Coursework 2 - Report

Iva Jorgusheska, UID: 11114620

March 28, 2025

1 Harris Corner Detector Function

Corner detection is a fundamental step in image processing and computer vision, especially for identifying keypoints that remain stable across different images of the same scene. Corners are highly distinctive because they exhibit significant changes in intensity along two perpendicular directions, making them robust against translation and rotation.

In this implementation, I developed a Harris Corner Detector in Python, following the standard approach outlined in the lecture slides. The algorithm computes the Harris matrix M at each pixel, applies a Gaussian weighting function, and extracts strong interest points based on the Harris response function R.

1.1 Computation of Image Gradients Using Sobel Operator and Reflection Padding

To compute the partial derivatives of the image with respect to x and y, I used the 3×3 Sobel operator, which approximates the gradient:

$$I_x = \frac{\partial I}{\partial x}, \quad I_y = \frac{\partial I}{\partial y} \tag{1}$$

These gradients are critical because, in a corner, intensity changes in both directions.

The Sobel operator was applied using OpenCV's `cv2.Sobel()` function. Reflection padding (`cv2.BORDER_REFLECT`) was used to prevent artificial border artifacts. The Sobel operator detects edges by computing differences between neighboring pixel intensities, making it ideal for detecting abrupt changes that indicate corners.

```
import cv2

# Compute image gradients using Sobel operator
Ix = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3, borderType=cv2.BORDER_REFLECT)
Iy = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3, borderType=cv2.BORDER_REFLECT)
```

Once the gradients were computed, their squared and cross-product terms were derived to construct the Harris matrix. These terms capture the structure of the local image patches, allowing us to analyze intensity variations in both directions.

```
# Compute squared and cross-product gradient terms
Ixx = Ix ** 2
Iyy = Iy ** 2
Ixy = Ix * Iy
```

1.2 Computation of Harris Response R Using a Gaussian Window

To enhance corner localization and suppress noise, the second-moment matrix terms (S_{xx}, S_{yy}, S_{xy}) were computed using a 5×5 Gaussian window with $\sigma = 0.5$. The Gaussian filter smooths the gradient products, reducing noise and making the detector more stable.

```
# Apply Gaussian blur to compute second-moment matrix terms
sigma = 0.5
gaussian_filter = (5, 5)
Sxx = cv2.GaussianBlur(Ixx, gaussian_filter, sigma)
Syy = cv2.GaussianBlur(Iyy, gaussian_filter, sigma)
Sxy = cv2.GaussianBlur(Ixy, gaussian_filter, sigma)
```

Gaussian smoothing ensures that corner detection is robust to small image variations and noise, making it more reliable.

The Harris response R is computed as:

$$R = \det(M) - k(\text{trace}(M))^2 \tag{2}$$

where $k = 0.05$ is an empirically chosen constant. R determines the likelihood of a pixel being a corner.

```
# Compute determinant and trace of the matrix M
detM = (Sxx * Syy) - (Sxy ** 2)
traceM = Sxx + Syy

# Compute Harris response function
R = detM - k * (traceM ** 2)
```

To facilitate thresholding, R was normalized using OpenCV's `cv2.normalize()` function. Normalization scales R between 0 and 255, ensuring consistency across different images and lighting conditions.

```
# Normalize R for better thresholding
R_norm = cv2.normalize(R, None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX)
threshold = threshold_scale * R_norm.max()
```

1.3 Identifying Strong Interest Points

1. Thresholding: We keep only points where the Harris response R exceeds a percentage of its maximum value, ensuring that only strong corners are retained.

2. Local Maxima Search in a 7×7 Window: This ensures that detected keypoints are spatially distinct, preventing clustering of detections around a single corner.

```
from scipy.ndimage import maximum_filter

# Identify local maxima in the response map
local_max = (R_norm == maximum_filter(R_norm, size=7))
```

3. Keypoint Storage with Orientation Information: Orientation information is crucial for feature matching in further applications.

```
import numpy as np

# Compute gradient magnitude and orientation
magnitude = np.sqrt(Ix**2 + Iy**2)
orientation = np.arctan2(Iy, Ix) * (180 / np.pi)  # Convert radians to degrees

# Extract keypoints based on threshold and local maxima
for y, x in zip(*np.where(local_max & (R_norm > threshold))):
    keypoints.append(cv2.KeyPoint(float(x), float(y), 1, orientation[y, x]))
```

1.4 Effect of Image Preprocessing on Corner Detection

Original Image For the original image, no pre-smoothing was applied before calculating the derivatives for detecting interest points. This approach worked perfectly, as smoothing would only reduce the function's quality. The same pre-procesing is also for the 180-degree rotated image since the Harris detector is invariant to rotation.

Noisy Image For the noisy image, a Gaussian blur with a 3×3 window was applied before computing derivatives. This preprocessing step helped reduce false detections caused by densely distributed noise pixels. A Gaussian sigma value of 3 was used to ensure effective noise suppression.

```
# Apply Gaussian blur to reduce noise
gray = cv2.GaussianBlur(gray, (3, 3), 3)
```

Choosing $\sigma = 3$ ensures that only nearby pixels contribute to smoothing, preventing interference from distant noise points.

Pixelated Image The same preprocessing technique as the noisy image was used for the pixelated image. Applying a Gaussian blur helped to smooth out artificial edges and improve the accuracy of derivative calculations.

Bright Image Corner detection on the bright image performed well without additional smoothing, similar to the original image. While the same threshold scaling could be used, a slightly lower threshold was preferred to improve corner detection accuracy.

Blurred Image A major issue with blurred images arises from strong Gaussian smoothing, which significantly reduces image gradients. This results in a more uniform intensity distribution, leading to weak corner responses. The second-moment matrix, which relies on intensity differences, produces small corner responses, causing excessive detections or failing to identify strong features.

To improve corner detection in blurred images, sharpening was applied before computing the gradients. This enhanced edge details, making corner features more distinguishable.

Steps to Improve Detection in Blurred Images: 1. Apply an unsharp masking filter (sharpening) before computing the gradients. 2. Modify Gaussian smoothing parameters to balance detail preservation and noise reduction. 3. Dynamically adjust the threshold based on the image's contrast.

```
# Apply sharpening to enhance edges
sharpening_kernel = np.array([[-1, -1, -1],
                              [-1,  9, -1],
                              [-1, -1, -1]])
gray = cv2.filter2D(gray, -1, sharpening_kernel)
```

This sharpening step enhances gradients, allowing the detector to identify corners more effectively.

2 Effect of Threshold Value on the Number of Detected Keypoints

Thresholding plays a crucial role in Harris Corner Detection, as it directly impacts the number of keypoints identified. A higher threshold selects only the strongest corners, reducing the total number of detected keypoints, whereas a lower threshold retains weaker corners, potentially increasing noise.

To analyze this effect, I experimented with varying threshold values and plotted the corresponding number of detected keypoints. The threshold values were chosen specifically for each image, as I observed that only within a certain range did significant changes occur. Notably, small changes in the threshold could cause a sharp jump from around 400 detected matches to over 10,000, after which the number stabilized.

My Harris corner detection function employs an adaptive threshold based on the maximum response value R, meaning the threshold operates on a relative scale. Consequently, I input different threshold scale values and also plotted the actual computed threshold values in the graphs. This explains why the numerical values might appear unconventional.

These graphs illustrate how increasing the threshold results in fewer detected keypoints, emphasizing the trade-off between precision and recall in feature detection.

2.1 Visualization of Keypoint Detection at Different Thresholds

Below are visualizations of the number of detected keypoints at various threshold values. Each image demonstrates how increasing the threshold impacts the number of detected corners.

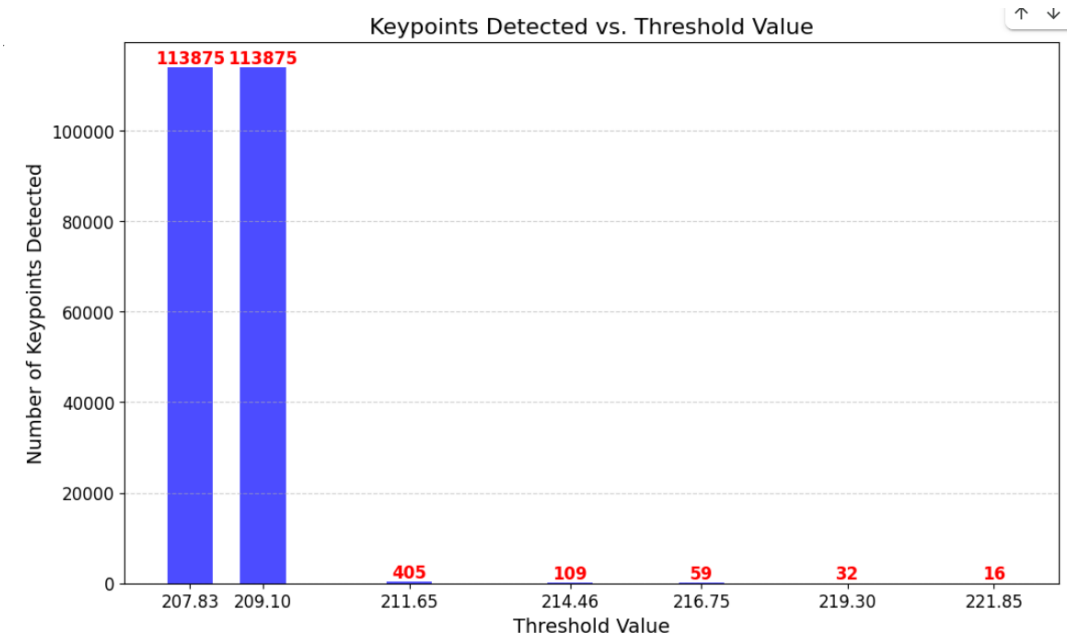

Figure 1: Detected Keypoints in Original Image

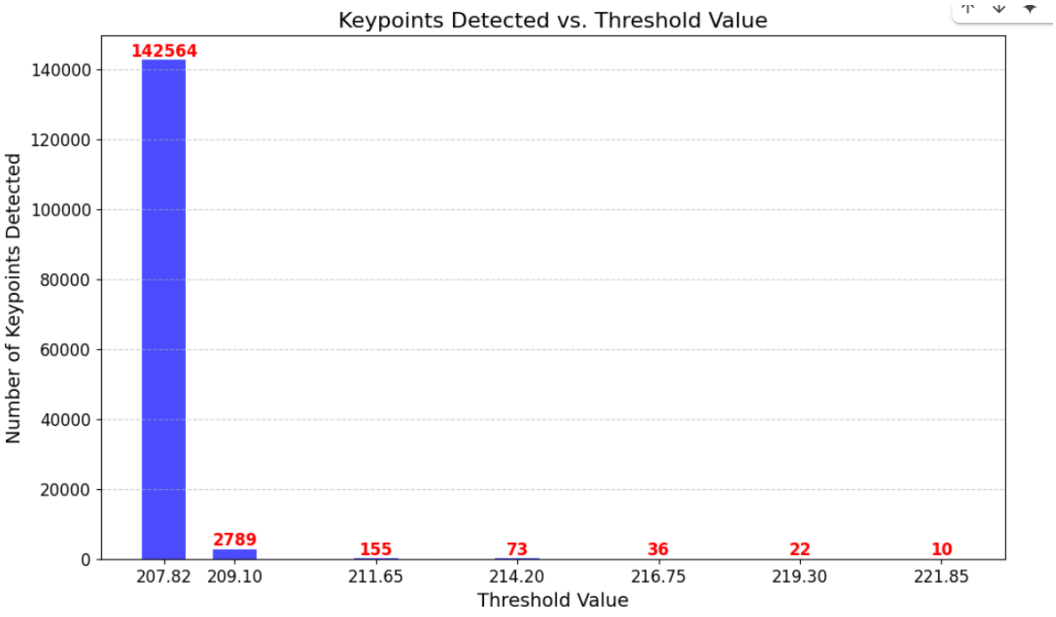

Figure 2: Detected Keypoints in Darker Image

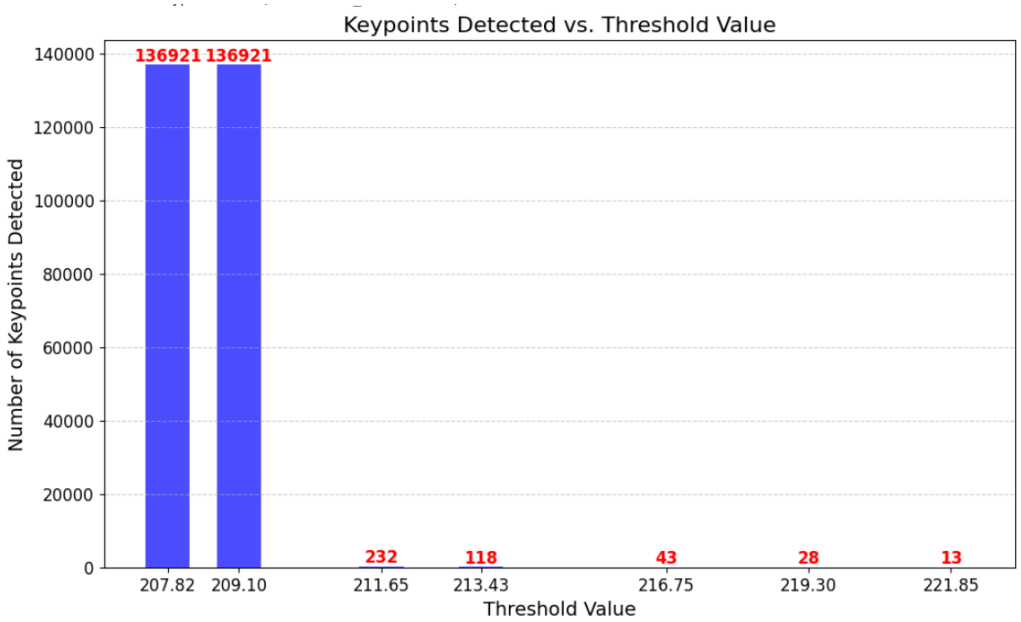

Figure 3: Detected Keypoints in Brighter Image

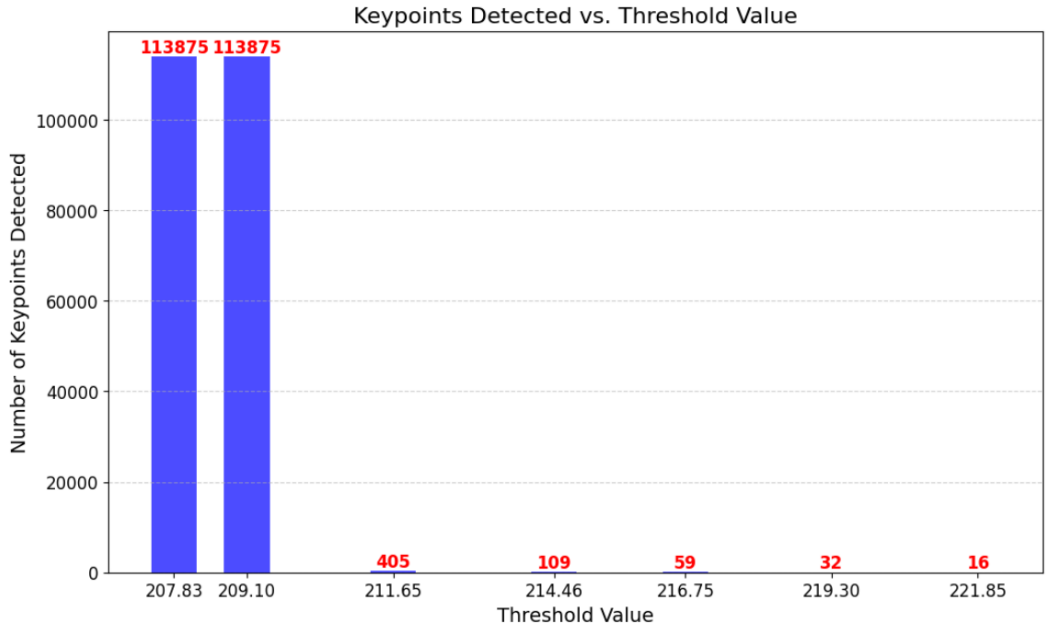

Figure 4: Detected Keypoints in 180-degree Rotated Image

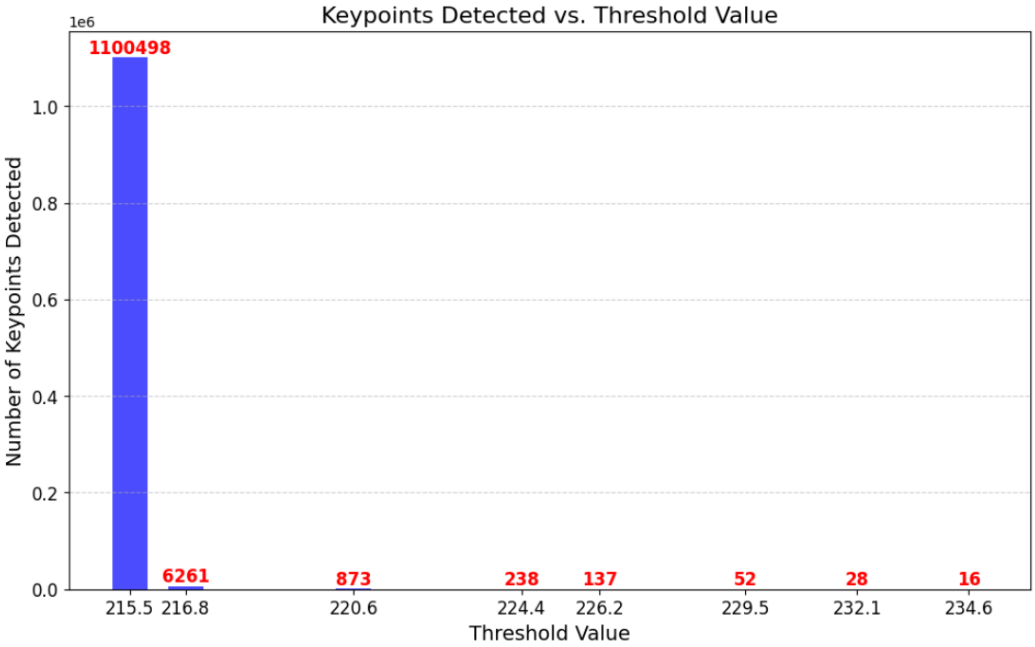

Figure 5: Detected Keypoints in Blurred Image

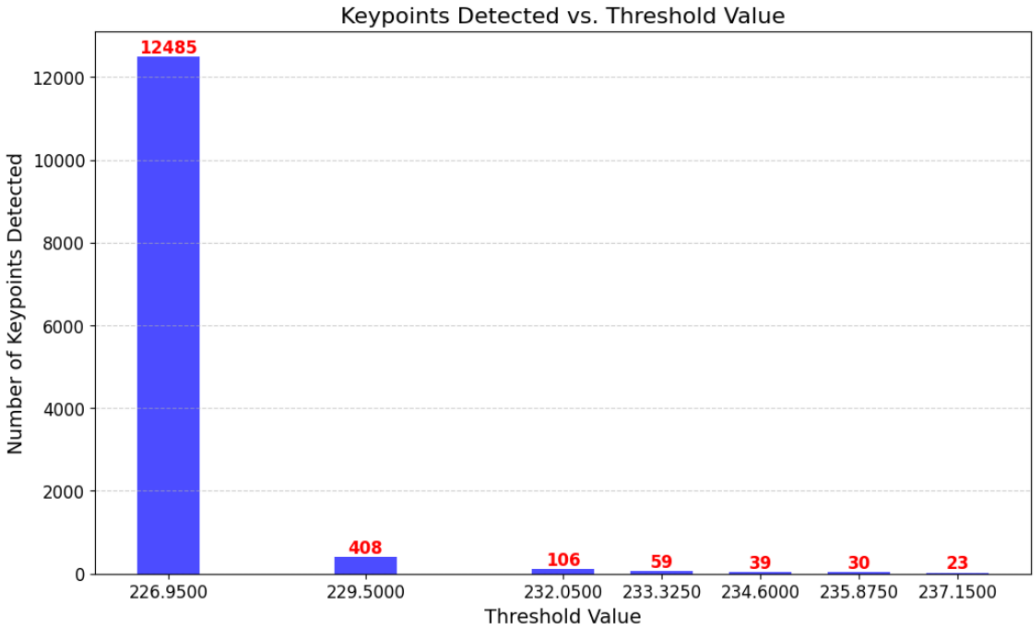

Figure 6: Detected Keypoints in Noisy Image

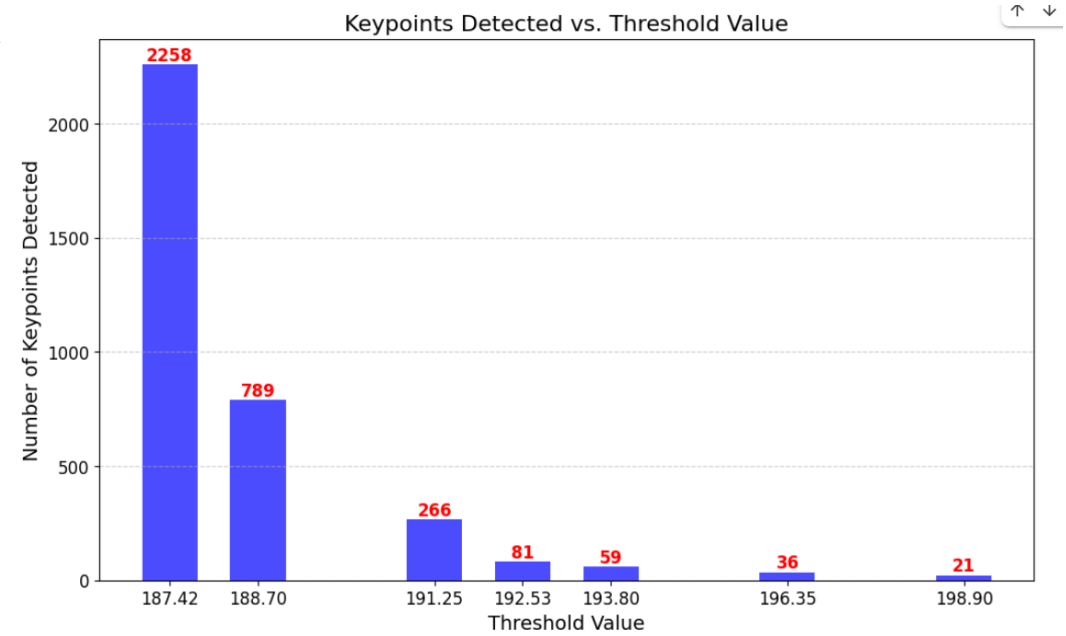

Figure 7: Detected Keypoints in Pixelated Image

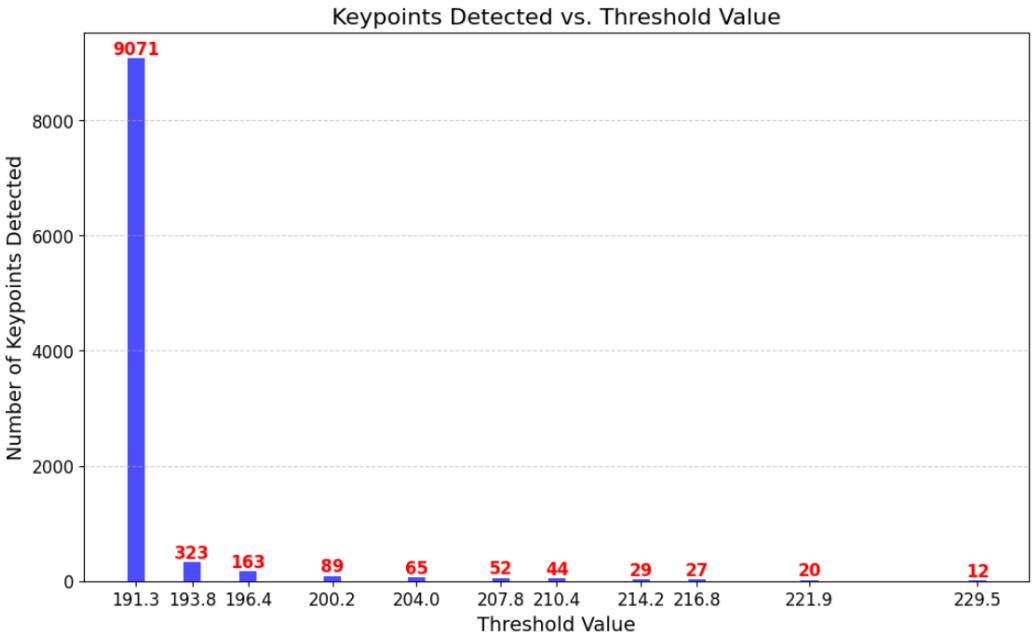

Figure 8: Detected Keypoints in Friends Image

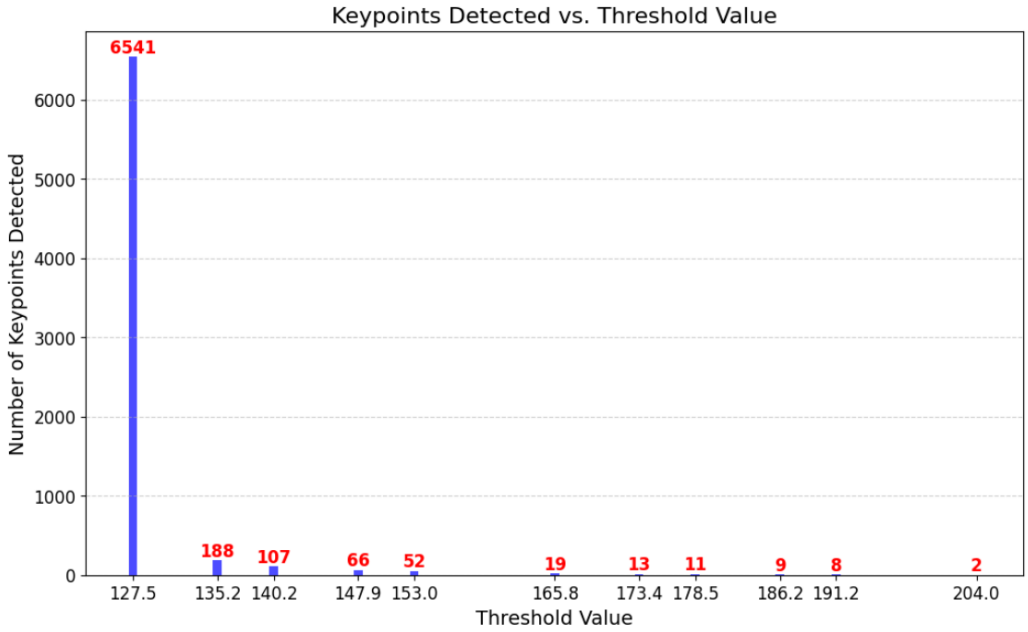

Figure 9: Detected Keypoints in Salon Image

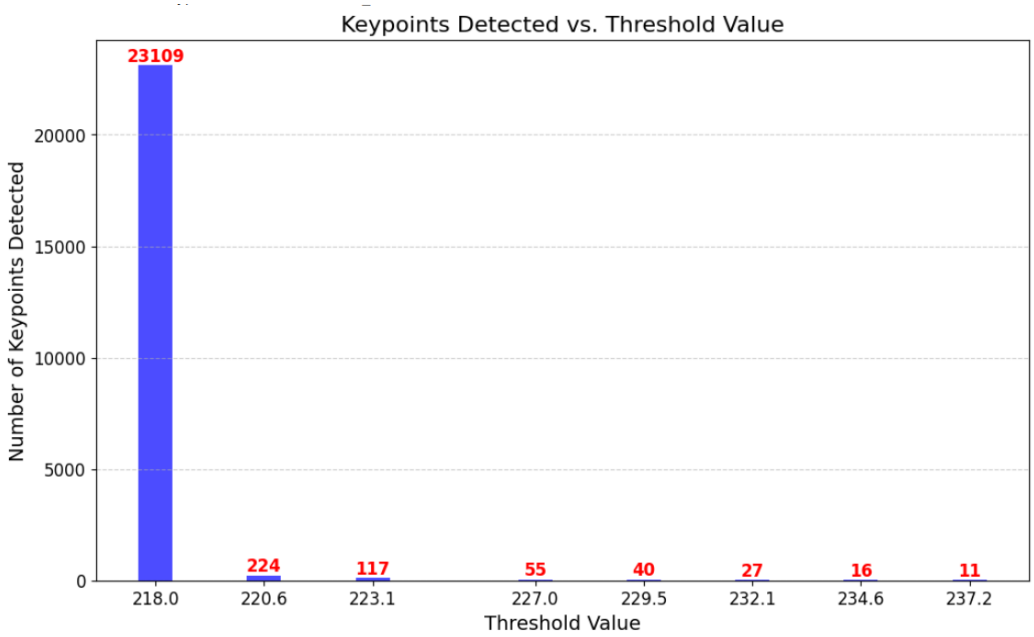

Figure 10: Detected Keypoints in School Image

2 Comparison of ORB Descriptors and Keypoint Detection

To analyze the performance of ORB descriptors, we computed local features using ORB's `compute` function with keypoints detected from our custom Harris detector. Additionally, as an experiment, we utilized the ORB `detect` function to automatically detect keypoints and then applied OpenCV's built-in Harris and FAST methods for further comparison.

The images below showcase keypoints detected using our descriptor with ORB `compute`, as well as keypoints obtained using ORB `detect` with OpenCV Harris and FAST. The images displaying our descriptor include the threshold values used for detection.

2.1 Threshold Values for Different Image Conditions

The following table summarizes the threshold values used for Harris and FAST detectors across different image conditions.

Image Condition	**Harris Threshold Scale**	**FAST Threshold**
Original Image	**0.01**	**30**
Darker	**0.01**	**30**
Brighter	**0.01**	**30**
Rotated	**0.01**	**30**
Friends (noisy background)	**0.05**	**40**
Salon (noisy background)	**0.05**	**40**
Blurred	**0.08**	**2**
Noisy	**0.45**	**400**
Pixelated	**0.25**	**500**
School	**0.08**	**50**

Table 1: Threshold values for different image conditions

2.2 Analysis of Threshold Selection

The selection of threshold values was influenced by the nature of the images:

- Blurred Images: The threshold for FAST was set to only 2 because blurring reduces the contrast at edges, making keypoints harder to detect. A lower threshold ensures that even faint corners are identified.
- Noisy Images: Higher thresholds were required to filter out false keypoints caused by noise. Harris needed 0.45, while FAST required 400, preventing excessive detections in random noise.
- Pixelated Images:Due to sharp artificial edges in pixelated images, high thresholds (0.25 for Harris and 500 for FAST) helped avoid excessive detection at blocky transitions.
- Noisy Backgrounds (Friends and Salon): A moderate increase in threshold helped reduce the number of non-distinctive keypoints in cluttered regions.
- Brightness and Rotation Variations: The same threshold values were retained for these conditions as they do not significantly affect keypoint localization.

Comparing my Harris implementation with OpenCV's Harris and FAST detectors, FAST produced the best results. This is because FAST is specifically designed for speed and efficiency, making it more robust in detecting keypoints across different transformations. While both Harris versions performed well, especially for rotation and brightness changes, they struggled with noise, blur, and background variations. FAST, on the other hand, adapted better to these challenging conditions due to its corner detection approach, which is less sensitive to such distortions.

Figure 1: My Harris + ORB.compute (origg_points.png)

Figure 2: ORB.detect + OpenCv Harris (haris_nivno_origg.png)

Figure 3: ORB.detect + OpenCv FAST (fast_origg.png)

Figure 4: My Harris + ORB.compute (dark_points.png)

Figure 5: ORB.detect + OpenCv Harris (haris_nivno_Dark.png)

Figure 6: ORB.detect + OpenCv FAST (fast_dark.png)

Figure 7: My Harris + ORB.compute (bright_points.png)

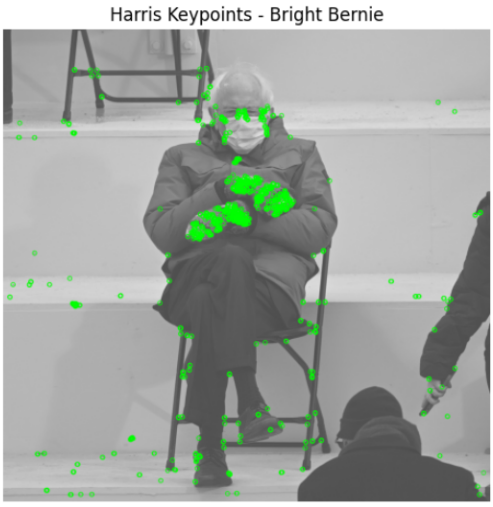

Figure 8: ORB.detect + OpenCv Harris (haris_nivno_bright.png)

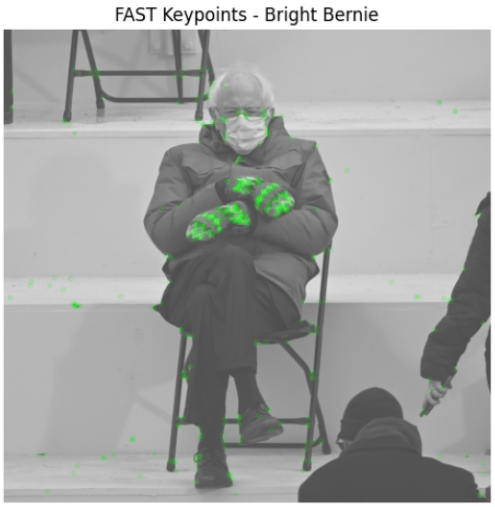

Figure 9: ORB.detect + OpenCv FAST (fast_bright.png)

Figure 10: My Harris + ORB.compute (180_points.png)

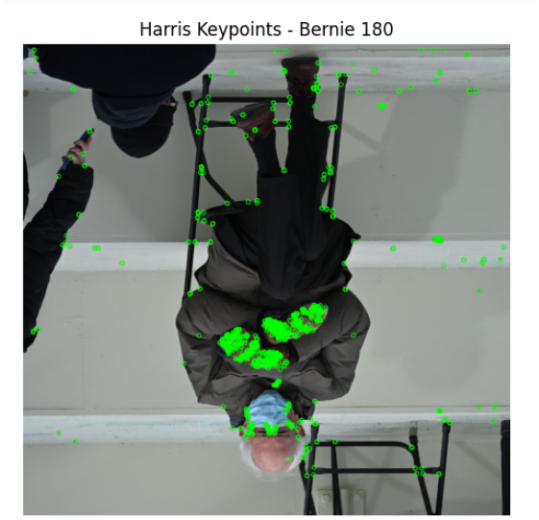

Figure 11: ORB.detect + OpenCv Harris (haris_nivno_180.png)

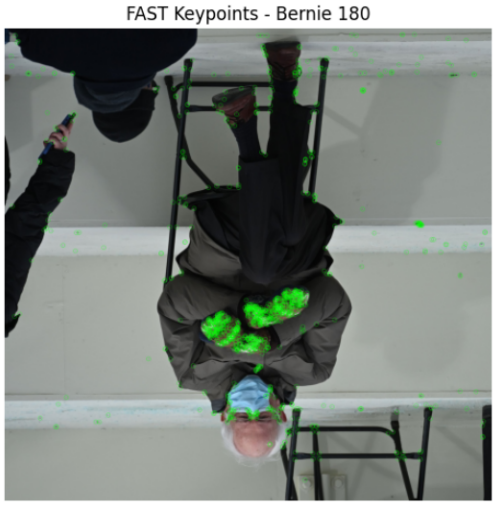

Figure 12: ORB.detect + OpenCv FAST (fast_180.png)

Figure 13: My Harris + ORB.compute (blur_points.png)

Figure 14: ORB.detect + OpenCv Harris (haris_nivno_blured.png)

Figure 15: ORB.detect + OpenCv FAST (fast_blur.png)

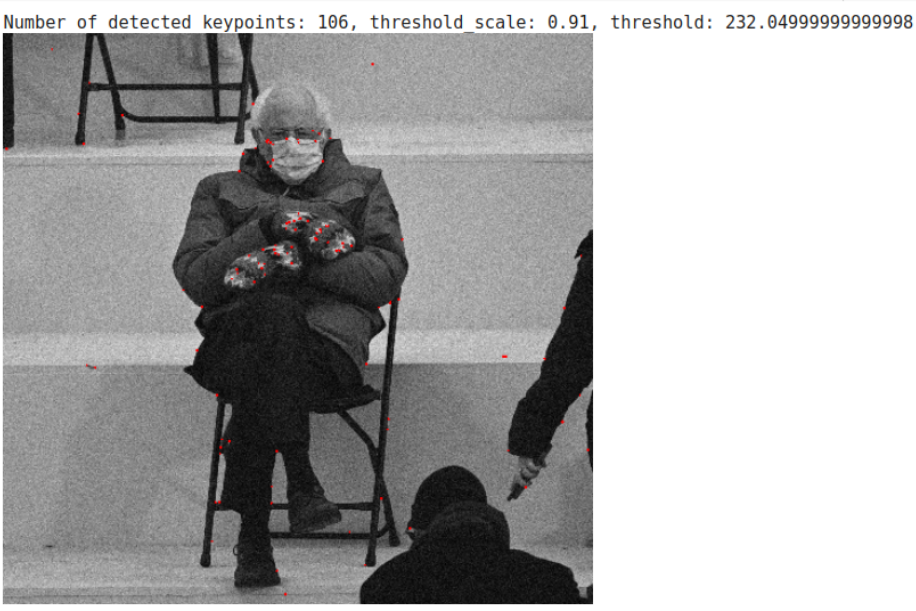

Figure 16: My Harris + ORB.compute (noisy2_points.png)

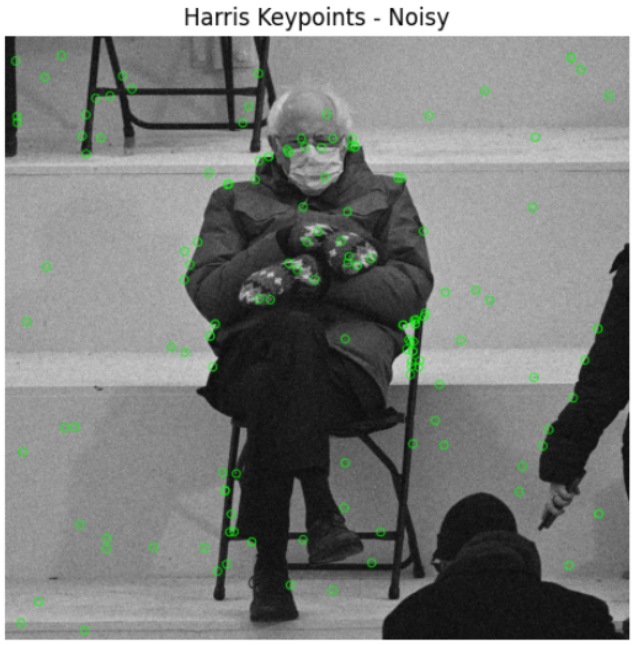

Figure 17: ORB.detect + OpenCv Harris (haris_nivno_noisy.png)

Figure 18: ORB.detect + OpenCv FAST (fats_noisy.png)

Figure 19: My Harris + ORB.compute (pixelated_points.png)

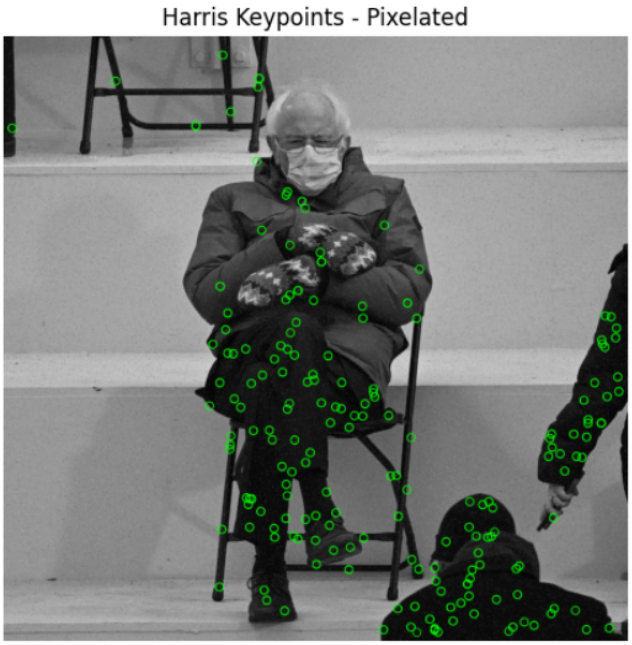

Figure 20: ORB.detect + OpenCv Harris (haris_nivno_pixelated.png)

Figure 21: ORB.detect + OpenCv FAST (fast_pixelated.png)

Figure 22: My Harris + ORB.compute (frineds_points.png)

Figure 23: ORB.detect + OpenCv Harris (haris_nivno_friends.png)

Figure 24: ORB.detect + OpenCv FAST (fast_friends.png)

Figure 25: My Harris + ORB.compute (salon_points.png)

Figure 26: ORB.detect + OpenCv Harris (haris_nivno_salon.png)

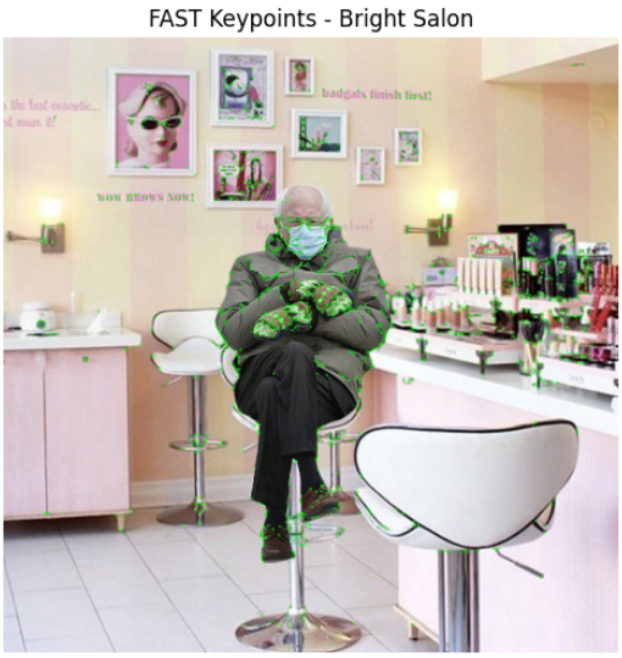

Figure 27: ORB.detect + OpenCv FAST (fast_salom.png)

Figure 28: My Harris + ORB.compute (school_points.png)

Figure 29: ORB.detect + OpenCv Harris (haris_nivno_school.png)

Figure 30: ORB.detect + OpenCv FAST (fast_school.png)

2 Distance functions

As shown in the appendix, I implemented a custom feature matching function using two different approaches: Sum of Squared Differences (SSD) and Lowe's Ratio Test. This function calculates the distance between feature descriptors to identify the best matches between keypoints in two images.

The SSD method computes the squared Euclidean distance between descriptors and selects the closest match for each feature. To discard poor matches, it applies a distance threshold, ensuring only high-confidence matches are retained.

The Ratio Test, on the other hand, helps eliminate ambiguous matches by comparing the best and second-best distances for each feature. If the best match is significantly closer (i.e., the distance is less than a certain ratio of the second-best match), it is accepted; otherwise, it is discarded. This ensures that only distinct, well-matching keypoints are selected, reducing false positives caused by repetitive or noisy features.

This refined approach enhances the accuracy of feature matching, making it more robust against mismatches and improving object detection performance.

3 Matching

Using both the SSD and ratio matching strategies, I obtained the following results. I will present some of them here. For the original image and its 180-degree rotated version, the matching was nearly perfect, as the Harris detector is robust to rotation. The same applied to the darker and brighter images.

However, for the blurred, noisy, and pixelated images, matching became more challenging. This is because the feature calculation window captures noise, and in blurred images, the noise is further smoothed, altering the computed values. Similarly, sharpening affects the gradients, making them slightly different from the original.

The most difficult cases were images with different backgrounds. The descriptors included background information, which differed from the plain white background of the original image. However, I observed that the edges of the gloves were correctly matched, suggesting that even though these images were significantly smaller and their descriptors were calculated at a different scale, the Harris detector was still powerful enough to match them—especially since they shared the same background (Bernie's jacket).

Below, I present some examples of my Harris-based approach using both SSD and ratio matching. Additionally, I include one example using OpenCV's Harris and FAST detectors, which produced slightly stronger results compared to mine.

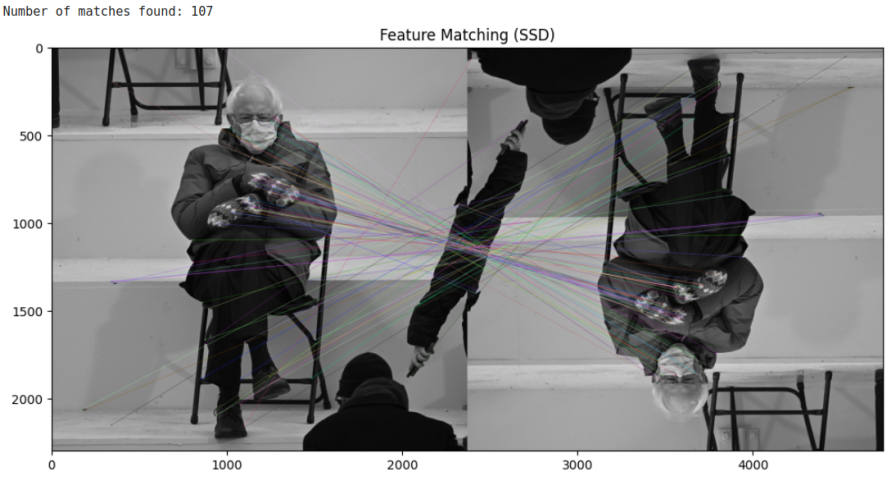

Figure 1: My Haris: 180 - SSD

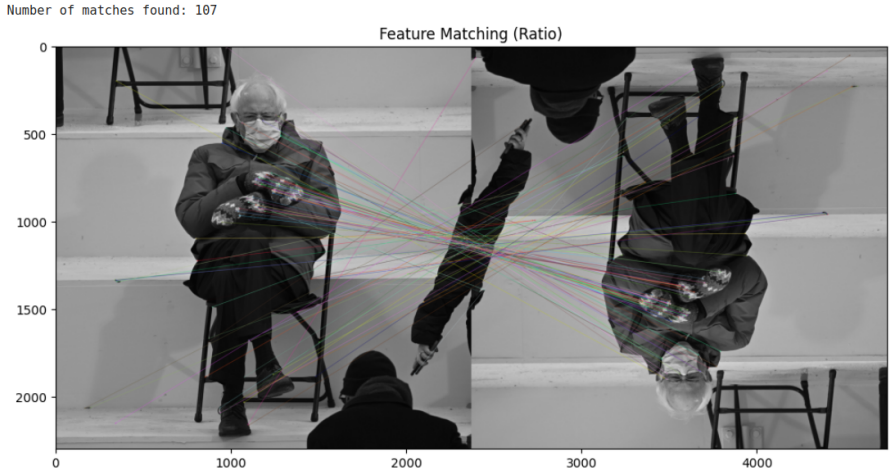

Figure 2: My Haris: 180 - Ratio

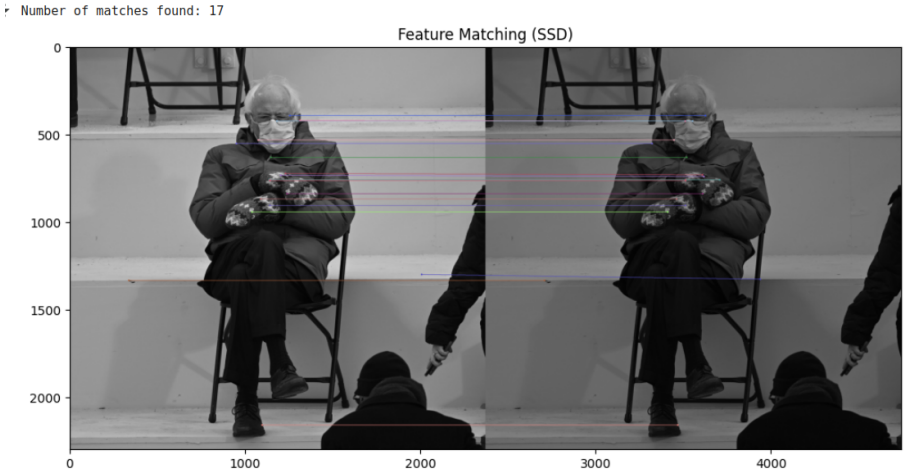

Figure 3: My Haris: Dark - SSD

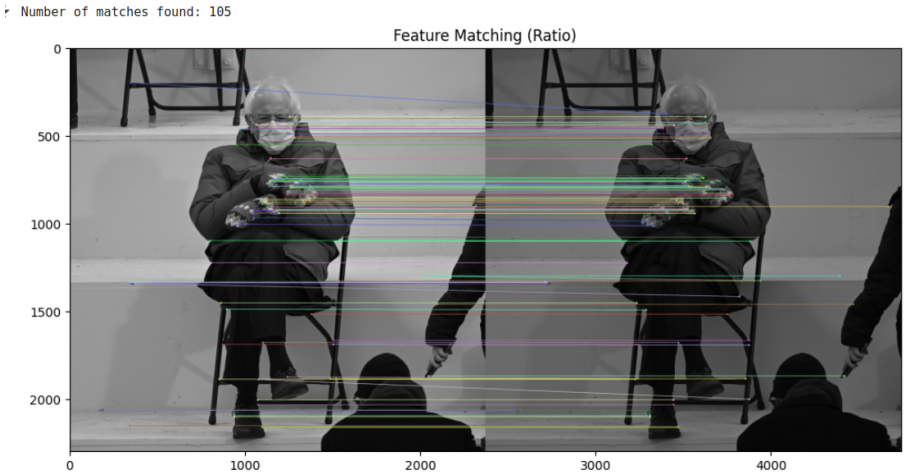

Figure 4: My Haris: Dark - Ratio

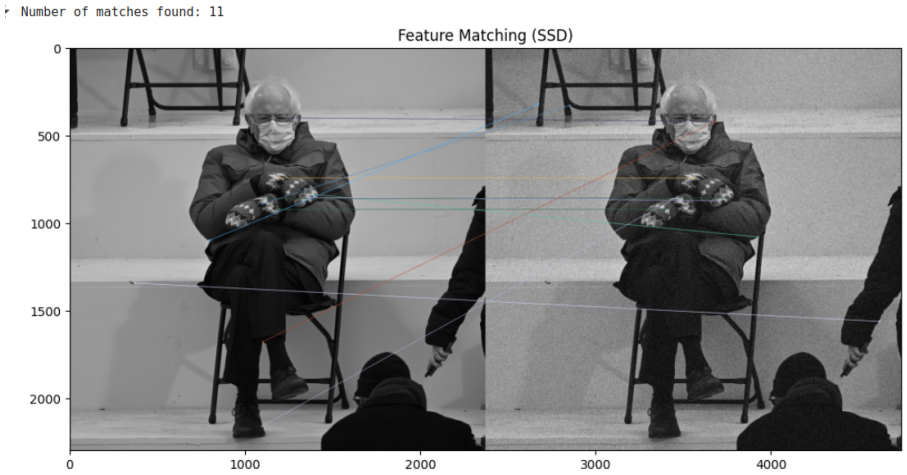

Figure 5: My Haris: Noisy - SSD

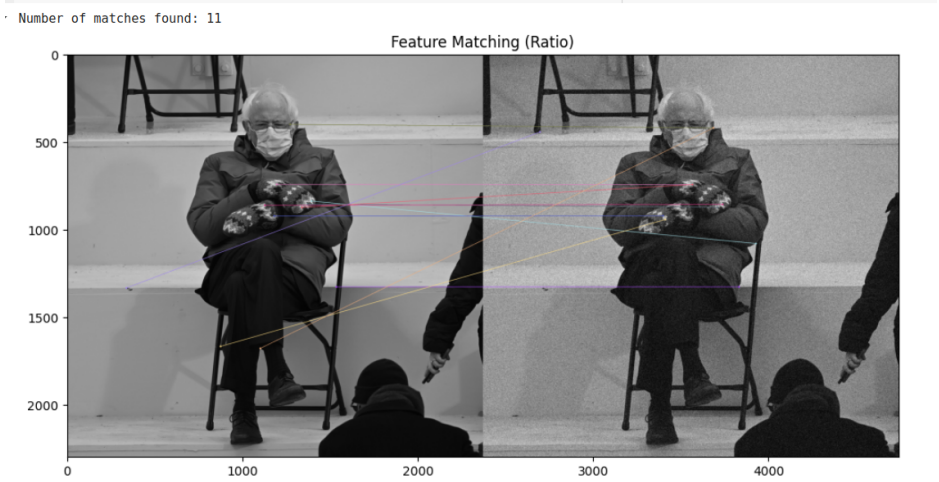

Figure 6: My Haris: Noisy - Ratio

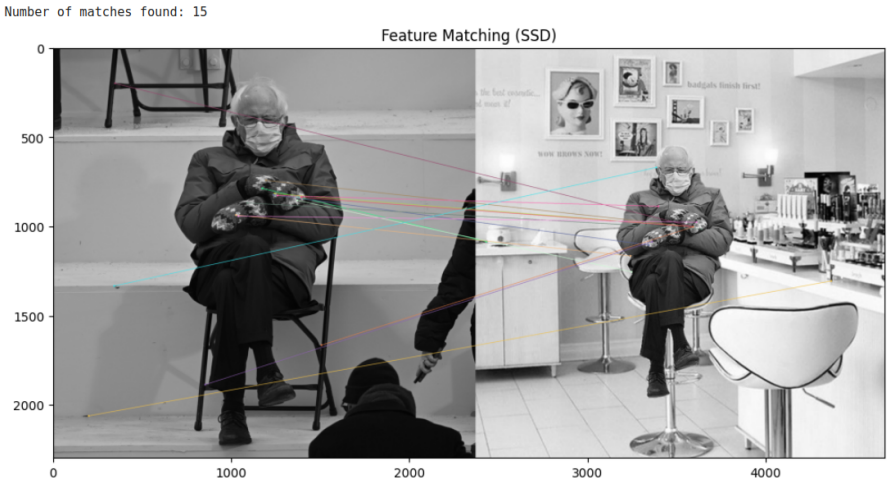

Figure 7: My Haris: Salon - SSD

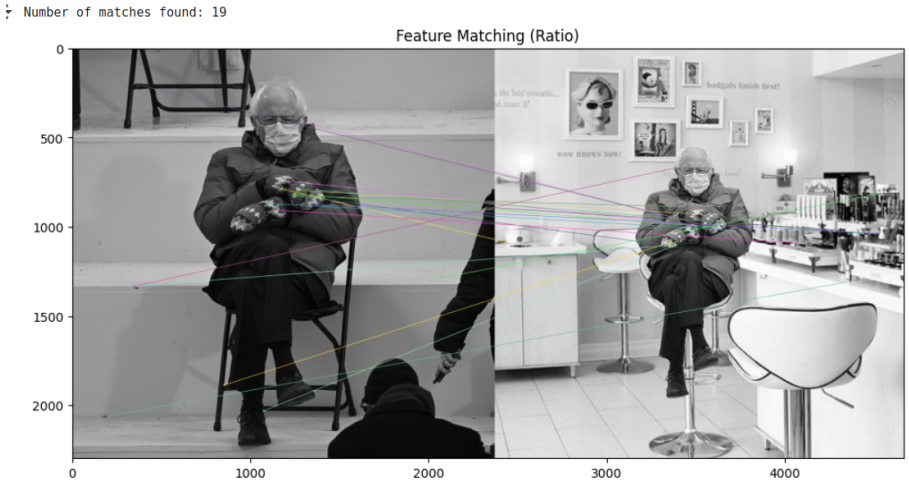

Figure 8: My Haris: Salon - Ratio

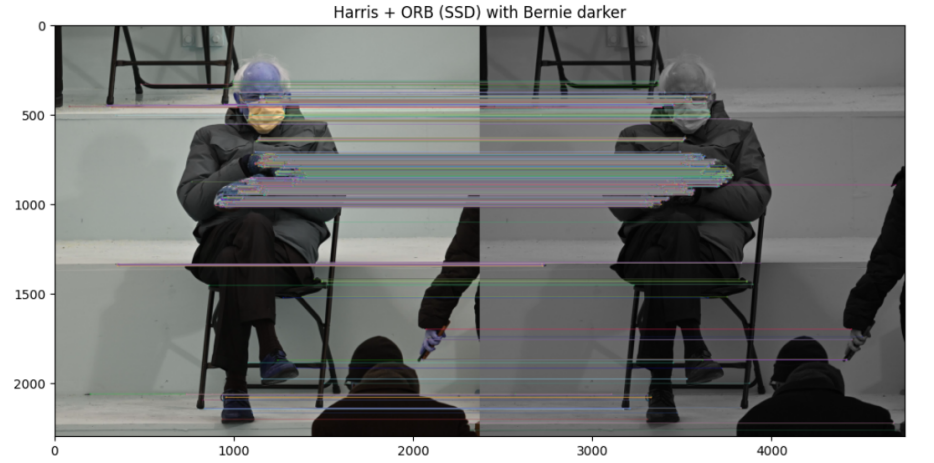

Figure 9: Their Haris: Dark - SSD

Figure 10: Their Haris: Dark - Ratio

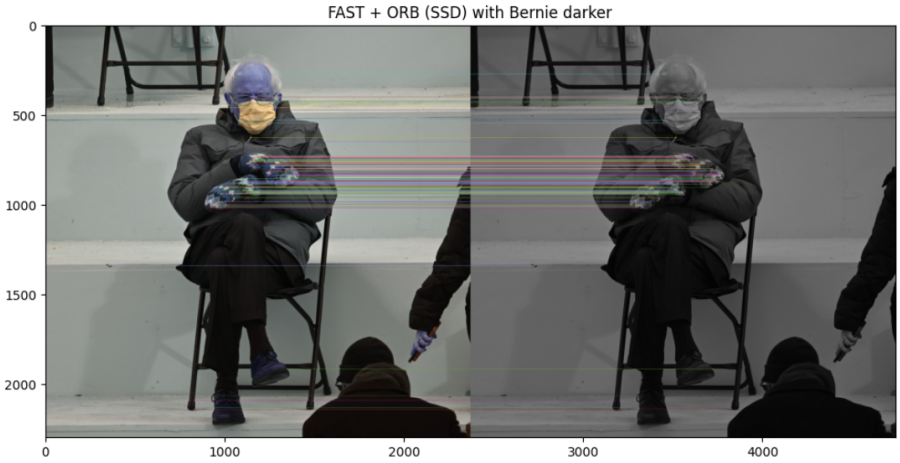

Figure 11: Their FAST: Dark - SSD

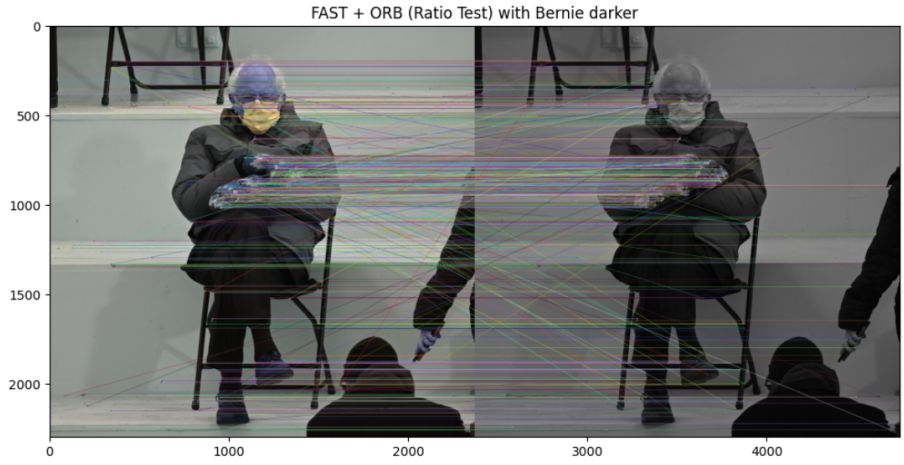

Figure 12: Their FAST: Dark - Ratio

A Appendix: Code Implementation

```python
import cv2
import numpy as np
import scipy.ndimage
from google.colab.patches import cv2_imshow

def HarrisPointsDetector(image, k=0.05, threshold_scale=0.8, visualize=True):
    """
    Detects keypoints using the Harris Corner Detection method.

    Parameters:
        image (numpy.ndarray): Input image (BGR or grayscale).
        k (float): Harris corner constant.
        threshold_scale (float): Scale for selecting strong keypoints.
        visualize (bool): Whether to display detected corners.

    Returns:
        keypoints (list of cv2.KeyPoint): List of OpenCV KeyPoint objects.
    """

    # Convert to grayscale if needed
    if len(image.shape) == 3:
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    else:
        gray = image.copy()

    gray = np.float32(gray)  # Convert to float32 for precision

    #gray = cv2.GaussianBlur(gray, (3, 3), 3)

    # #Apply sharpening to counter excessive blurring
    # sharpening_kernel = np.array([[-1, -1, -1],
    #                               [-1,  9, -1],
    #                               [-1, -1, -1]])
    # gray = cv2.filter2D(gray, -1, sharpening_kernel)




    # Compute gradients using 3x3 Sobel operator (with reflection padding)
    Ix = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3, borderType=cv2.BORDER_REFLECT)
    Iy = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3, borderType=cv2.BORDER_REFLECT)

    # Compute gradient magnitude and orientation (in degrees)
    magnitude = np.sqrt(Ix**2 + Iy**2)
    orientation = np.arctan2(Iy, Ix) * (180 / np.pi)  # Convert radians to degrees

    # Compute second-moment matrix elements
    Ixx = Ix ** 2
    Iyy = Iy ** 2
    Ixy = Ix * Iy

    # Apply 5x5 Gaussian weighting (sigma = 0.5)
    gaussian_filter = (5, 5)
    sigma = 0.5
    Sxx = cv2.GaussianBlur(Ixx, gaussian_filter, sigma)
    Syy = cv2.GaussianBlur(Iyy, gaussian_filter, sigma)
    Sxy = cv2.GaussianBlur(Ixy, gaussian_filter, sigma)

    # Compute Harris response
    detM = (Sxx * Syy) - (Sxy ** 2)
    traceM = Sxx + Syy
    R = detM - k * (traceM ** 2)

    # Normalize R for thresholding
    R_norm = cv2.normalize(R, None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX)
    threshold = threshold_scale * R_norm.max()

    # Find local maxima in a 7x7 neighborhood
    local_max = (R_norm == scipy.ndimage.maximum_filter(R_norm, size=7))
```

```
    keypoints = []
    for y, x in zip(*np.where(local_max & (R_norm > threshold))):
        keypoints.append(cv2.KeyPoint(float(x), float(y), 1, orientation[y, x]))  # Store orientation

    print(f"Number of detected keypoints: {len(keypoints)}, threshold_scale: {threshold_scale}, threshold: {threshold}")

    # Visualization
    if visualize:
        output_image = cv2.cvtColor(gray, cv2.COLOR_GRAY2BGR)
        for kp in keypoints:
            cv2.circle(output_image, (int(kp.pt[0]), int(kp.pt[1])), 4, (0, 0, 255), -1)  # Red dots for keypoints
        # Resize the output image (to 50% of its original size)
        output_image_resized = cv2.resize(output_image, (0, 0), fx=0.5, fy=0.5)  # Resize by 50%
        cv2_imshow(output_image_resized)

    return keypoints, threshold


image = cv2.imread("bernieShoolLunch.jpeg")

# Detect keypoints
keypoints = HarrisPointsDetector(image, k=0.05, threshold_scale=0.87)


import cv2
import numpy as np
import scipy.ndimage
import matplotlib.pyplot as plt

def evaluate_thresholds(image):
    threshold_scales =[0.855, 0.865, 0.875, 0.89, 0.9, 0.91, 0.92, 0.93]
    num_keypoints = []
    actual_thresholds = []

    for threshold_scale in threshold_scales:
        keypoints, threshold = HarrisPointsDetector(image, threshold_scale=threshold_scale, visualize=False)
        num_keypoints.append(len(keypoints))
        actual_thresholds.append(threshold)

    plt.figure(figsize=(12, 7))
    bars = plt.bar(actual_thresholds, num_keypoints, color='b', alpha=0.7)

    plt.xticks(actual_thresholds, labels=[f"{t:.1f}" for t in actual_thresholds], fontsize=12)
    plt.yticks(fontsize=12)
    plt.xlabel("Threshold Value", fontsize=14)
    plt.ylabel("Number of Keypoints Detected", fontsize=14)
    plt.title("Keypoints Detected vs. Threshold Value", fontsize=16)
    plt.grid(axis='y', linestyle='--', alpha=0.6)

    for bar, txt in zip(bars, num_keypoints):
        plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.5, str(txt),
                 ha='center', va='bottom', fontsize=12, color='red', weight='bold')

    plt.show()


image = cv2.imread("bernieShoolLunch.jpeg")
evaluate_thresholds(image)


image_origg = cv2.imread("bernieSanders.jpg")

# Detect keypoints
keypoints_origg, threshold_origg = HarrisPointsDetector(image_origg, threshold_scale=0.841)

image_noisy = cv2.imread("bernieNoisy2.png")
```

```
137
138 # Detect keypoints
139 keypoints_noisy, threshold_noisy = HarrisPointsDetector(image_noisy, threshold_scale=0.91)
140
141 image_friends = cv2.imread("BernieFriends.png")
142
143 # Detect keypoints
144 keypoints_friends, _ = HarrisPointsDetector(image_friends, threshold_scale=0.77)
145
146 image_school = cv2.imread("bernieShoolLunch.jpeg")
147
148 # Detect keypoints
149 keypoints_school, _ = HarrisPointsDetector(image_school, threshold_scale=0.87)
150
151
152 image_blur = cv2.imread("bernieMoreblurred.jpg")
153
154 # Detect keypoints
155 keypoints_blur, _ = HarrisPointsDetector(image_blur, threshold_scale=0.887)
156
157 image_180 = cv2.imread("bernie180.jpg")
158
159 # Detect keypoints
160 keypoints_180, _ = HarrisPointsDetector(image_180, threshold_scale=0.841)
161
162 image_pixel = cv2.imread("berniePixelated2.png")
163
164 # Detect keypoints
165 keypoints_pixel, _ = HarrisPointsDetector(image_pixel, threshold_scale=0.75
166
167 image_bright = cv2.imread("brighterBernie.jpg")
168
169 # Detect keypoints
170 keypoints_bright, _ = HarrisPointsDetector(image_bright, threshold_scale=0.837)
171
172
173 image_dark = cv2.imread("darkerBernie.jpg")
174
175 # Detect keypoints
176 keypoints_dark, _ = HarrisPointsDetector(image_dark, threshold_scale=0.83)
177
178 image_salon = cv2.imread("bernieBenefitBeautySalon.jpeg")
179
180 # Detect keypoints
181 keypoints_salon, _ = HarrisPointsDetector(image_salon, threshold_scale=0.55)
182
183 import cv2
184 import numpy as np
185
186 def featureDescriptor(image, keypoints):
187     """
188     Computes feature descriptors for given keypoints using ORB.
189
190     Parameters:
191         image (numpy.ndarray): Input grayscale image.
192         keypoints (list of cv2.KeyPoint): Detected keypoints with location and orientation.
193
194     Returns:
195         descriptors (numpy.ndarray): ORB descriptors of shape (num_keypoints, feature_dim).
196     """
197
198     # Convert image to grayscale
199     if len(image.shape) == 3:
200         gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
201     else:
202         gray = image.copy()
203
204     # Create ORB detector
205     orb = cv2.ORB_create(500)
206
207
```

```
208
209     if not keypoints:
210         print("No keypoints detected!")
211         return [], np.array([])
212
213
214     # Compute descriptors using ORB
215     keypoints, descriptors = orb.compute(gray, keypoints)
216
217     if descriptors is None:
218         print("No descriptors found!")
219         return np.array([])  # Return empty array if no descriptors are found
220
221     print(f"Computed {descriptors.shape[0]} descriptors with dimension {descriptors.shape[1]}")
222
223     return keypoints,  descriptors
224
225
226 image_origg = cv2.imread("bernieSanders.jpg")
227
228 _, descriptors_origg = featureDescriptor(image_origg, keypoints_origg)
229
230 # Print descriptor shape
231 print(f"Descriptor shape: {descriptors_origg.shape}")
232 image_school = cv2.imread("bernieShoolLunch.jpeg")
233 # Step 2: Compute ORB descriptors for these keypoints
234 _, descriptors_school = featureDescriptor(image_school, keypoints_school)
235
236
237 print(f"Descriptor shape: {descriptors_school.shape}")
238
239 image_friends = cv2.imread("BernieFriends.png")
240 # Step 2: Compute ORB descriptors for these keypoints
241 _, descriptors_friends = featureDescriptor(image_friends, keypoints_friends)
242
243 # Print descriptor shape
244 print(f"Descriptor shape: {descriptors_friends.shape}")
245 image_noisy = cv2.imread("bernieNoisy2.png")
246 # Step 2: Compute ORB descriptors for these keypoints
247 _, descriptors_noisy = featureDescriptor(image_noisy, keypoints_noisy)
248
249 # Print descriptor shape
250 print(f"Descriptor shape: {descriptors_noisy.shape}")
251
252 image_180 = cv2.imread("bernie180.jpg")
253 # Step 2: Compute ORB descriptors for these keypoints
254 _, descriptors_180 = featureDescriptor(image_180, keypoints_180)
255
256 # Print descriptor shape
257 print(f"Descriptor shape: {descriptors_180.shape}")
258
259 image_bright = cv2.imread("brighterBernie.jpg")
260 # Step 2: Compute ORB descriptors for these keypoints
261 _, descriptors_bright = featureDescriptor(image_bright, keypoints_bright)
262
263 # Print descriptor shape
264 print(f"Descriptor shape: {descriptors_bright.shape}")
265
266 image_dark = cv2.imread("darkerBernie.jpg")
267 # Step 2: Compute ORB descriptors for these keypoints
268 _, descriptors_dark = featureDescriptor(image_dark, keypoints_dark)
269
270 # Print descriptor shape
271 print(f"Descriptor shape: {descriptors_dark.shape}")
272
273 image_pixel = cv2.imread("berniePixelated2.png")
274 # Step 2: Compute ORB descriptors for these keypoints
275 _, descriptors_pixel = featureDescriptor(image_pixel, keypoints_pixel)
276
277 # Print descriptor shape
278 print(f"Descriptor shape: {descriptors_pixel.shape}")
```

```
279
280 image_school = cv2.imread("bernieShoolLunch.jpeg")
281 # Step 2: Compute ORB descriptors for these keypoints
282 _, descriptors_school = featureDescriptor(image_school, keypoints_school)
283
284 # Print descriptor shape
285 print(f"Descriptor␣shape:␣{descriptors_school.shape}")
286
287
288 image_blur = cv2.imread("bernieMoreblurred.jpg")
289 # Step 2: Compute ORB descriptors for these keypoints
290 _, descriptors_blur = featureDescriptor(image_blur, keypoints_blur)
291
292 # Print descriptor shape
293 print(f"Descriptor␣shape:␣{descriptors_blur.shape}")
294
295 image_salon = cv2.imread("bernieBenefitBeautySalon.jpeg")
296 # Step 2: Compute ORB descriptors for these keypoints
297 _, descriptors_salon = featureDescriptor(image_salon, keypoints_salon)
298
299 # Print descriptor shape
300 print(f"Descriptor␣shape:␣{descriptors_salon.shape}")
301
302
303
304 import numpy as np
305 import cv2
306 import matplotlib.pyplot as plt
307 import random
308 from scipy.spatial.distance import cdist
309
310 def match_and_visualize(image1, keypoints1, descriptors1, image2, keypoints2, descriptors2,
        matcher_type="Ratio", max_distance=85000, ratio_threshold=0.8):
311     """
312     Matches features between two images and visualizes them with vibrant colored lines.
313
314     Parameters:
315         image1, image2: Input images.
316         keypoints1, keypoints2: Detected keypoints.
317         descriptors1, descriptors2: Feature descriptors.
318         matcher_type: "SSD" for Sum of Squared Differences, "Ratio" for Lowes Ratio Test.
319         max_distance: Threshold for SSD distance filtering.
320         ratio_threshold: Ratio threshold for Lowes Ratio Test.
321
322     Returns:
323         Matches visualization.
324     """
325
326     # Resize second image to match height of first image
327     h1, w1 = image1.shape[:2]
328     h2, w2 = image2.shape[:2]
329     scale_factor = h1 / h2
330     image2_resized = cv2.resize(image2, (int(w2 * scale_factor), h1))
331
332     # Convert images to grayscale if needed
333     image1_gray = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY) if len(image1.shape) == 3 else image1
334     image2_gray = cv2.cvtColor(image2_resized, cv2.COLOR_BGR2GRAY) if len(image2.shape) == 3 else
            image2_resized
335
336     # Ensure descriptors exist
337     if descriptors1 is None or descriptors2 is None:
338         print("No␣descriptors␣found␣in␣one␣or␣both␣images!")
339         return []
340
341     # Convert descriptors to float for compatibility with cdist
342     descriptors1 = descriptors1.astype(np.float32)
343     descriptors2 = descriptors2.astype(np.float32)
344
345     matches = []
346
347     if matcher_type == "SSD":
```

```
        # Compute SSD matrix
        ssd_matrix = cdist(descriptors1, descriptors2, metric='sqeuclidean')
        best_matches = np.argmin(ssd_matrix, axis=1)

        for i, j in enumerate(best_matches):
            distance = ssd_matrix[i, j]
            if distance < max_distance:  # Thresholding to remove bad matches
                matches.append(cv2.DMatch(_queryIdx=i, _trainIdx=j, _distance=distance))

    elif matcher_type == "Ratio":
        # Compute SSD and apply Ratio Test
        ssd_matrix = cdist(descriptors1, descriptors2, metric='sqeuclidean')
        sorted_indices = np.argsort(ssd_matrix, axis=1)[:, :2]  # Get top 2 matches

        for i in range(len(descriptors1)):
            best_idx, second_best_idx = sorted_indices[i]
            best_distance, second_best_distance = ssd_matrix[i, best_idx], ssd_matrix[i,
                second_best_idx]

            # Apply Lowe's Ratio Test
            if best_distance < ratio_threshold * second_best_distance:
                matches.append(cv2.DMatch(_queryIdx=i, _trainIdx=best_idx, _distance=best_distance
                    ))

    print(f"Number of matches found: {len(matches)}")

    # Create a combined image for drawing matches
    output_image = np.zeros((h1, w1 + image2_resized.shape[1], 3), dtype=np.uint8)

    # Convert grayscale images to 3-channel for color drawing
    output_image[:, :w1] = cv2.cvtColor(image1_gray, cv2.COLOR_GRAY2BGR)
    output_image[:, w1:] = cv2.cvtColor(image2_gray, cv2.COLOR_GRAY2BGR)

    # Draw matches with vibrant colors
    for match in matches:
        pt1 = tuple(map(int, keypoints1[match.queryIdx].pt))
        pt2 = tuple(map(int, keypoints2[match.trainIdx].pt))
        pt2 = (int(pt2[0] * scale_factor) + w1, int(pt2[1] * scale_factor))  # Adjust pt2 after
            resizing

        # Generate a vibrant random color
        color = tuple(random.randint(50, 255) for _ in range(3))

        # Draw line between matched points
        cv2.line(output_image, pt1, pt2, color, thickness=2)

        # Draw keypoints with bold circles
        cv2.circle(output_image, pt1, 5, color, -1)
        cv2.circle(output_image, pt2, 5, color, -1)

    # Show the results
    plt.figure(figsize=(12, 6))
    plt.imshow(cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB))
    plt.title(f"Feature Matching ({matcher_type})")
    plt.show()

    return matches

from google.colab.patches import cv2_imshow

match_and_visualize(image_origg, keypoints_origg, descriptors_origg, image_school,
    keypoints_school, descriptors_school, matcher_type="Ratio")


import cv2
import numpy as np
import matplotlib.pyplot as plt
import random
from scipy.spatial.distance import cdist

def harris_corner_detector(image, block_size=2, ksize=3, k=0.05, threshold=0.08):
```

```
    """
    Detects keypoints using Harris Corner Detector.

    Parameters:
        image: Grayscale input image.
        block_size: Neighborhood size.
        ksize: Aperture parameter for the Sobel operator.
        k: Harris detector free parameter.
        threshold: Threshold for selecting keypoints.

    Returns:
        Keypoints list.
    """
    harris_response = cv2.cornerHarris(image, block_size, ksize, k)
    harris_response = cv2.dilate(harris_response, None)  # Dilate for better visualization
    keypoints = np.argwhere(harris_response > threshold * harris_response.max())
    keypoints = [cv2.KeyPoint(float(x[1]), float(x[0]), 4) for x in keypoints]
    return keypoints

def fast_detector(image, threshold=50):
    """
    Detects keypoints using FAST feature detector.

    Parameters:
        image: Grayscale input image.
        threshold: FAST threshold.

    Returns:
        Keypoints list.
    """
    fast = cv2.FastFeatureDetector_create(threshold)
    keypoints = fast.detect(image, None)
    return keypoints

def compute_orb_descriptors(image, keypoints):
    """
    Computes ORB descriptors for the given keypoints.

    Parameters:
        image: Grayscale input image.
        keypoints: Keypoints list.

    Returns:
        Keypoints and descriptors.
    """
    orb = cv2.ORB_create()
    keypoints, descriptors = orb.compute(image, keypoints)
    return keypoints, descriptors

def match_features_ssd(descriptors1, descriptors2, max_distance=100):
    """
    Matches features using Sum of Squared Differences (SSD).

    Parameters:
        descriptors1, descriptors2: Feature descriptors.
        max_distance: Threshold for SSD distance filtering.

    Returns:
        List of cv2.DMatch objects.
    """
    if descriptors1 is None or descriptors2 is None:
        return []

    descriptors1 = descriptors1.astype(np.float32)
    descriptors2 = descriptors2.astype(np.float32)

    ssd_matrix = cdist(descriptors1, descriptors2, metric='sqeuclidean')
    best_matches = np.argmin(ssd_matrix, axis=1)

    matches = []
    for i, j in enumerate(best_matches):
```

```
            distance = ssd_matrix[i, j]
            if distance < max_distance:
                matches.append(cv2.DMatch(_queryIdx=i, _trainIdx=j, _distance=distance))

    return matches

def match_features_ratio(descriptors1, descriptors2, ratio_threshold=0.9):
    """
    Matches features using Lowe's Ratio Test.

    Parameters:
        descriptors1, descriptors2: Feature descriptors.
        ratio_threshold: Ratio test threshold.

    Returns:
        List of cv2.DMatch objects.
    """
    if descriptors1 is None or descriptors2 is None:
        return []

    descriptors1 = descriptors1.astype(np.float32)
    descriptors2 = descriptors2.astype(np.float32)

    ssd_matrix = cdist(descriptors1, descriptors2, metric='sqeuclidean')
    sorted_indices = np.argsort(ssd_matrix, axis=1)[:, :2]  # Get top 2 matches

    matches = []
    for i in range(len(descriptors1)):
        best_idx, second_best_idx = sorted_indices[i]
        best_distance, second_best_distance = ssd_matrix[i, best_idx], ssd_matrix[i,
            second_best_idx]

        if best_distance < ratio_threshold * second_best_distance:
            matches.append(cv2.DMatch(_queryIdx=i, _trainIdx=best_idx, _distance=best_distance))

    return matches

def visualize_matches(image1, keypoints1, image2, keypoints2, matches, title):
    """
    Visualizes feature matches with vibrant colors.

    Parameters:
        image1, image2: Input images.
        keypoints1, keypoints2: Keypoints.
        matches: List of cv2.DMatch objects.
        title: Title of the plot.

    Returns:
        None
    """
    img_matches = cv2.drawMatches(image1, keypoints1, image2, keypoints2, matches, None, flags=cv2
        .DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

    plt.figure(figsize=(12, 6))
    plt.imshow(img_matches)
    plt.title(title)
    plt.show()

def visualize_keypoints(image, keypoints, title="Keypoints Visualization"):
    """
    Draws keypoints on an image and displays it with larger dots.

    Parameters:
        image: Original image.
        keypoints: List of detected keypoints.
        title: Title for the visualization.

    Returns:
        None
    """
    # Increase the size of keypoints by setting a custom diameter
```

```
    for kp in keypoints:
        kp.size = 10 # Adjust size as needed

    image_with_keypoints = cv2.drawKeypoints(image, keypoints, None, color=(0, 255, 0), flags=cv2.
        DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

    plt.figure(figsize=(6, 6))
    plt.imshow(cv2.cvtColor(image_with_keypoints, cv2.COLOR_BGR2RGB))
    plt.title(title)
    plt.axis("off")
    plt.show()




# Load images
image1 = cv2.imread("bernieShoolLunch.jpeg")
image2 = cv2.imread("darkerBernie.jpg")


# Convert to grayscale
image1_gray = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
image2_gray = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)


# Feature detection using Harris
harris_keypoints1 = harris_corner_detector(image1_gray)
harris_keypoints2 = harris_corner_detector(image2_gray)
# Feature detection using FAST
fast_keypoints1 = fast_detector(image1_gray)
fast_keypoints2 = fast_detector(image2_gray)


# Compute ORB descriptors
harris_keypoints1, harris_descriptors1 = compute_orb_descriptors(image1_gray, harris_keypoints1)
harris_keypoints2, harris_descriptors2 = compute_orb_descriptors(image2_gray, harris_keypoints2)


fast_keypoints1, fast_descriptors1 = compute_orb_descriptors(image1_gray, fast_keypoints1)
fast_keypoints2, fast_descriptors2 = compute_orb_descriptors(image2_gray, fast_keypoints2)




visualize_keypoints(image1, harris_keypoints1, "Harris␣Keypoints␣-␣School")
visualize_keypoints(image2, harris_keypoints2, "Harris␣Keypoints␣-␣Pixelated")


visualize_keypoints(image1, fast_keypoints1, "FAST␣Keypoints␣-␣School")
visualize_keypoints(image2, fast_keypoints2, "FAST␣Keypoints␣-␣Pixelated")




# Feature Matching (SSD & Ratio Test) for both Harris & FAST
matches_ssd_harris_1 = match_features_ssd(harris_descriptors1, harris_descriptors2)
matches_ratio_harris_1 = match_features_ratio(harris_descriptors1, harris_descriptors2)

matches_ssd_fast_1 = match_features_ssd(fast_descriptors1, fast_descriptors2)
matches_ratio_fast_1 = match_features_ratio(fast_descriptors1, fast_descriptors2)


# Visualize Results
visualize_matches(image1, harris_keypoints1, image2, harris_keypoints2, matches_ssd_harris_1, "
    Harris␣+␣ORB␣(SSD)␣with␣Bernie␣darker")
visualize_matches(image1, harris_keypoints1, image2, harris_keypoints2, matches_ratio_harris_1, "
    Harris␣+␣ORB␣(Ratio␣Test)␣with␣Bernie␣darker")

visualize_matches(image1, fast_keypoints1, image2, fast_keypoints2, matches_ssd_fast_1, "FAST␣+␣
    ORB␣(SSD)␣with␣Bernie␣darker")
```

```
visualize_matches(image1, fast_keypoints1, image2, fast_keypoints2, matches_ratio_fast_1, "FAST + ORB (Ratio Test) with Bernie darker")
```