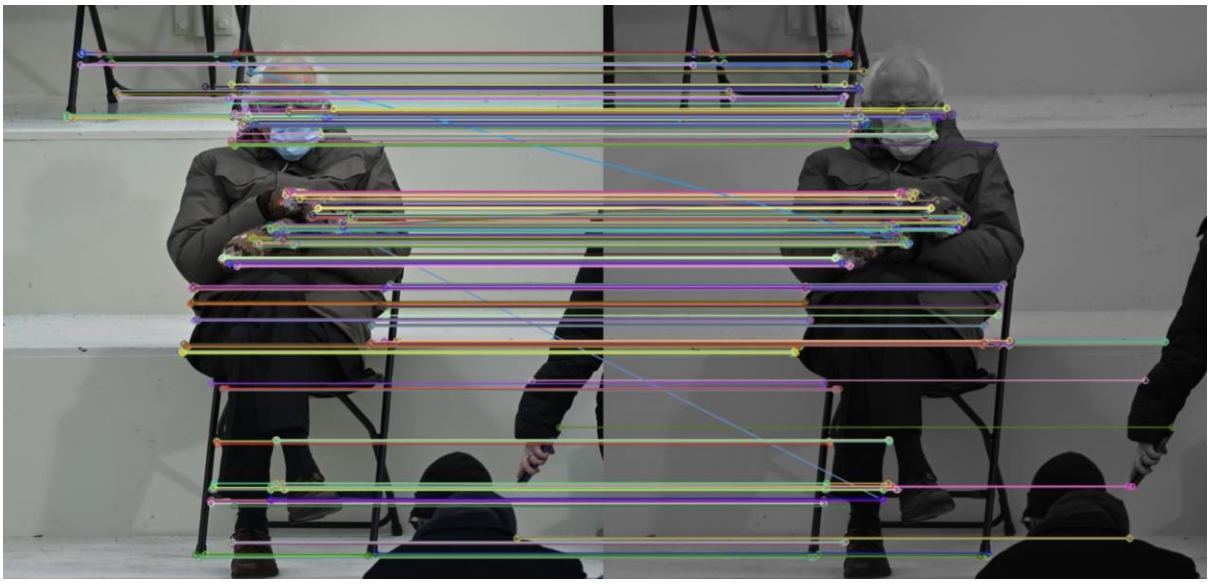

Figure 2

Figure 2 shows the Ratio matches between the default image and the darker Bernie image.

The Ratio function:

- Also computes the SSD between all descriptors.
- Performs a sort to rank the SSD values for each descriptor in the first image against all descriptors in the second image.
- Implements the ratio test by comparing the distance of the best match to the distance of the second-best match.
- Only retains a match if the best match distance is less than ratio times the second-best match distance, which helps to filter out poor matches.

The difference between the two images is due to the ratio function removing bad matches that don't meet the ratio threshold. The second image consists of better matches as the ratio test is used to improve the matching by ensuring that the best match is significantly closer than the second-best match, thereby reducing the likelihood of false positives. Changing the ratio also affects the quality of matches that are drawn, a lower ratio results in the acceptance of stronger matches and the disqualification of weaker ones. The ratio used in figure 2 was 0.7. In comparison to figure 3 below, this image uses a ratio of 0.4, demonstrating the elimination of a lot of the bad matches.

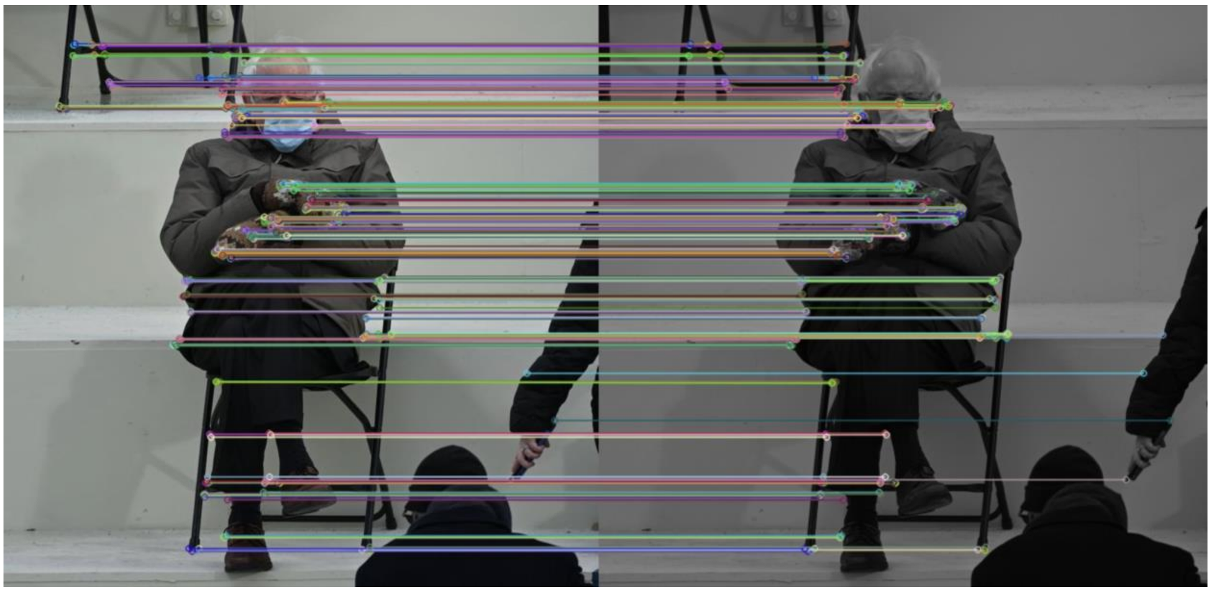

Figure 3

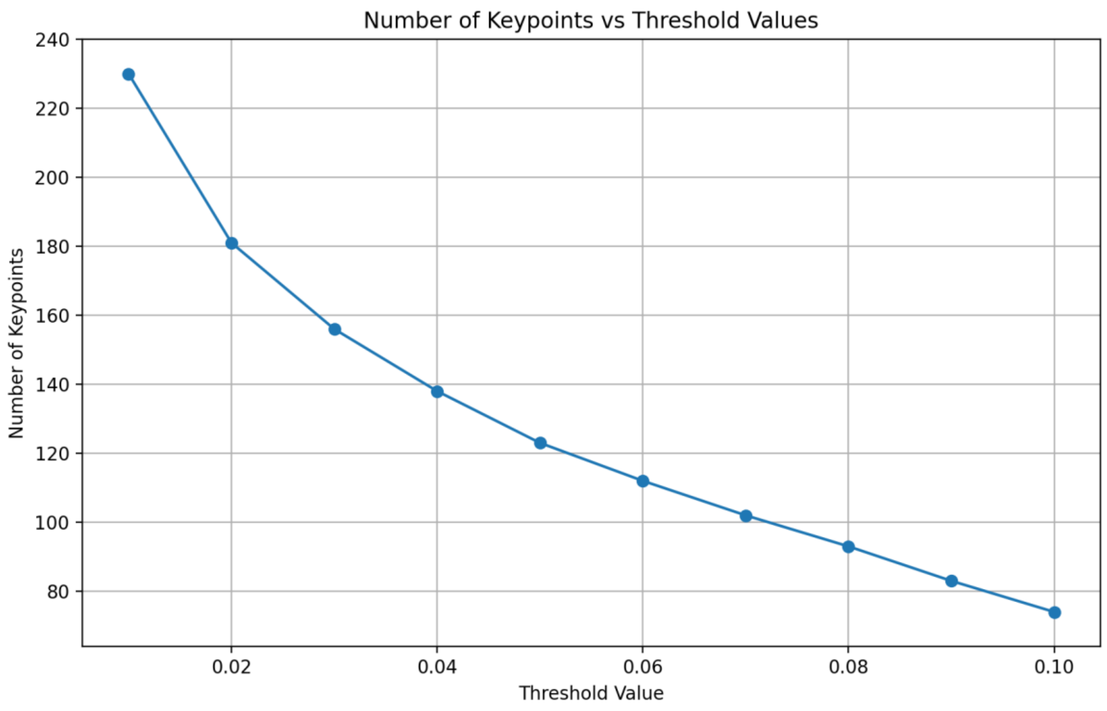

Figure 4

The graph displays a negative trend, showing that fewer keypoints are discovered as the threshold value rises. When the threshold is low, even points with a small corner response are considered keypoints; this results in a higher number of keypoints being detected. As the threshold increases, only points with a stronger corner response are detected and classed as keypoints resulting in fewer points of interest.

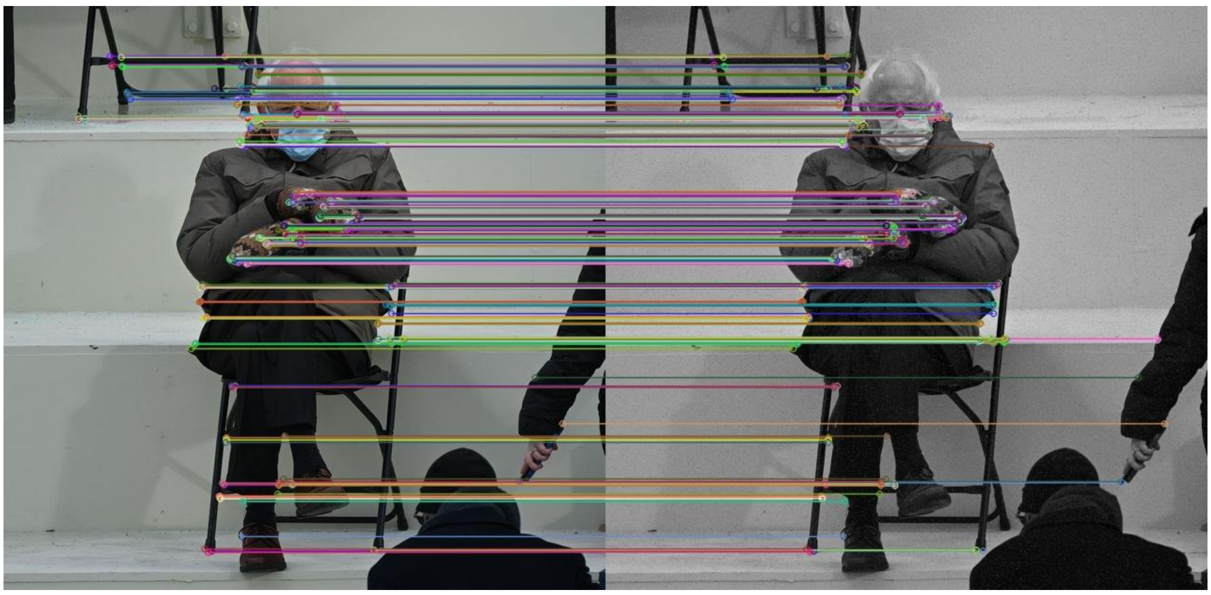

Figure 5

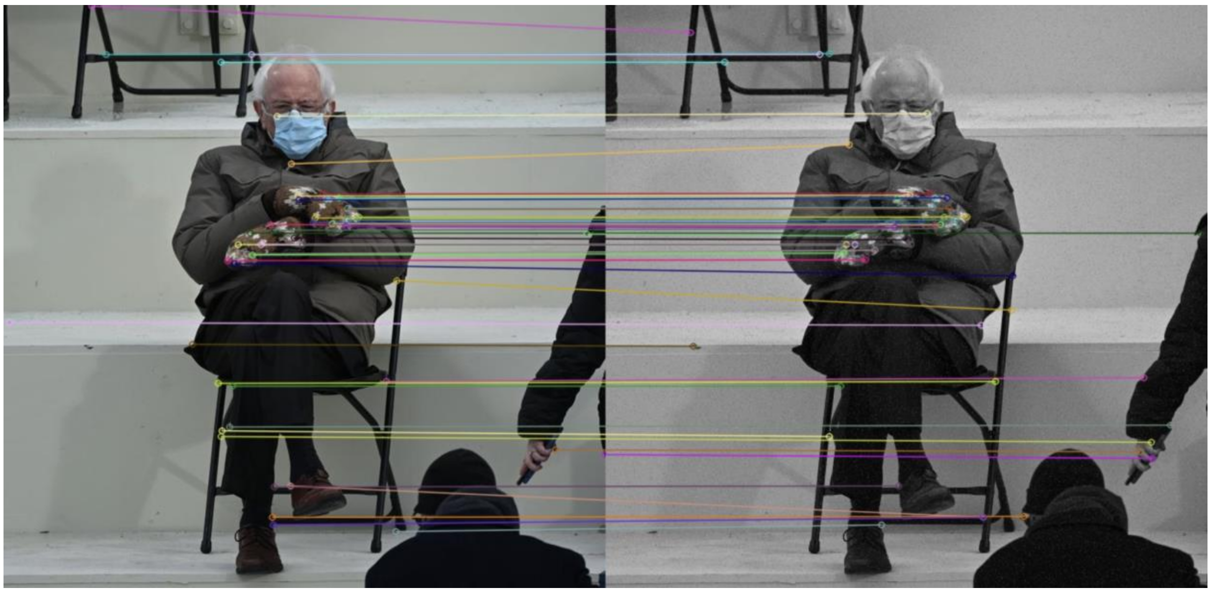

Figure 6

Figure 5 and 6 both show the use of the HarrisPointsDetector and the FAST detector (figure 5) on the berniePixelated image. The Orb FAST detector seems to have produced more accurate matches in comparison to the HarrisPointsDetector which could potentially suggest an issue with this function or the function struggling with identifying matches due to the changes in gradient caused by pixelation.

```
def HarrisPointsDetector(image, blockSize=5, apertureSize=3, k=0.05,
threshold=0.01):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    ddepth = cv2.CV_64F
    Ix = cv2.Sobel(gray, ddepth, 1, 0, ksize=apertureSize,
borderType=cv2.BORDER_REFLECT101)
    Iy = cv2.Sobel(gray, ddepth, 0, 1, ksize=apertureSize,
borderType=cv2.BORDER_REFLECT101)

    # Weighting the gradients with a Gaussian filter
    Ixx = gaussian_filter(Ix**2, sigma=0.5)
    Iyy = gaussian_filter(Iy**2, sigma=0.5)
    Ixy = gaussian_filter(Ix*Iy, sigma=0.5)


    # Compute the Harris matrix M for each pixel
    detM = (Ixx * Iyy) - (Ixy ** 2)
    traceM = Ixx + Iyy
    R = detM - (k * (traceM ** 2))

    # Orientation calculation
    orientation = np.arctan2(Iy, Ix)
    orientation = np.rad2deg(orientation)
    orientation = (orientation + 360) % 360  # Convert to range [0, 360)
    # Non-maxima suppression to get keypoints
    local_max = maximum_filter(R, size=(7,7))
    R_max = (R == local_max)
    keypoints = np.argwhere(R_max)
    keypoints_with_scores = [
        cv2.KeyPoint(float(x[1]), float(x[0]), blockSize,
angle=float(orientation[x[0], x[1]]), response=float(R[x[0], x[1]]))
        for x in keypoints
    ]

    threshold = threshold * R.max()
    keypoints_final = [kp for kp in keypoints_with_scores if kp.response >
threshold]
    return keypoints_final

def featureDescriptor(image, keypoints):
    orb = cv2.ORB_create()
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Compute the descriptors with ORB for the keypoints
    _, descriptors = orb.compute(gray, keypoints)

    return descriptors

def ssd_match(descriptors1, descriptors2):
    distances = cdist(descriptors1, descriptors2, 'sqeuclidean')
    matches = []
    for i in range(len(distances)):
        best_match_idx = np.argmin(distances[i])
        matches.append(cv2.DMatch(_queryIdx=i, _trainIdx=best_match_idx,
_distance=distances[i][best_match_idx]))
    return matches

def ratio_match(descriptors1, descriptors2, ratio=0.7):
    distances = cdist(descriptors1, descriptors2, 'sqeuclidean')
    matches = []
    for i in range(len(distances)):
        ordered_distance = np.argsort(distances[i])
        if len(ordered_distance) > 1:
            if distances[i][ordered_distance[0]] < ratio *
distances[i][ordered_distance[1]]:
                matches.append(cv2.DMatch(_queryIdx=i,
_trainIdx=ordered_distance[0], _distance=distances[i][ordered_distance[0]]))
    return matches
```

```
threshold_values = np.linspace(0.01, 0.1, 10)
keypoint_counts = []
```

```python
for threshold in threshold_values:
    keypoints = HarrisPointsDetector(defaultBernie, threshold=threshold)
    keypoint_count = len(keypoints)
    keypoint_counts.append(keypoint_count)

plt.figure(figsize=(10, 6))
plt.plot(threshold_values, keypoint_counts, marker='o')
plt.title('Number of Keypoints vs Threshold Values')
plt.xlabel('Threshold Value')
plt.ylabel('Number of Keypoints')
# Set the y-axis limits to expected range of keypoint counts
plt.ylim([min(keypoint_counts) - 10, max(keypoint_counts) + 10])
plt.gca().get_yaxis().set_major_formatter(plt.FuncFormatter(lambda x, loc:
"{:,}".format(int(x))))
plt.grid(True)
plt.show()
```