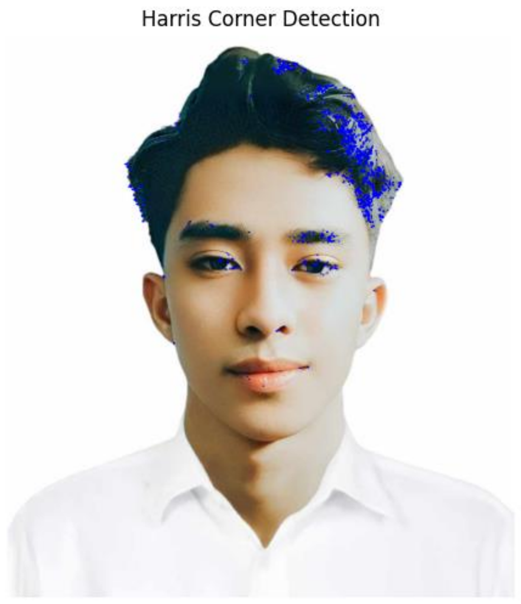
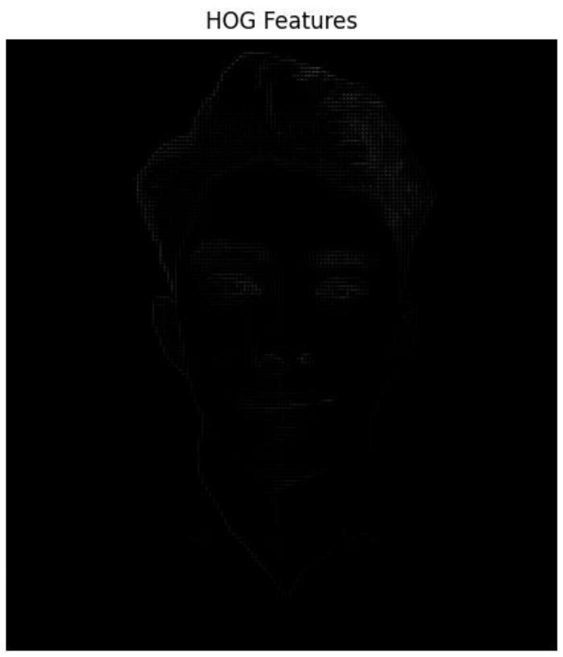
Performance Anaysis

Exercise 1: Harris Corner Detection



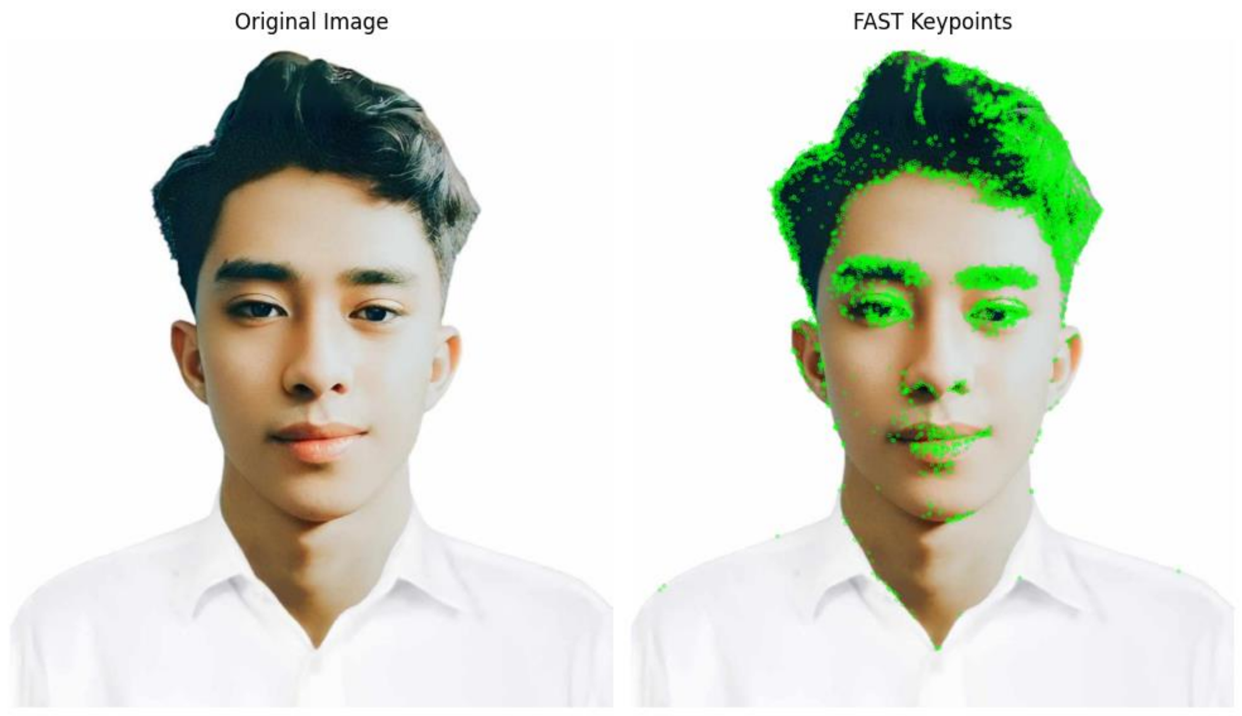
I use OpenCV to perform Harris Corner Detection on my image. I start by importing the necessary libraries: OpenCV, NumPy, and Matplotlib. Then, I load the image from a specified path and convert it to grayscale. I apply the cv2.cornerHarris() function to detect corners. After that, I dilate the corners to enhance their visibility and set a threshold to mark the significant corners in red. Finally, I display the original image with the detected corners highlighted using Matplotlib. It's a straightforward way to visualize the results of corner detection, and I can easily adapt it by changing the image path to use my own images.

Exercise 2: HOG (Histogram of Oriented Gradients) Feature Extraction



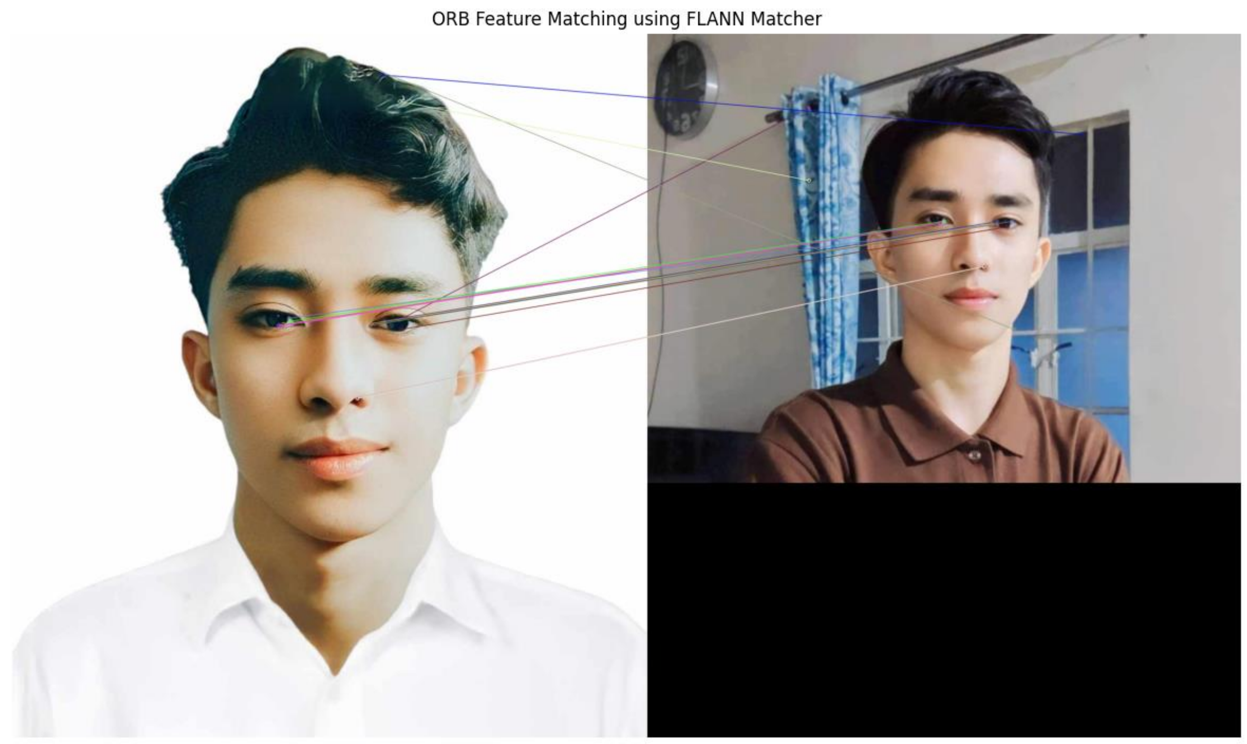
I utilize OpenCV and the skimage library to extract and visualize HOG (Histogram of Oriented Gradients) features from an image. I start by importing the necessary libraries: OpenCV, NumPy, Matplotlib, and relevant functions from skimage. I load an image from a specified path and convert it to grayscale, which is essential for HOG feature extraction. Using the hog() function, I extract the HOG features and also get a visual representation of these features. I configure parameters like orientations and cell sizes to tailor the HOG descriptor to my needs. To improve the visibility of the HOG image, I rescale its intensity. Finally, I display both the original image and the HOG visualization side by side using Matplotlib. This approach provides a clear view of how the image's gradients contribute to its features, making it easier to analyze the object's shapes and edges.

Exercise 3: FAST (Features from Accelerated Segment Test) Keypoint Detection



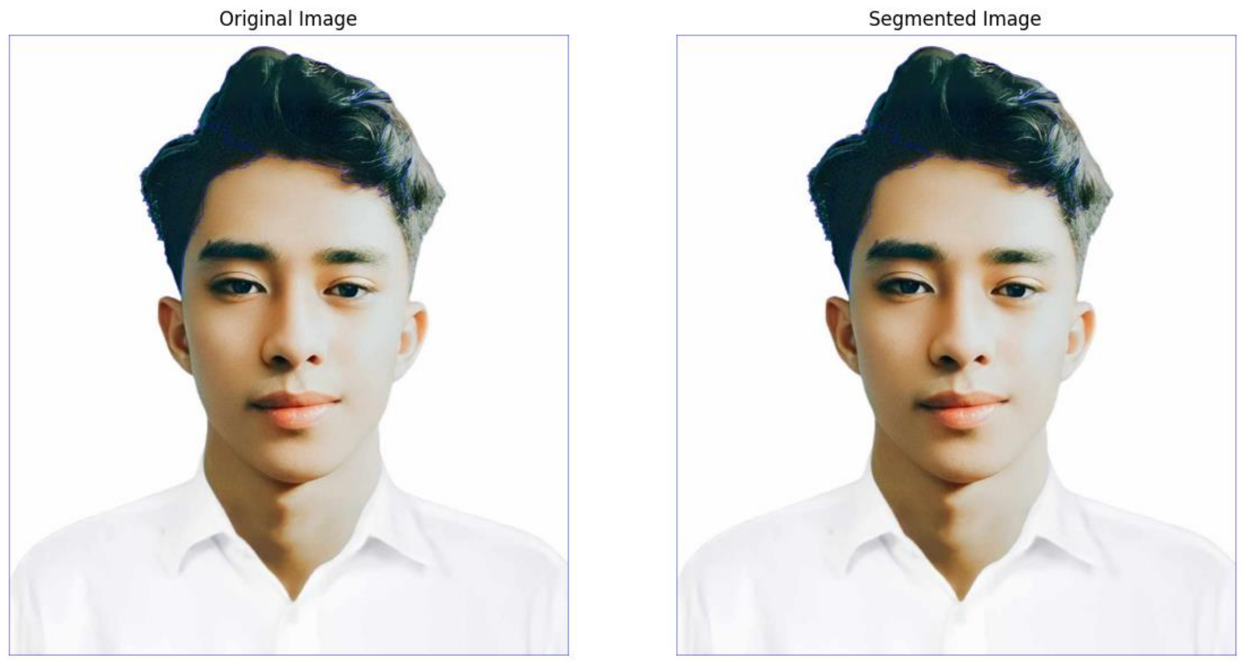
In this part of the exercise, I use OpenCV to perform keypoint detection using the FAST (Features from Accelerated Segment Test) algorithm. First, I import the necessary libraries, including OpenCV and Matplotlib. I load an image from a specified path and convert it to grayscale, which is essential for detecting keypoints. I then initialize the FAST detector and use it to identify keypoints in the grayscale image. After detecting the keypoints, I overlay them onto the original image, marking them in green for visibility. Finally, I display both the original image and the image with the detected keypoints side by side using Matplotlib. This visualization helps illustrate the key points in the image.

Exercise 4: Feature Matching using ORB and FLANN



I perform feature matching between two images using the ORB descriptor and the FLANN matcher. First, I load the two images and convert them to grayscale for better processing. Then, I initialize the ORB detector to identify keypoints and compute their descriptors in both images. Next, I configure the parameters for the FLANN matcher, specifying the algorithm and search settings. After initializing the matcher, I use the k-nearest neighbors (KNN) approach to find matches between the descriptors of the two images. To improve the matching quality, I apply Lowe's ratio test, keeping only the good matches where the distance of the closest match is significantly lower than that of the second closest. Finally, I visualize the matched keypoints by drawing lines between them on a combined image of both original images. This visualization helps demonstrate how well the keypoints from the two images correspond to each other, which is useful in various computer vision applications like object recognition and image stitching.

Exercise 5: Image Segmentation using Watershed Algorithm



In the last part of the exercise, I perform image segmentation using the Watershed algorithm. First, I load an image and convert it to grayscale to simplify the processing. I then apply a binary threshold to create a binary image where the foreground and background are clearly separated. To enhance the segmentation, I use morphological operations, specifically an opening operation, to remove noise from the thresholded image. Next, I identify the sure background by dilating the opened image and use the distance transform to find the sure foreground areas. By applying a threshold on the distance transform, I can distinguish between sure and unknown regions. After marking the sure foreground, I label the connected components in the sure foreground and adjust the markers to ensure the background is labeled appropriately. I also mark the unknown regions as zero. Finally, I apply the Watershed algorithm to the image, which segments it by treating the markers as basins and the boundaries as ridges. I visualize the result by marking the segmented boundaries in red and displaying both the original and segmented images side by side.