Giorgio Mendoza
CS549-E24-E01
Dr. Alexandros Lioulemes

Lab Report #3

Part 0:

In PART 0, we enabled the SURF algorithm in OpenCV by rebuilding the opencv-contrib-python module with the OPENCV_ENABLE_NONFREE flag set to ON. We uninstalled the existing module, installed necessary dependencies, and reinstalled OpenCV from source with the command CMAKE_ARGS="-DOPENCV_ENABLE_NONFREE=ON" pip install --no-binary=opencv-contrib-python opencv-contrib-python, allowing the use of patented algorithms like SURF, etc.

Part 1:

In Part 1, we studied and implemented the SIFT and SURF algorithms using OpenCV tutorials. For SIFT, we learned to detect keypoints, assign orientations, and compute descriptors, noting its robustness and invariance to scale and rotation.

For SURF, we implemented keypoint detection and feature description, understanding its advantages in speed and efficiency over SIFT. These tutorials provided practical experience and highlighted the distinctions between the two feature detection algorithms.

Figure 1: SIFT example

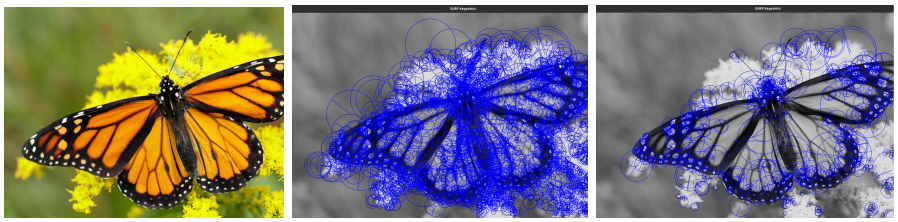

Figure 2: SURF example

The SIFT example successfully detected numerous keypoints in the image, as shown in Figure 1. The left image is the original, while the right image displays keypoints detected by SIFT, marked with circles and lines indicating the orientation and scale. The results demonstrate SIFT's effectiveness in identifying distinctive features across various parts of the image, including the house, trees, and other elements, highlighting its feature detection.

Similarly, the SURF example effectively detected numerous keypoints in the image, as illustrated in Figure 2. The left image shows the original butterfly, while the right image displays the detected keypoints using SURF, marked with blue circles. These circles indicate the location, scale, and orientation of the keypoints.

The numerous blue circles in the SURF example image indicate a high number of detected keypoints, influenced by the `hessianThreshold` parameter in the SURF detector. Adjusting this parameter to a higher value will reduce the number of detected keypoints, highlighting only the most prominent features in the image. This demonstrates the flexibility of the SURF algorithm in controlling the sensitivity of feature detection.

Part 2:

In Part 2, we implemented feature matching using SIFT features by following an OpenCV tutorial. We used the Brute-Force matcher to find the closest matches between feature descriptors in two images. Additionally, we explored the FLANN (Fast Library for Approximate Nearest Neighbors) matcher for faster, real-time feature matching. This helped us understand the practical applications and performance differences between Brute-Force and FLANN matchers.

Figure 3: Reference Images

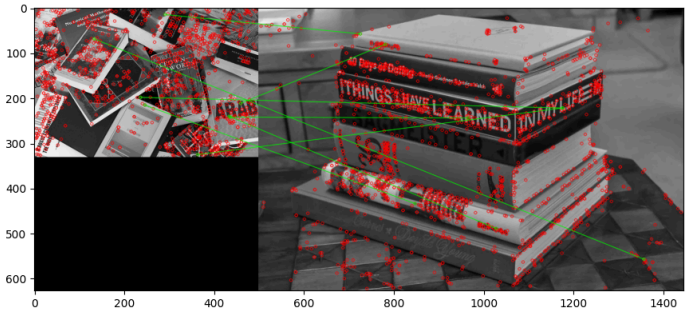

Figure 4: SIFT Results

The SIFT results in Figure 4 demonstrate the effectiveness of the feature matching process. The left image shows the reference books, and the right image depicts a stack of books where SIFT has detected and matched keypoints between the two images. The red circles represent the detected keypoints, and the green lines indicate the matched features. The successful matching of features across both images highlights SIFT's robustness in identifying and comparing distinct image features, even in cluttered and complex scenes.

For this part, I didn't use an isolated object for reference picture 1 that appears in reference picture 2. I just wanted to test if it could detect similar shapes, word styles, etc.

Part 3:

For part 3, we applied both SIFT and SURF feature detectors to extract the interest points for the reference image below. We als applied the Brute-Force and FLANN matching algorithms to identify the book on the table.

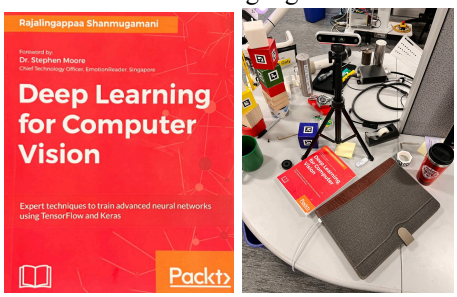

Figure 4: Reference Images

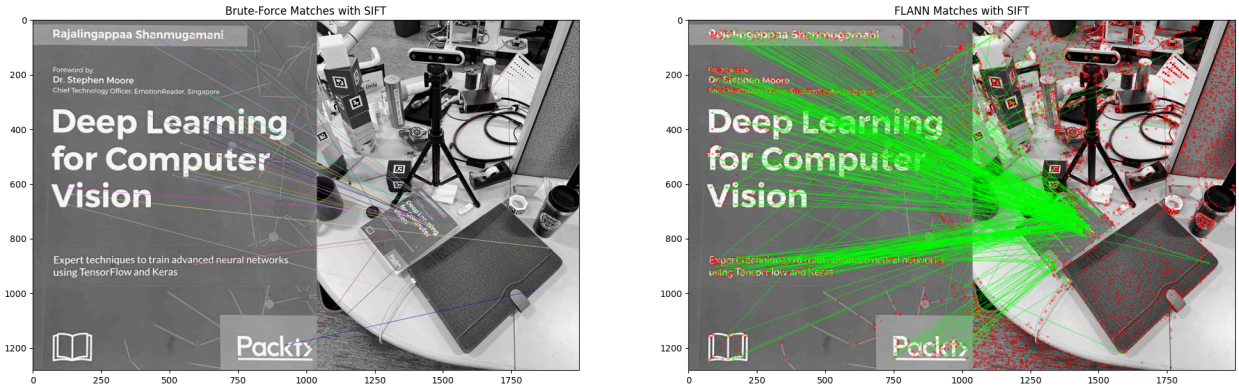

Figure 5: Part 5 Results

In the left image, the Brute-Force matcher with SIFT successfully identified the book but with fewer keypoints and matches, leading to less clutter in the detection. The right image, using the FLANN matcher with SIFT, identified the book with a significantly higher number of keypoints and matches, resulting in more detailed but also noisier detection, as seen by the red points on other objects on the table. This demonstrates that while FLANN is faster and identifies more features, it may also introduce more noise compared to the Brute-Force matcher.

It's also important to note that the algorithms used in this part can be further tuned to detect more or less points depending on the requirements of the application.

Part 4:

In Part 4, we tested different combinations of feature detectors and matchers. We used SIFT with both Brute-Force and FLANN matchers, as well as SURF with Brute-Force and FLANN matchers. These experiments allowed us to compare the accuracy, speed, and robustness of each combination in feature detection and matching.

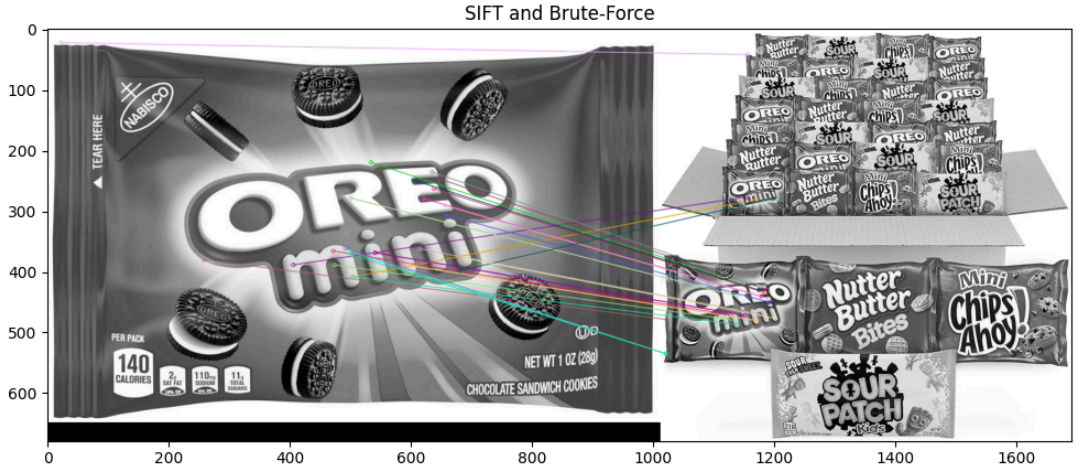

Figure 6: SIFT and Brute-Force

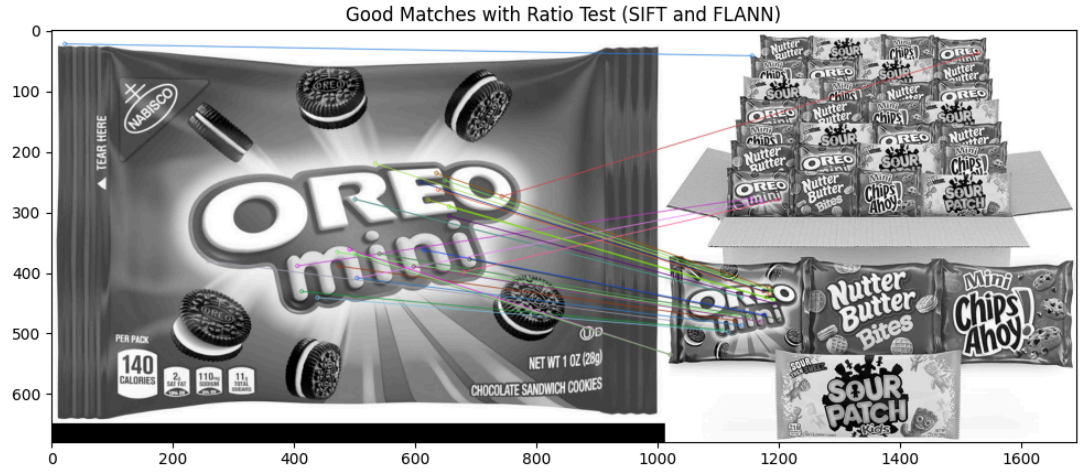

Figure 7: SIFT and FLANN

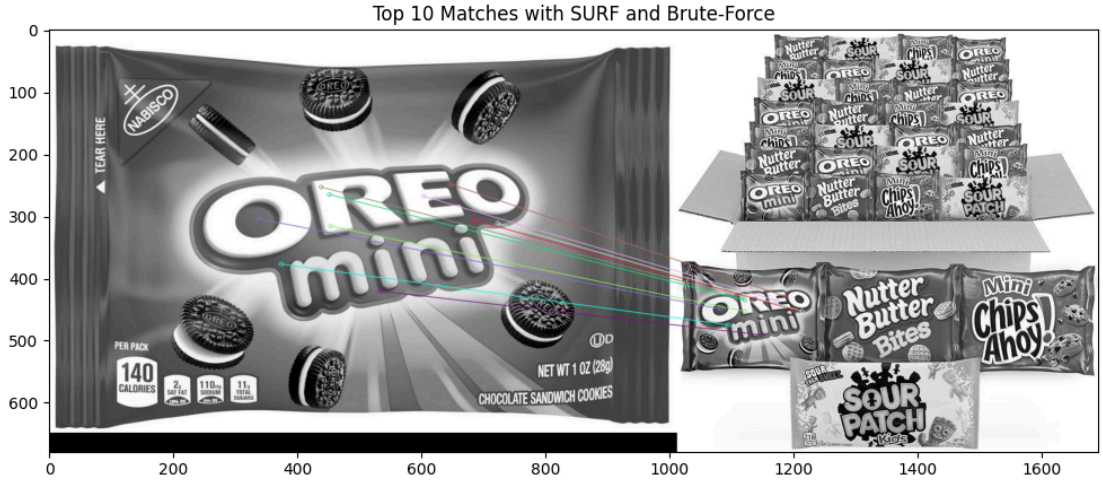

Figure 8: SURF and Brue-Force

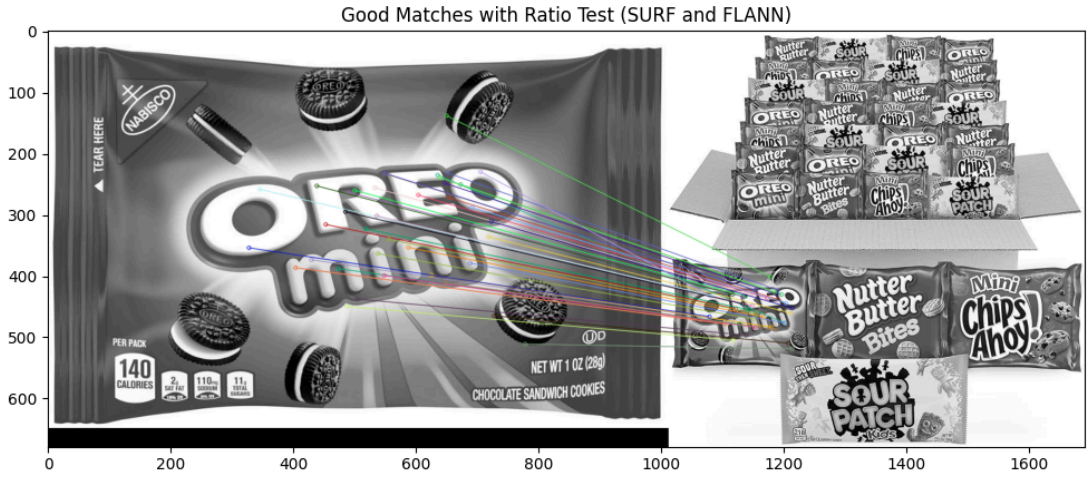

Figure 9: SURF and FLANN

The results are very interesting because, in all cases, the area of most interest was the Oreo logo and some edges of the package. The ratio test (m.distance < 0.55 * n.distance) was also used to maintain consistency in match filtering for all cases except SURF and Brute-Force so they all show a similar performance as shown above. The results of the SIFT & FLANN were also interesting because it was the only one that identified another Oreo package in the far back.

SIFT and Brute-Force: In Figure 6, we used SIFT for feature detection and Brute-Force for matching. The result shows a moderate number of matches, effectively identifying the Oreo Mini pack among the various snacks.

SIFT and FLANN: In Figure 7, SIFT combined with FLANN resulted in a higher number of matches, demonstrating a better detection of the Oreo Mini pack. The FLANN matcher provided a more extensive set of matches, but also included some noise.

SURF and Brute-Force: In Figure 8, using SURF with Brute-Force produced fewer matches compared to SIFT, but the matches were accurate, successfully identifying the Oreo Mini pack. This indicates that SURF is effective but detects fewer features.

SURF and FLANN: In Figure 9, SURF with FLANN showed a significant number of matches, effectively detecting the Oreo Mini pack with high accuracy. The combination was efficient, though it also introduced some noise similar to the SIFT and FLANN results.

Part 5:

In PART 5a, we integrated real-time object detection into our camera/photo-booth app. We defined the detect_and_display function to detect and match features between a reference image and the live camera frame. This function uses a feature detector and a FLANN-based matcher to find keypoints and descriptors, applies Lowe's ratio test to filter good matches, computes the homography matrix, and draws a bounding box around the detected object in the frame.

We initialized the camera and loaded the reference image, set up the SIFT feature detector, and configured the FLANN-based matcher. In the real-time detection loop, frames from the camera were continuously captured, processed to detect the reference object, and displayed. The user could exit the application by pressing the 'q' key.

This setup enabled real-time detection and visualization of the reference object in the live camera feed as shown below.

Figure 10: Reference Image & Identified Image

I also implemented a real-time object detection using ORB and FLANN which works in a similar manner. I had to use that for this part, because I couldn't get the camera to work in my Ubuntu VM. Also, I couldn't install the equivalent of $CMAKE_ARGS="-DOPENCV_ENABLE_NONFREE=ON" pip install --no-binary=opencv-contrib-python opencv-contrib-python for Windows 10. So I used pip install --no-binary :opencv-contrib-python: opencv-contrib-python in Win10.

Figure 10: Reference Image & Identified Image using ORB

For reference, combines the FAST keypoint detector and BRIEF descriptor, with modifications to improve performance and orientation invariance. ORB is free to use (non-patented) and offers a good balance between speed and accuracy, though it may not be as robust as SIFT or SURF for certain applications. ORB was designed to be fast and efficient, so it can also be used for real-time applications.

One interesting observation for both test cases is that they did not perform well under poor lighting conditions. When I turned on my phone flashlight, the algorithms quickly identified the object. This suggests that these algorithms are significantly affected by the camera's capabilities and the lighting of the scene.

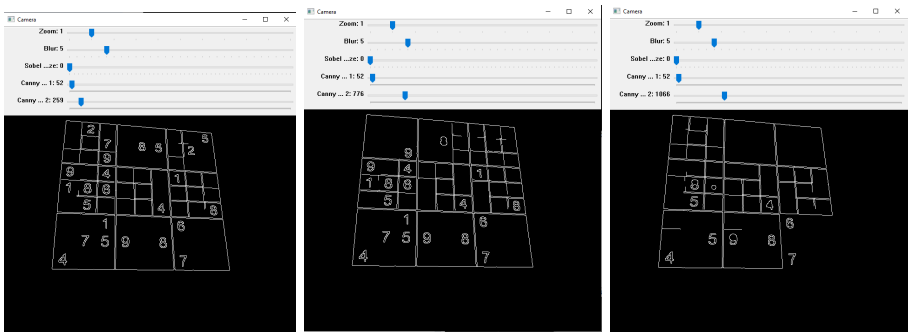

Figure 5: Canny Edge Second Threshold + Small First Threshold Value

Figure 5 uses both the first and second thresholds in the Canny edge detection on a Sudoku puzzle image; the first threshold is maintained at a constant value of 152 across all three images for consistency.

In the leftmost image with a second threshold of 259, the algorithm effectively retains most of the major grid lines while beginning to suppress some of the internal square edges. As the second threshold increases to 776 in the middle image, the edge detection becomes more selective, further reducing the visibility of internal square edges and focusing primarily on the most pronounced edges, such as the outer borders and major grid lines of the Sudoku puzzle.

In the rightmost image, the selectivity is heightened to a point where even some outer borders begin to disappear, showcasing how higher second threshold values progressively eliminate weaker and moderately strong edges, leaving only the most distinct edges visible.

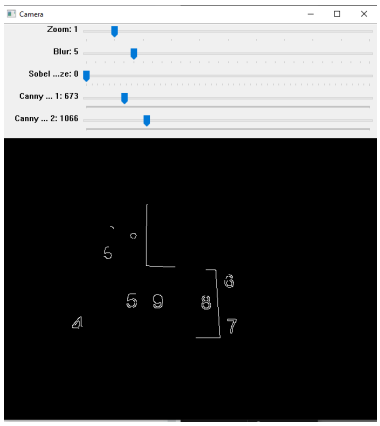

Figure 6: Both Thresholds with High Values

The image above illustrates the case when both thresholds have high values. The key point here is that for a real application it's important to balance both thresholds to retain the areas of interest. Otherwise, even relevant data can be lost.

Part 2:

Part 2 is similar to part 1 except that we are implementing the filters from scratch instead of using the API from OpenCV additionally we are adding the Laplacian filter.

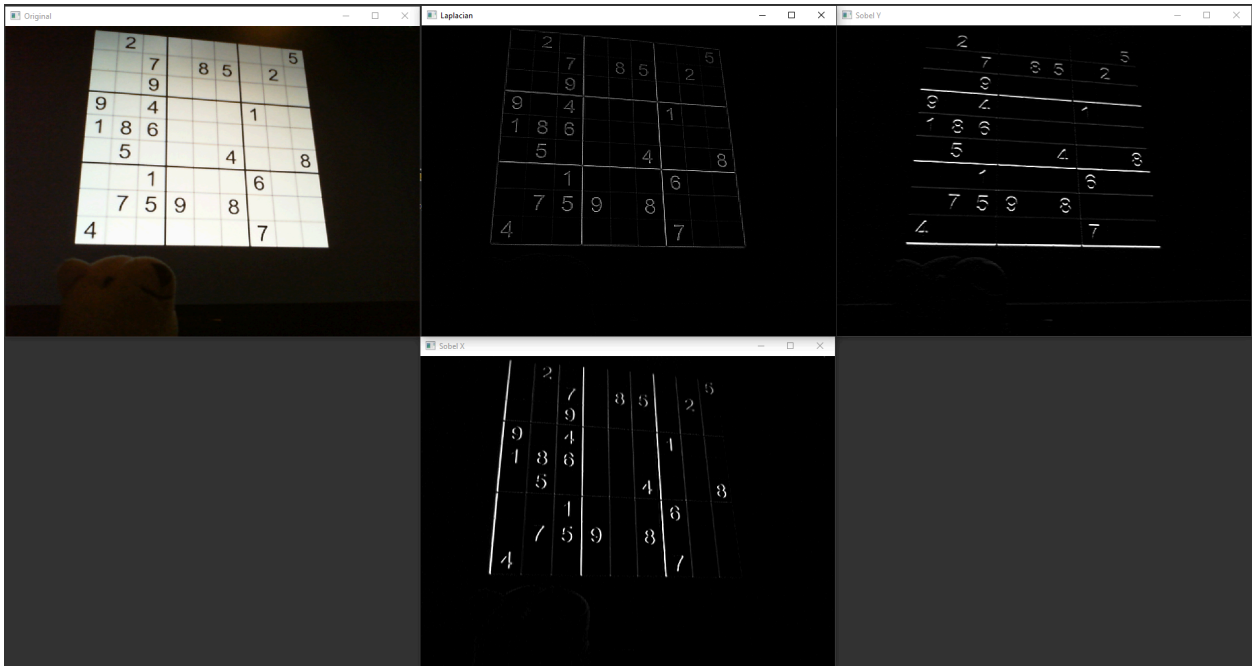

Figure 7: Original Image, Custom Laplacian, Sobel X and Sobel Y comparison

The second window shows the result of applying the custom Laplacian filter which emphasizes areas of rapid intensity change indicative of edges. Unlike the Sobel operator, the Laplacian is isotropic meaning it detects edges in all directions. The image clearly highlights all the boundaries and numbers with strong white lines against a dark background, capturing finer details compared to the original.

The third window demonstrates the effect of the custom Sobel X operator, which is designed to detect horizontal gradients. Therefore, it accentuates vertical edges in the Sudoku grid, as seen by the strong vertical lines and somewhat clearer visibility of the vertical aspects of the numbers.

The final window displays the output of the custom Sobel Y operator, focusing on vertical gradients and thus highlighting horizontal edges. This is evident from the enhanced visibility of the horizontal lines of the Sudoku grid and the tops and bottoms of the numbers. By pressing '4', the camera app successfully toggles to display these four windows, each labeled accordingly.

Part 3:

This part is about applying a few geometric transformations to a predetermined image, ‘Unity-Hall”.

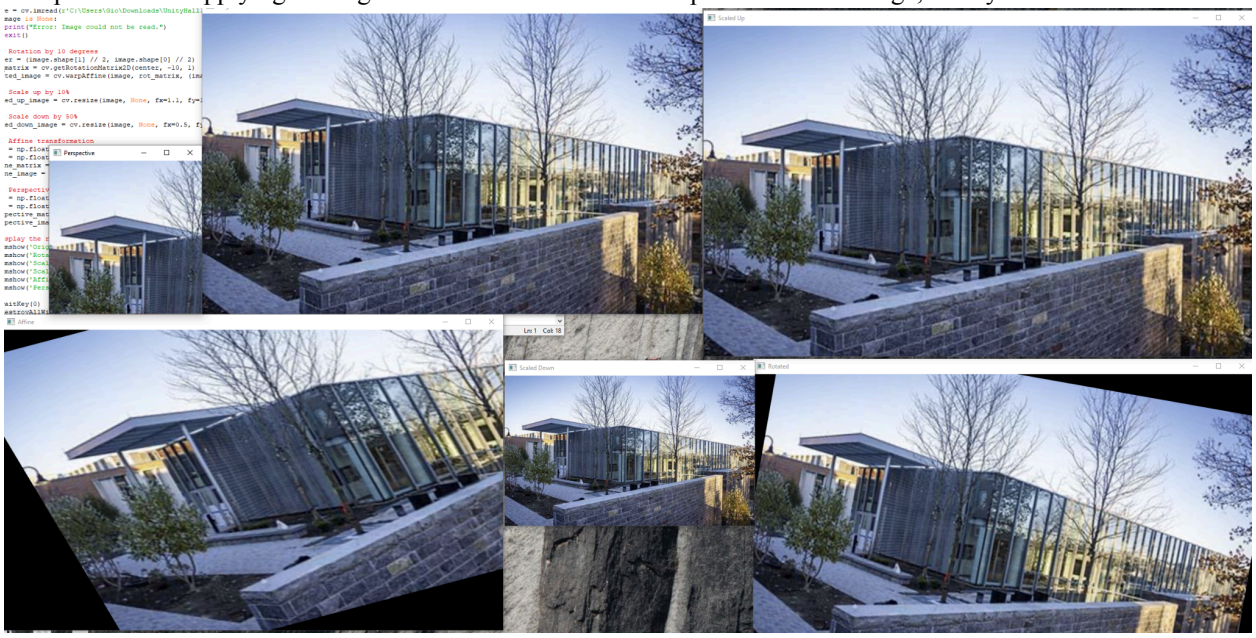

Figure 8: Original Image, Scale Up/Down, Perspective, Affine and Rotate.

The original, scaled-up, and scaled-down images are self-explanatory. The perspective transformation results in a zoomed-in effect because it enlarges a smaller part of the original image to fill a bigger area. This occurs when the designated new points are set closer together than the original points they correspond to, which causes the area they outline to appear larger.

The rotated and affine transformations change the image based on where specific points are placed. The affine transformation changes the image through a setup where three pairs of points dictate how the image will transform.

For example, moving a point from (50, 50) to (10, 100) shifts it leftward and downward. A point at (200, 50) slightly moves up but stays in line horizontally. Another point moves from (50, 200) to (100, 250), going rightward and downward. These shifts make the image seem as if it is rotating or tilting the other way, shaped by the movement of these points against the original layout of the image.

Part 4:

This segment of the code incorporates Harris Corner Detection and SIFT algorithms to the previous images. The primary goal is to show how these transformations impact the detection capabilities within an image.

The apply_harris function processes the image by converting it to grayscale before applying the Harris algorithm, which identifies corner points. These corners are then enhanced and distinctly marked in red to ensure visibility. Similarly, the apply_sift function converts the image to grayscale and employs the SIFT algorithm, known for its scale-invariance, to detect and highlight key points in green.

Once the transformations are applied, both Harris Corner Detection and SIFT are executed on each modified image. The output of these feature detection processes are displayed in separate subplots dedicated to each method. This visualization not only illustrates how geometric changes influence the detection results but also involves converting the images from BGR to RGB to maintain accurate color representation in the visual plots.

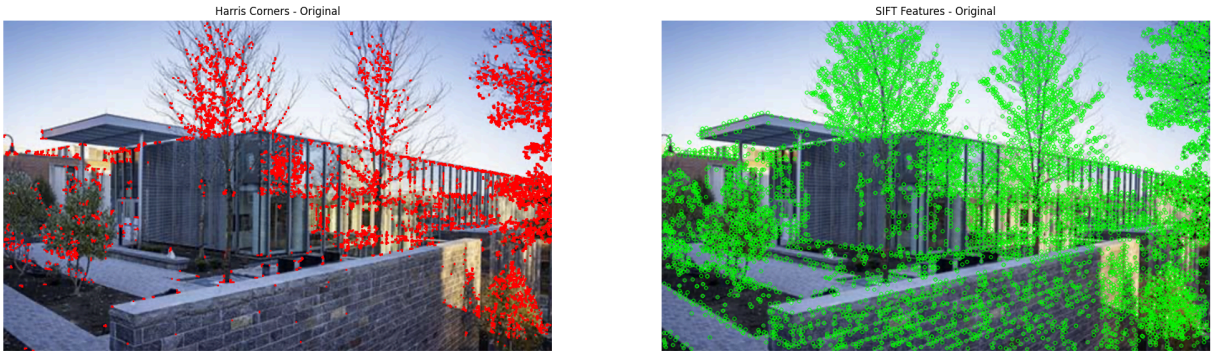

Figure 9: Original Images HCD & SIFT Algorithm Comparison

Harris Corner Detection focuses on identifying corners where the gradient of the image changes significantly in multiple directions. This algorithm is particularly sensitive to junctions where two edges meet, which typically result in high variations in image brightness.

In our context, the Harris algorithm emphasizes the areas in the trees because the patterns and textures within the foliage create many intersections where the gradient changes sharply; these are interpreted as corners by the Harris detector.

On the other hand, SIFT is designed to detect and describe local features in images. The algorithm looks for regions in the image where there are significant changes in all directions, which are robust to changes in scale, noise, and illumination. SIFT not only identifies corners but also more complex features such as blobs or textured areas.

Therefore, in our image, SIFT highlights a broader range of features, not just the corners but also edges, architectural elements, and other textures throughout the scene, including but not limited to the trees.

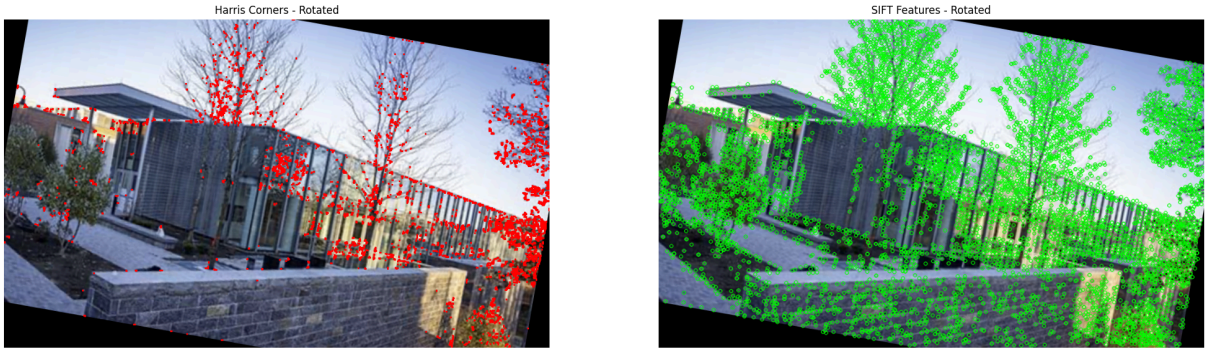

Figure 10: Rotated Images HCD & SIFT Algorithm Comparison

For the Rotate Image case described, the HCD algorithm displays slightly fewer dots compared to Figure 9, while SIFT exhibits a similar number of green dots as seen in the previous case. This observation suggests that the SIFT algorithm may be more resilient to transformations than Harris Corner Detection, indicating greater stability in feature detection under these conditions."

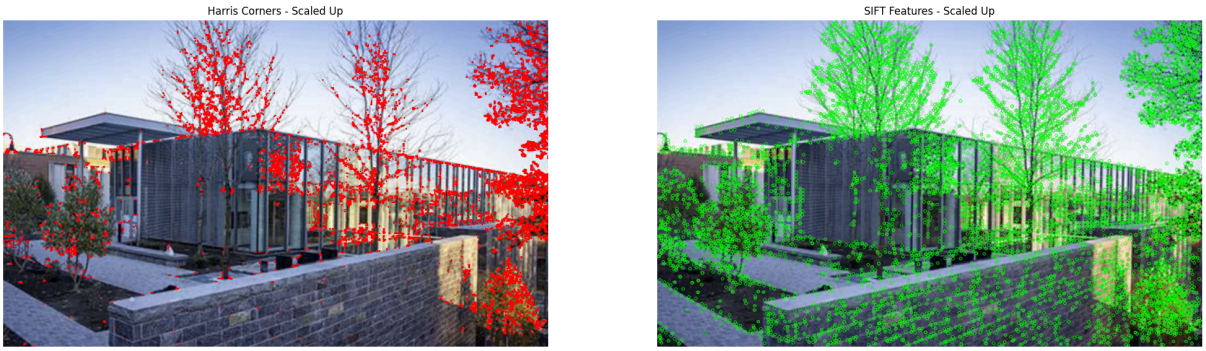

Figure 10: Scaled Up Images HCD & SIFT Algorithm Comparison

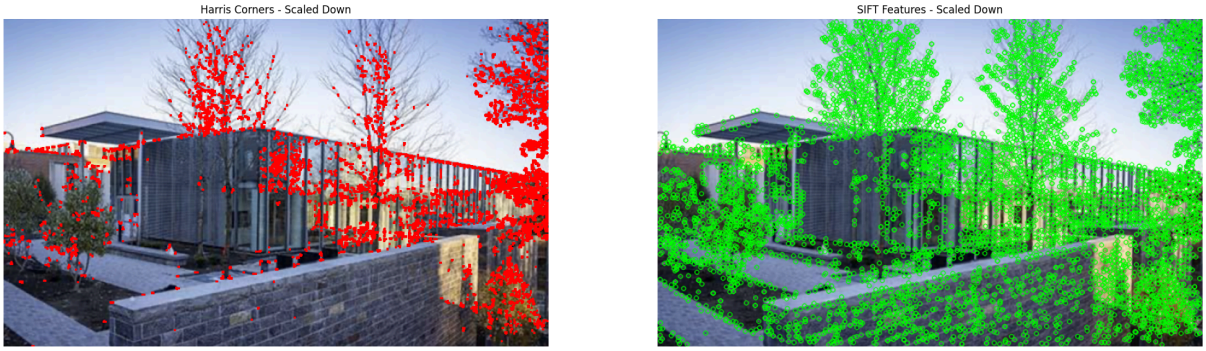

Figure 11: Scaled Down Images HCD & SIFT Algorithm Comparison

At first glance, the number of red and green dots seems similar for both the scaled-up and scaled-down images. However, upon closer inspection, the red dots in the scaled-up images appear more spread out, while in the scaled-down images, the density of both red and green dots increases. This is because the features are closer to each other in the smaller-scaled image, making them appear more concentrated.

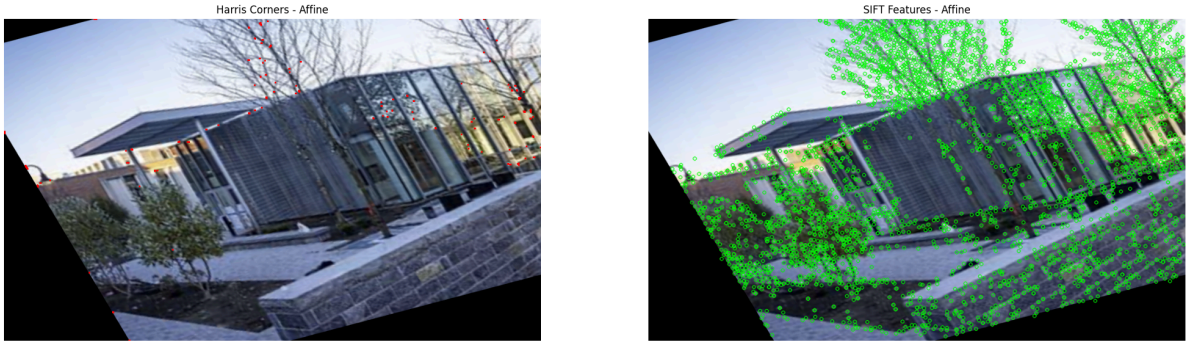

Figure 12: Affine Images HCD & SIFT Algorithm Comparison

The comparison of feature detection results after an affine transformation reveals some notable differences. In the HCD output, significantly fewer red dots are observed compared to previous cases, suggesting that HCD may be more sensitive to the effects of affine transformations, which include scaling, rotating, and shearing.

This sensitivity could lead to fewer corners being detected as the geometry of the image changes. On the other hand, the SIFT algorithm continues to perform consistently, although it too shows a slight decrease in the number of green

dots compared to the original image. This indicates that while SIFT is generally more robust to such transformations, it is not entirely unaffected.

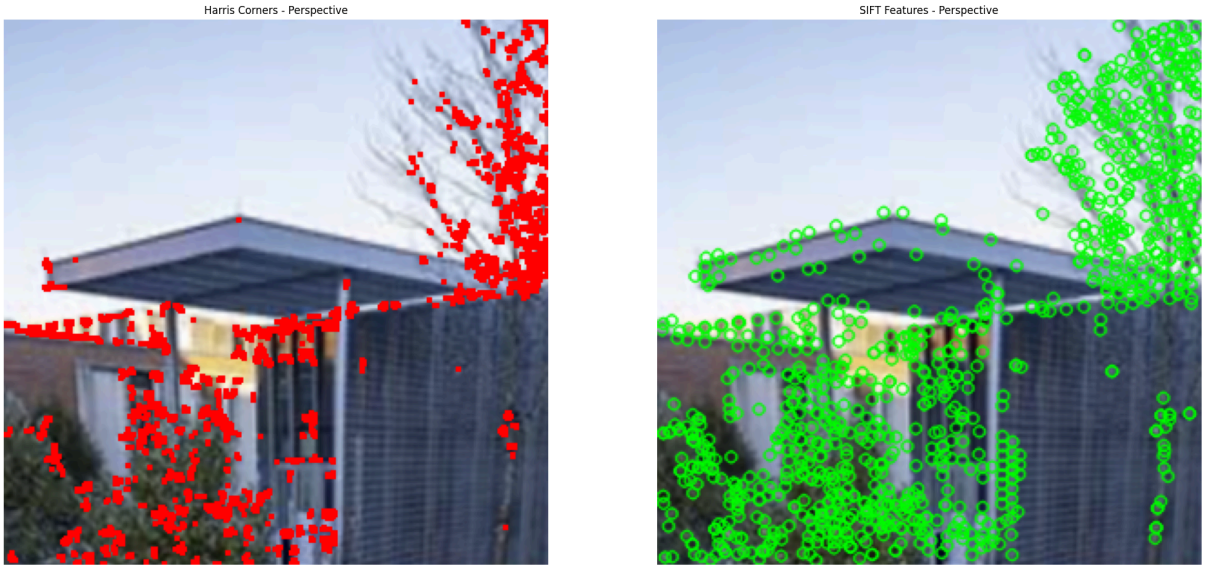

Figure 13: Perspective Images HCD & SIFT Algorithm Comparison

The last comparison involves images with a perspective transformation, which visually resemble a zoomed-in effect as shown above. In this case, both algorithms HCD and SIFT demonstrate similar effectiveness in detecting features within the trees and branches.

However, the SIFT algorithm distinguishes itself by identifying a greater number of features, marked by more green dots compared to the red dots from the HCD. Notably, SIFT also successfully captures additional features along the edges and corners of the building's roof, showcasing its comprehensive feature-tracking capability under the influence of perspective changes.