Republic of the Philippines
Laguna State Polytechnic University
Province of Laguna

Machine Problem No. 4			
Topic:	**Module 2.0: Feature Extraction and Object Detection**	**Week No.**	6-7
Course Code:	**CSST106**	**Term:**	1st Semester
Course Title:	**Perception and Computer Vision**	**Academic Year:**	2024-2025
Student Name		**Section**	
Due date		**Points**	

Machine Problem No. 4: Feature Extraction and Image Matching in Computer Vision

Overview:

In this machine problem, you will implement various feature extraction and matching algorithms to process, analyze, and compare different images. You will utilize techniques such as SIFT, SURF, ORB, HOG, and Harris Corner Detection to extract keypoints and descriptors. You will also perform feature matching between pairs of images using the FLANN and Brute-Force matchers. Finally, you will explore image segmentation using the Watershed algorithm.

Objectives:

1. To apply different feature extraction methods (SIFT, SURF, ORB, HOG, Harris Corner Detection).
2. To perform feature matching using Brute-Force and FLANN matchers.
3. To implement the Watershed algorithm for image segmentation.
4. To visualize and analyze keypoints and matches between images.
5. To evaluate the performance of different feature extraction methods on different images.

Problem Statement:

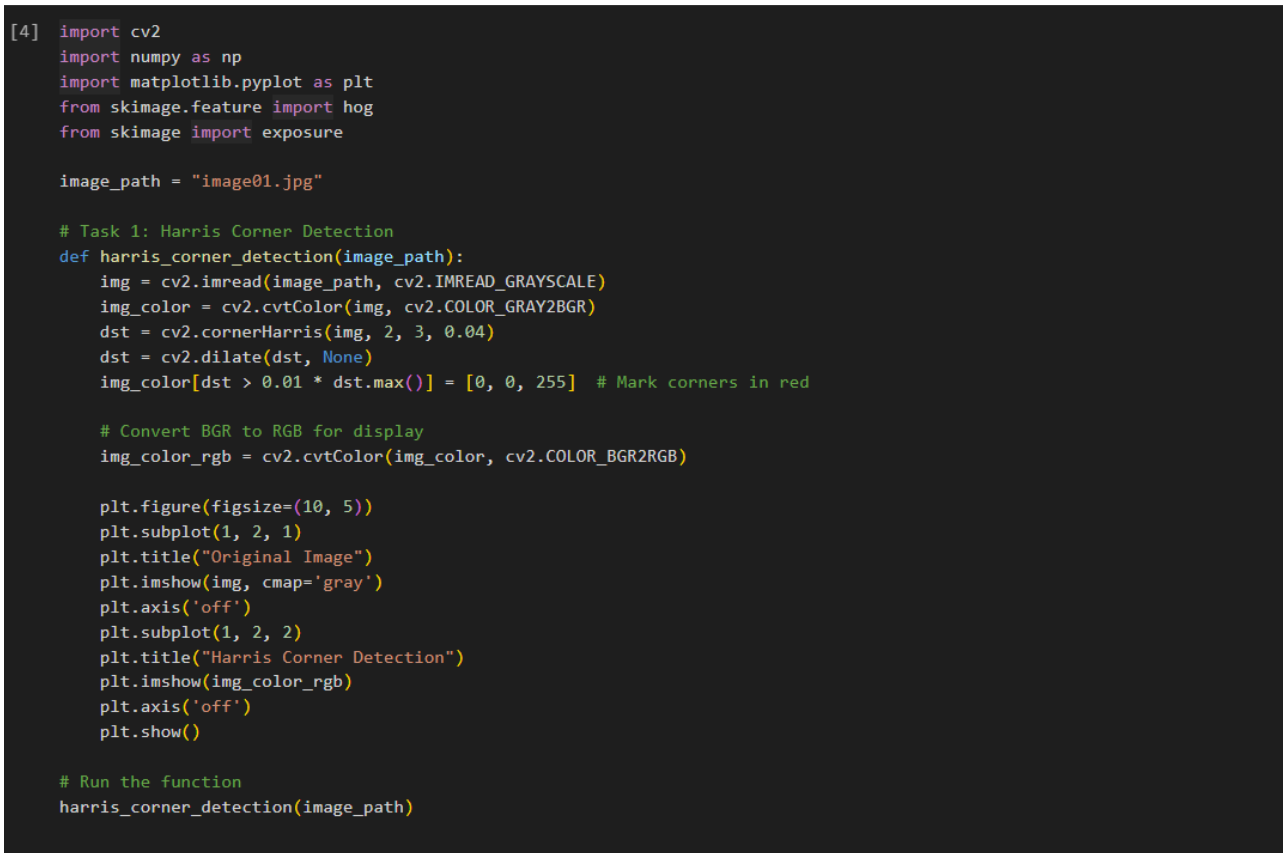
You are tasked with building a Python program using OpenCV and related libraries to accomplish the following tasks. Each task should be implemented in a separate function, with appropriate comments and visual outputs. You will work with images provided in your directory or chosen from any online source.

Tasks:

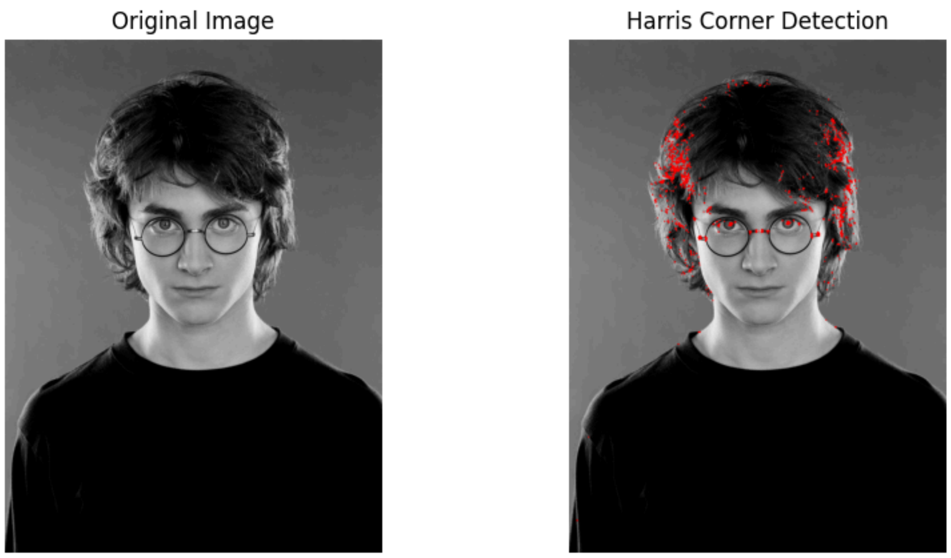
Task 1: Harris Corner Detection

- Load any grayscale image.
- Apply the Harris Corner Detection algorithm to find corners.
- Display the original image and the image with detected corners marked in red.

Function signature: def harris_corner_detection(image_path):



```
[4]  import cv2
     import numpy as np
     import matplotlib.pyplot as plt
     from skimage.feature import hog
     from skimage import exposure

     image_path = "image01.jpg"

     # Task 1: Harris Corner Detection
     def harris_corner_detection(image_path):
         img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
         img_color = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
         dst = cv2.cornerHarris(img, 2, 3, 0.04)
         dst = cv2.dilate(dst, None)
         img_color[dst > 0.01 * dst.max()] = [0, 0, 255]  # Mark corners in red

         # Convert BGR to RGB for display
         img_color_rgb = cv2.cvtColor(img_color, cv2.COLOR_BGR2RGB)

         plt.figure(figsize=(10, 5))
         plt.subplot(1, 2, 1)
         plt.title("Original Image")
         plt.imshow(img, cmap='gray')
         plt.axis('off')
         plt.subplot(1, 2, 2)
         plt.title("Harris Corner Detection")
         plt.imshow(img_color_rgb)
         plt.axis('off')
         plt.show()

     # Run the function
     harris_corner_detection(image_path)
```

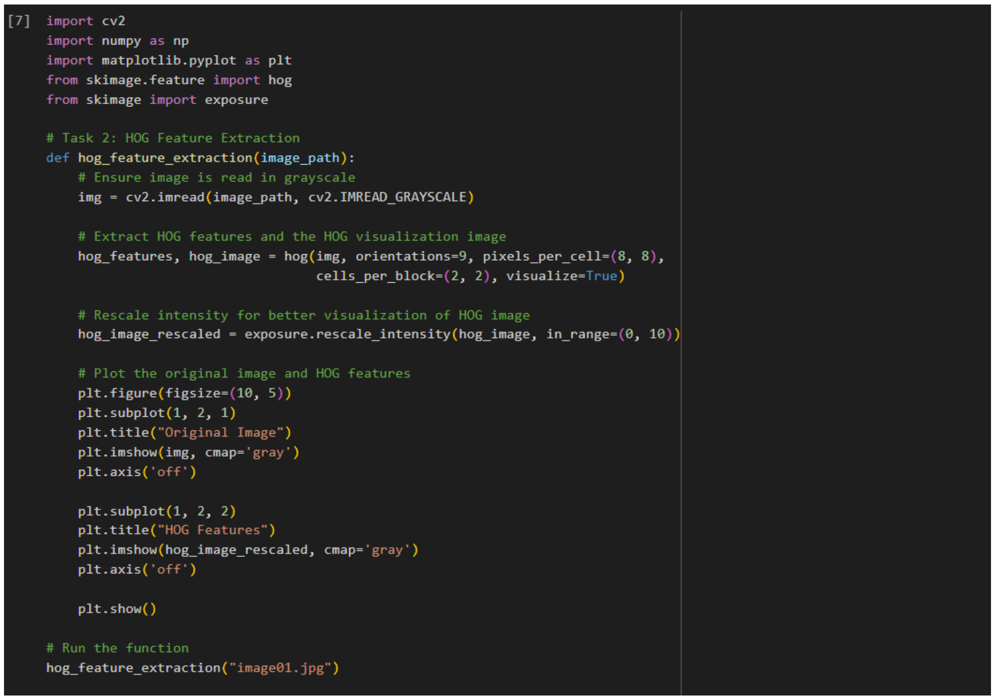


➔ The `harris_corner_detection` function detects corners in an image using the Harris Corner Detection algorithm, a popular method for identifying distinct points in an image. It first loads the image in grayscale to simplify processing, then applies the Harris corner detector, which computes regions in the image with high intensity variation, indicating potential corners. To make the corners stand out, the function dilates the detected corners and marks them in red on a color version of the image. Finally, the function displays the original grayscale image alongside the result, where corners are highlighted in red, making it easy to see areas of interest. This visualization can be helpful in computer vision tasks where corner points are important for tasks like image matching, object detection, and scene understanding.

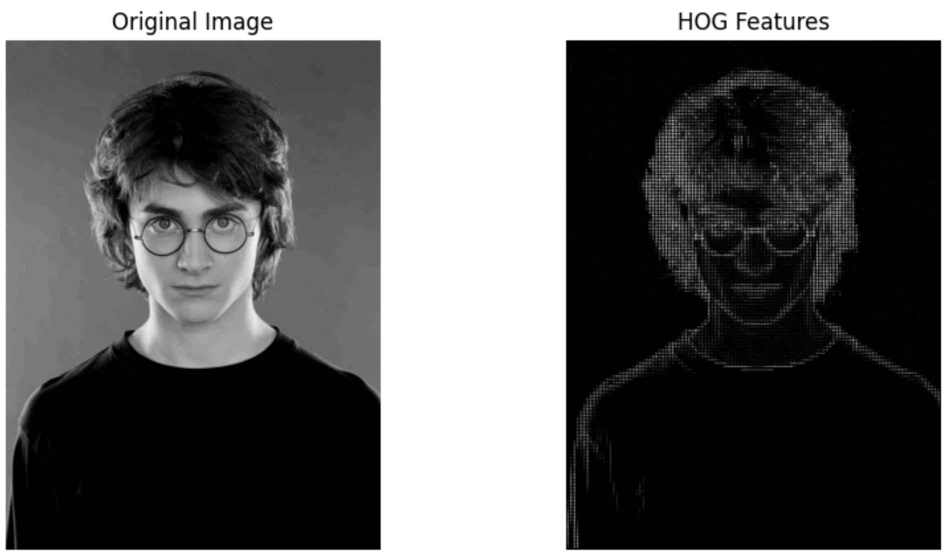
Task 2: HOG Feature Extraction

- Load an image of a person (or another object).
- Convert the image to grayscale.
- Extract HOG (Histogram of Oriented Gradients) features from the image.
- Display the original image and the visualized HOG features.

Function signature: def hog_feature_extraction(image_path):



```
[7] import cv2
    import numpy as np
    import matplotlib.pyplot as plt
    from skimage.feature import hog
    from skimage import exposure

    # Task 2: HOG Feature Extraction
    def hog_feature_extraction(image_path):
        # Ensure image is read in grayscale
        img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

        # Extract HOG features and the HOG visualization image
        hog_features, hog_image = hog(img, orientations=9, pixels_per_cell=(8, 8),
                                      cells_per_block=(2, 2), visualize=True)

        # Rescale intensity for better visualization of HOG image
        hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

        # Plot the original image and HOG features
        plt.figure(figsize=(10, 5))
        plt.subplot(1, 2, 1)
        plt.title("Original Image")
        plt.imshow(img, cmap='gray')
        plt.axis('off')

        plt.subplot(1, 2, 2)
        plt.title("HOG Features")
        plt.imshow(hog_image_rescaled, cmap='gray')
        plt.axis('off')

        plt.show()

    # Run the function
    hog_feature_extraction("image01.jpg")
```

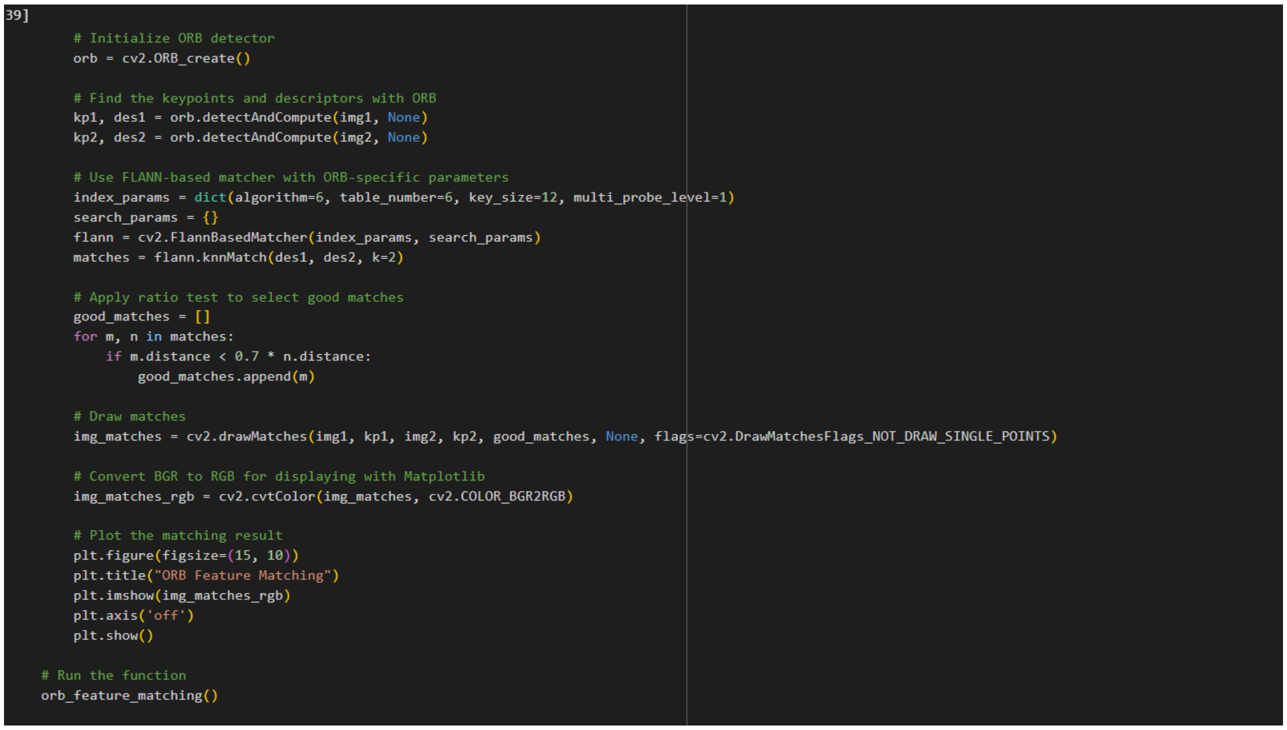


➔ The `hog_feature_extraction` function extracts Histogram of Oriented Gradients (HOG) features from an image, which are useful for capturing edge directions and textures in a way that aids in recognizing objects and shapes. The function first loads the image in grayscale to simplify feature extraction. Using the `hog` function, it calculates the HOG features as well as a visualization of the HOG image, which displays gradients and edges in the image. The visualization is enhanced by rescaling the intensity, making the gradient patterns more distinct. Finally, it displays both the original grayscale image and the HOG feature visualization side by side. This side-by-side display helps illustrate how HOG features capture structural information, often used in object detection and recognition tasks, especially with humans and vehicles.

Task 3: ORB Feature Extraction and Matching

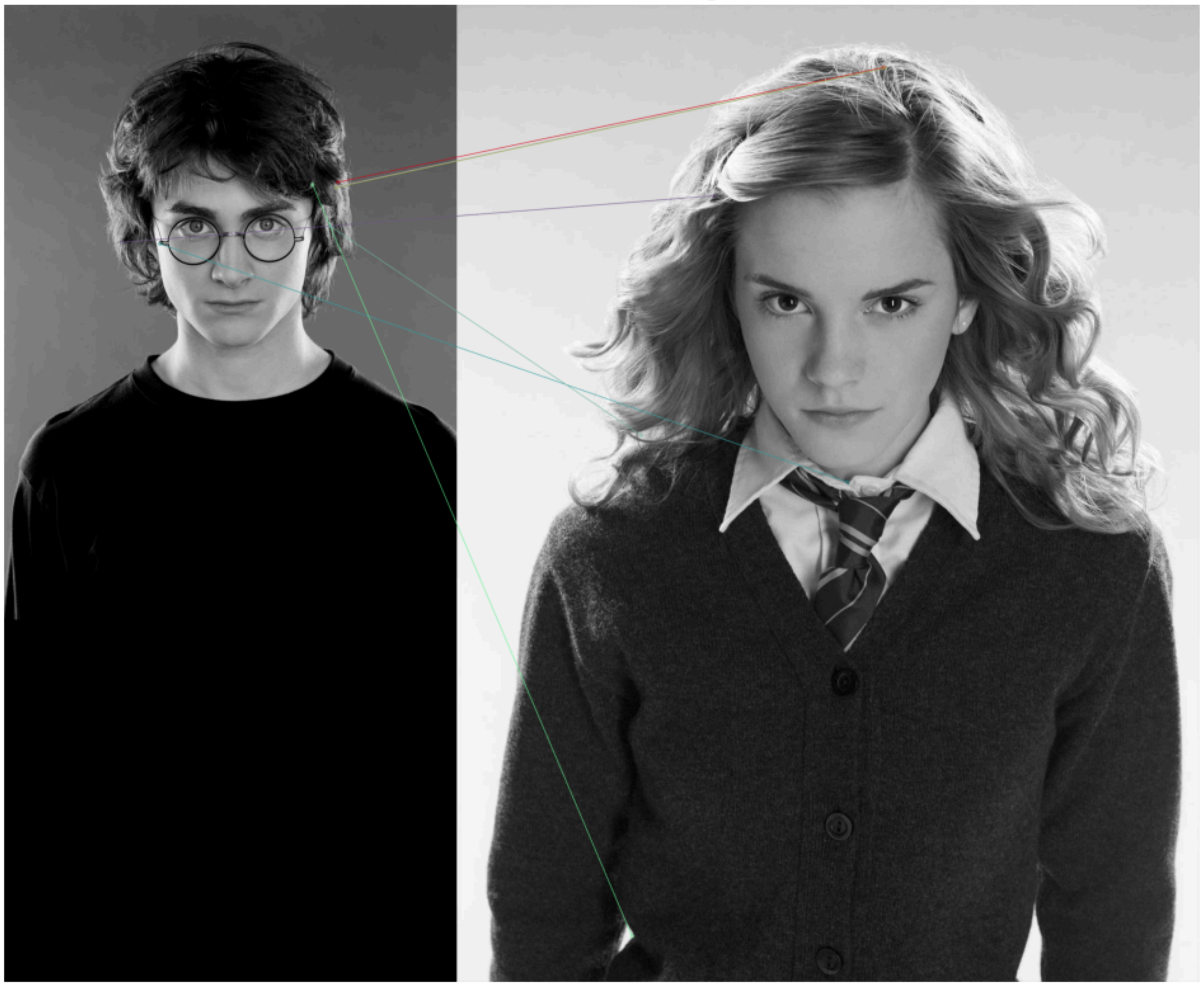
- Load two different images.
- Apply ORB (Oriented FAST and Rotated BRIEF) to detect and compute keypoints and descriptors for both images.
- Use the FLANN-based matcher to match the ORB descriptors of the two images.
- Visualize the matching keypoints between the two images.

Function signature: def orb_feature_matching(image_path1, image_path2):



```
39]
        # Initialize ORB detector
        orb = cv2.ORB_create()

        # Find the keypoints and descriptors with ORB
        kp1, des1 = orb.detectAndCompute(img1, None)
        kp2, des2 = orb.detectAndCompute(img2, None)

        # Use FLANN-based matcher with ORB-specific parameters
        index_params = dict(algorithm=6, table_number=6, key_size=12, multi_probe_level=1)
        search_params = {}
        flann = cv2.FlannBasedMatcher(index_params, search_params)
        matches = flann.knnMatch(des1, des2, k=2)

        # Apply ratio test to select good matches
        good_matches = []
        for m, n in matches:
            if m.distance < 0.7 * n.distance:
                good_matches.append(m)

        # Draw matches
        img_matches = cv2.drawMatches(img1, kp1, img2, kp2, good_matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

        # Convert BGR to RGB for displaying with Matplotlib
        img_matches_rgb = cv2.cvtColor(img_matches, cv2.COLOR_BGR2RGB)

        # Plot the matching result
        plt.figure(figsize=(15, 10))
        plt.title("ORB Feature Matching")
        plt.imshow(img_matches_rgb)
        plt.axis('off')
        plt.show()

    # Run the function
    orb_feature_matching()
```

ORB Feature Matching



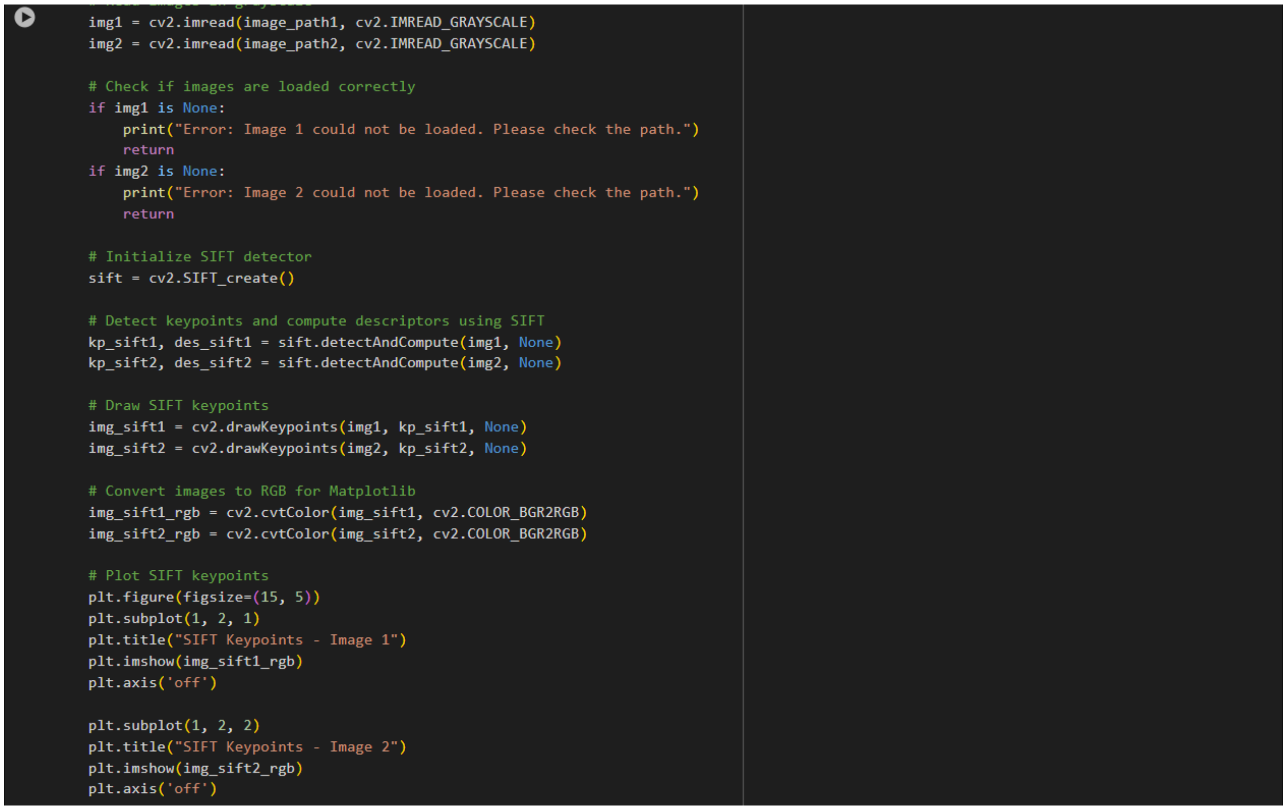
➔ The orb_feature_matching function performs feature extraction and matching between two images using the ORB (Oriented FAST and Rotated BRIEF) algorithm, which is well-suited for identifying and matching keypoints in images. The function starts by loading two images in grayscale, checks if they were loaded successfully, and then initializes the ORB detector to identify keypoints and compute descriptors.

➔ To find matches between the two sets of descriptors, the function uses a FLANN-based matcher with ORB-specific parameters to perform a nearest-neighbor search. It applies a ratio test to filter out poor matches, retaining only the "good matches" that meet a specified distance threshold. Finally, it displays the two images side by side with lines drawn between the matched keypoints, offering a visual representation of feature similarity between the images. This process is commonly used in applications like image stitching, object recognition, and scene reconstruction.

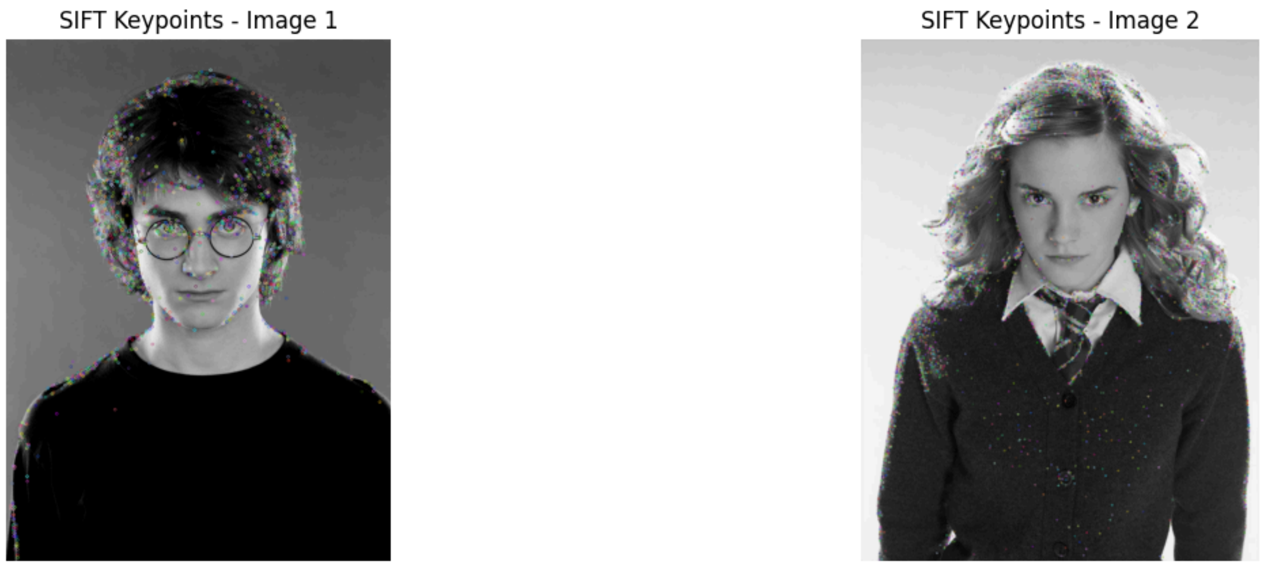
Task 4: SIFT and SURF Feature Extraction

- Load two images of your choice.
- Apply both SIFT and SURF algorithms to detect keypoints and compute descriptors.
- Visualize the keypoints detected by both methods in two separate images.

Function signature: def sift_and_surf_feature_extraction(image_path1, image_path2):



```
    img1 = cv2.imread(image_path1, cv2.IMREAD_GRAYSCALE)
    img2 = cv2.imread(image_path2, cv2.IMREAD_GRAYSCALE)

    # Check if images are loaded correctly
    if img1 is None:
        print("Error: Image 1 could not be loaded. Please check the path.")
        return
    if img2 is None:
        print("Error: Image 2 could not be loaded. Please check the path.")
        return

    # Initialize SIFT detector
    sift = cv2.SIFT_create()

    # Detect keypoints and compute descriptors using SIFT
    kp_sift1, des_sift1 = sift.detectAndCompute(img1, None)
    kp_sift2, des_sift2 = sift.detectAndCompute(img2, None)

    # Draw SIFT keypoints
    img_sift1 = cv2.drawKeypoints(img1, kp_sift1, None)
    img_sift2 = cv2.drawKeypoints(img2, kp_sift2, None)

    # Convert images to RGB for Matplotlib
    img_sift1_rgb = cv2.cvtColor(img_sift1, cv2.COLOR_BGR2RGB)
    img_sift2_rgb = cv2.cvtColor(img_sift2, cv2.COLOR_BGR2RGB)

    # Plot SIFT keypoints
    plt.figure(figsize=(15, 5))
    plt.subplot(1, 2, 1)
    plt.title("SIFT Keypoints - Image 1")
    plt.imshow(img_sift1_rgb)
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.title("SIFT Keypoints - Image 2")
    plt.imshow(img_sift2_rgb)
    plt.axis('off')
```



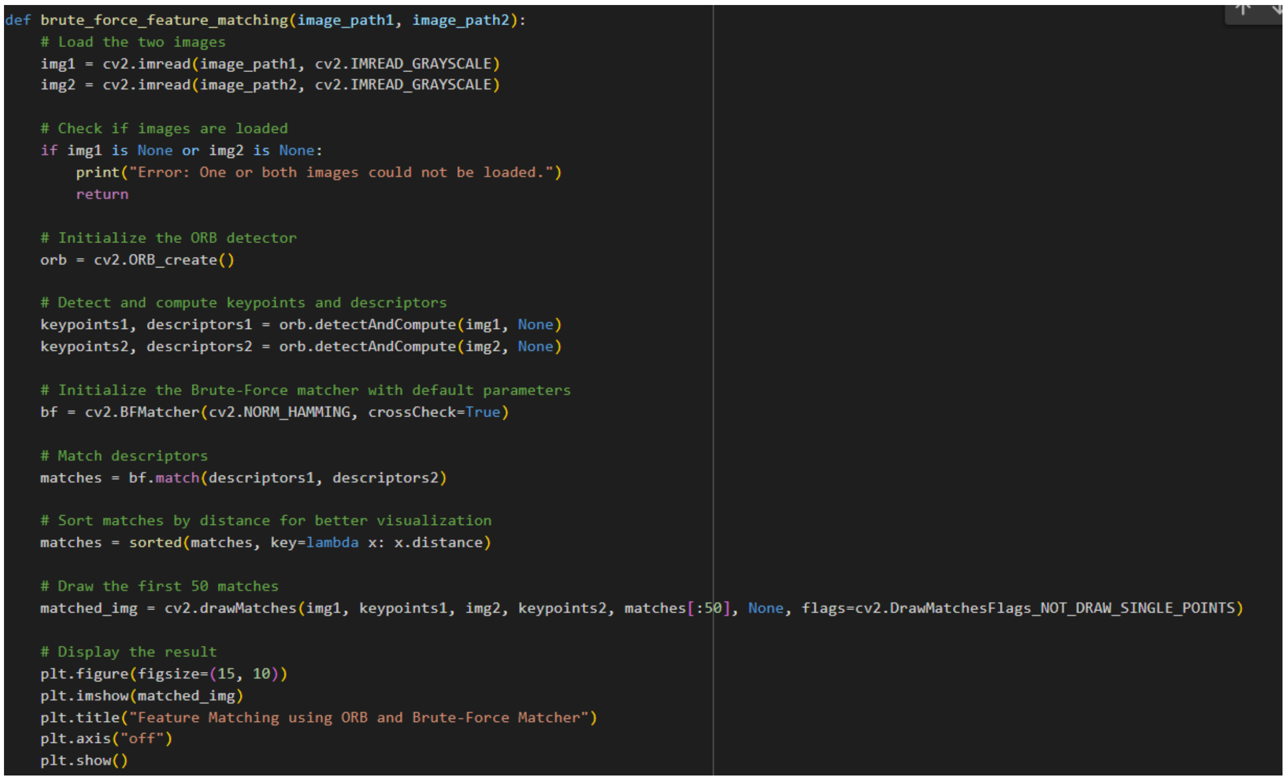
➔ The sift_feature_extraction function extracts and visualizes keypoints from two images using the SIFT (Scale-Invariant Feature Transform) algorithm, which is commonly used in computer vision for detecting distinct, scale-invariant features. The function starts by loading the images in grayscale to focus on structural features rather than color. It then initializes the SIFT detector, which identifies keypoints and computes descriptors for both images. These descriptors are unique representations of local features, making them useful for tasks like image matching and object recognition.

➔ After detecting keypoints, the function uses cv2.drawKeypoints to overlay them on each image, helping to visualize the detected features. Finally, it displays both images with their keypoints side by side, allowing a comparison of the distinct regions SIFT identified in each image. This feature detection process is often used in applications that require image alignment or recognition under different scales or lighting conditions.

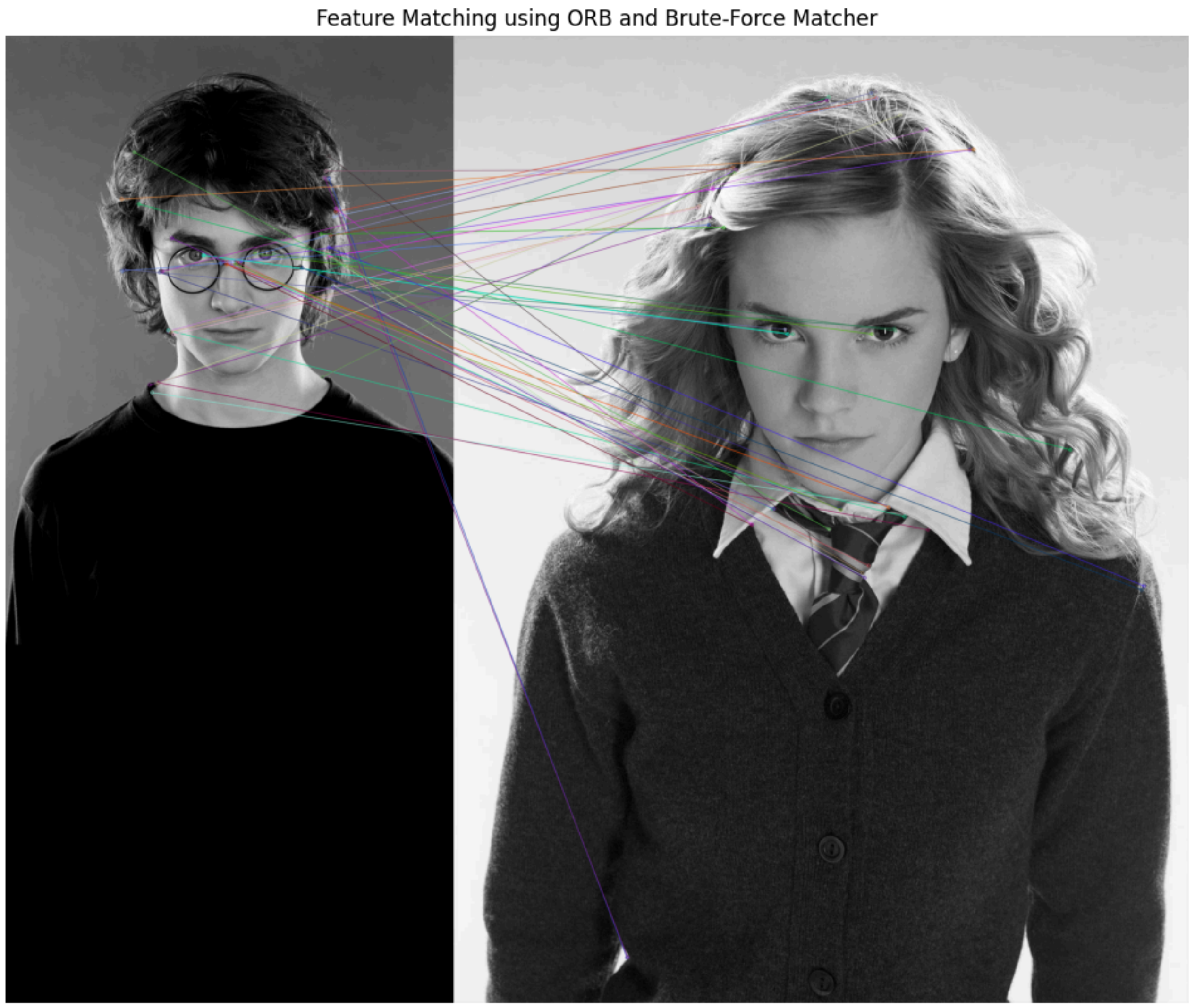
Task 5: Feature Matching using Brute-Force Matcher

- Load two images and extract ORB descriptors.
- Match the descriptors using the Brute-Force Matcher.
- Display the matched keypoints between the two images.

Function signature: def brute_force_feature_matching(image_path1, image_path2):



```python
def brute_force_feature_matching(image_path1, image_path2):
    # Load the two images
    img1 = cv2.imread(image_path1, cv2.IMREAD_GRAYSCALE)
    img2 = cv2.imread(image_path2, cv2.IMREAD_GRAYSCALE)

    # Check if images are loaded
    if img1 is None or img2 is None:
        print("Error: One or both images could not be loaded.")
        return

    # Initialize the ORB detector
    orb = cv2.ORB_create()

    # Detect and compute keypoints and descriptors
    keypoints1, descriptors1 = orb.detectAndCompute(img1, None)
    keypoints2, descriptors2 = orb.detectAndCompute(img2, None)

    # Initialize the Brute-Force matcher with default parameters
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

    # Match descriptors
    matches = bf.match(descriptors1, descriptors2)

    # Sort matches by distance for better visualization
    matches = sorted(matches, key=lambda x: x.distance)

    # Draw the first 50 matches
    matched_img = cv2.drawMatches(img1, keypoints1, img2, keypoints2, matches[:50], None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

    # Display the result
    plt.figure(figsize=(15, 10))
    plt.imshow(matched_img)
    plt.title("Feature Matching using ORB and Brute-Force Matcher")
    plt.axis("off")
    plt.show()
```



➔ The `brute_force_feature_matching` function matches keypoints between two images using the ORB (Oriented FAST and Rotated BRIEF) detector and a Brute-Force matcher. The function begins by loading the two images in grayscale, simplifying the process by focusing only on intensity values. It then initializes ORB to detect keypoints—distinct points in the images—and compute descriptors, which are unique numeric representations of each keypoint, useful for matching.

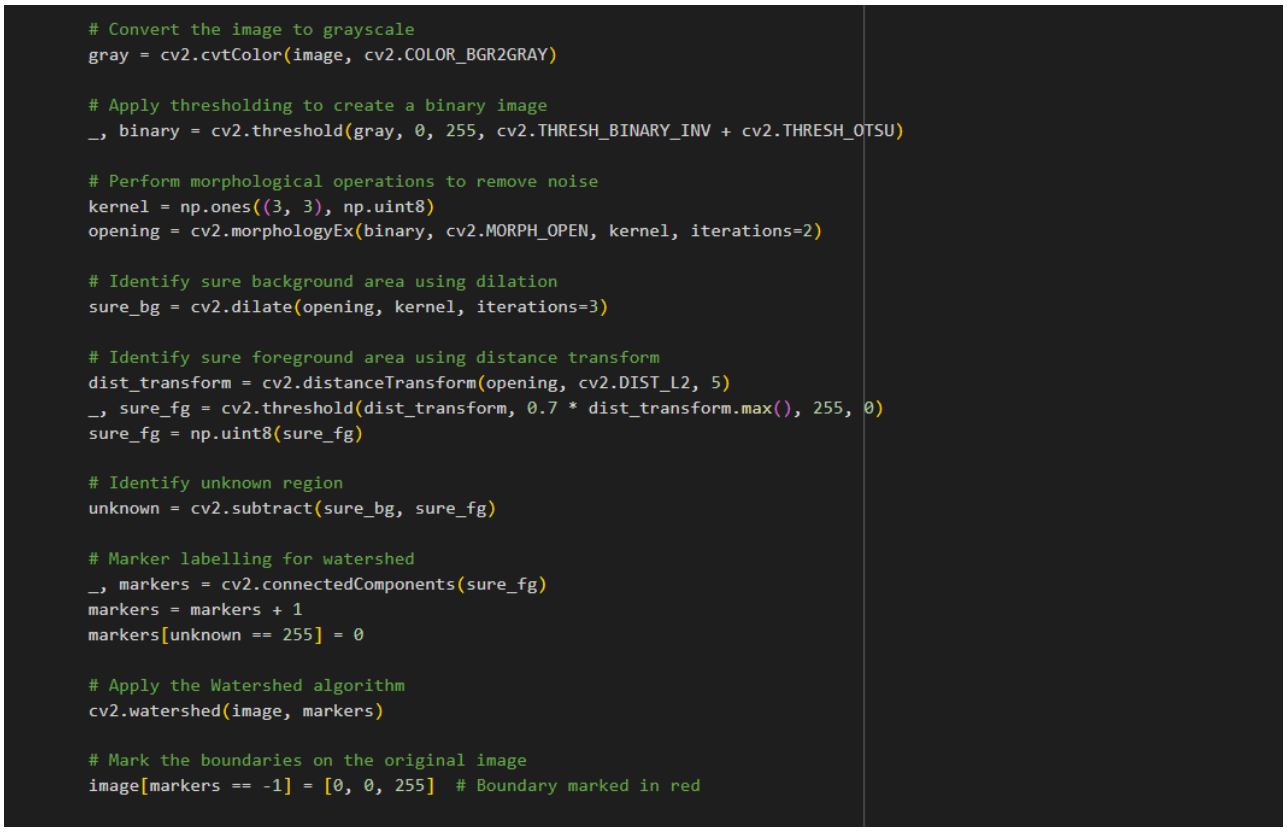
➔ Once the descriptors are obtained, the function uses a Brute-Force matcher with Hamming distance to compare the descriptors from both images. It sorts these matches by distance, which ranks them by how similar they are. Finally, it visualizes the top 50 matches by drawing lines between matched keypoints in both images, providing an intuitive display of similarities. This kind of matching is often useful in applications such as object recognition, image stitching, and 3D reconstruction.

Task 6: Image Segmentation using Watershed Algorithm

- Load any image of your choice.
- Convert the image to grayscale and apply a threshold to separate foreground from the background.

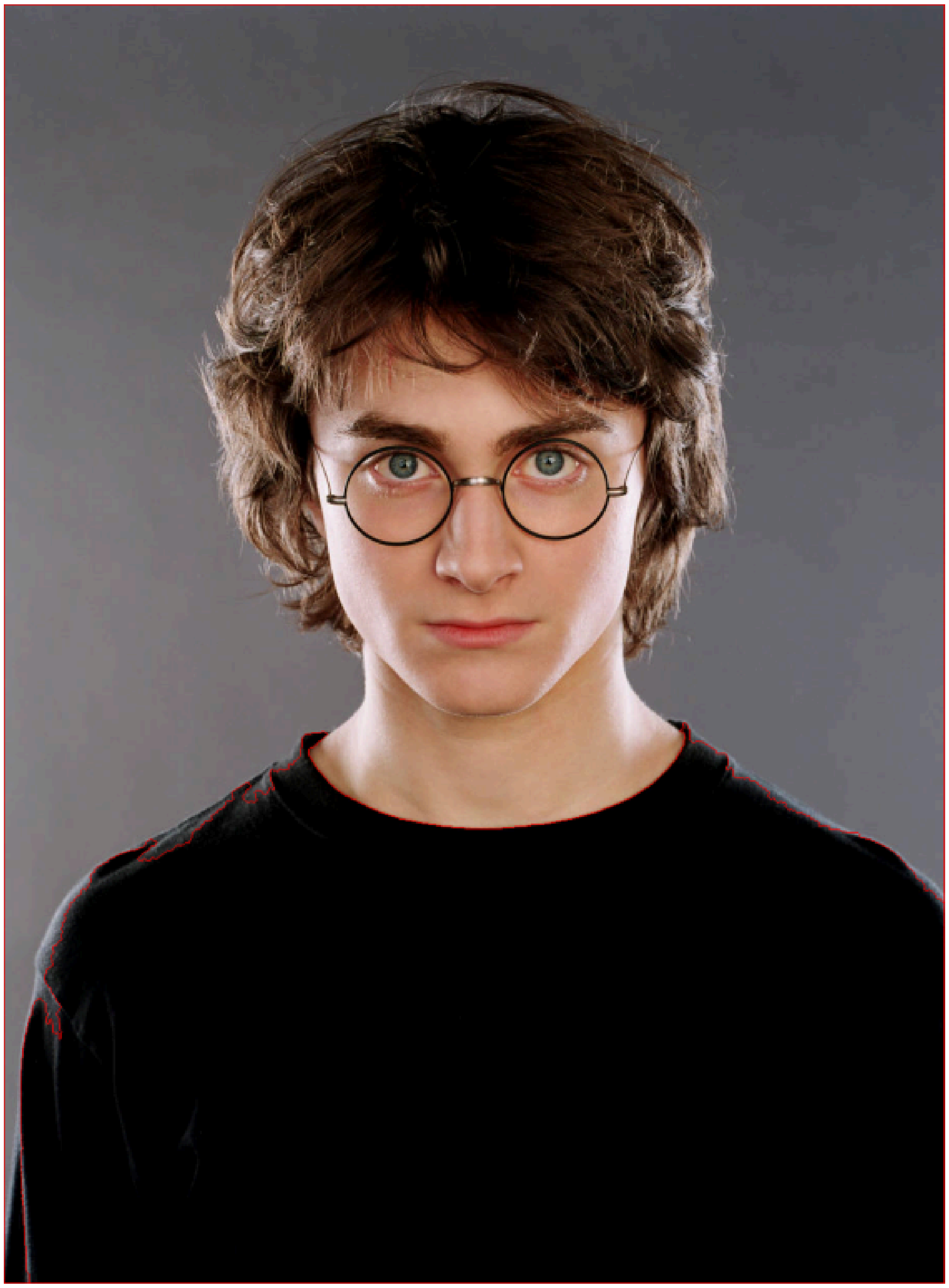
- Apply the Watershed algorithm to segment the image into distinct regions.
- Display the segmented image.

Function signature: def watershed_segmentation(image_path):



```
# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply thresholding to create a binary image
_, binary = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

# Perform morphological operations to remove noise
kernel = np.ones((3, 3), np.uint8)
opening = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel, iterations=2)

# Identify sure background area using dilation
sure_bg = cv2.dilate(opening, kernel, iterations=3)

# Identify sure foreground area using distance transform
dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
_, sure_fg = cv2.threshold(dist_transform, 0.7 * dist_transform.max(), 255, 0)
sure_fg = np.uint8(sure_fg)

# Identify unknown region
unknown = cv2.subtract(sure_bg, sure_fg)

# Marker labelling for watershed
_, markers = cv2.connectedComponents(sure_fg)
markers = markers + 1
markers[unknown == 255] = 0

# Apply the Watershed algorithm
cv2.watershed(image, markers)

# Mark the boundaries on the original image
image[markers == -1] = [0, 0, 255]  # Boundary marked in red
```

Segmented Image with Watershed



- ➔ The `watershed_segmentation` function segments an image using the Watershed algorithm, which is commonly used to distinguish overlapping objects or regions with unclear boundaries. It starts by loading and converting the image to grayscale, followed by thresholding to create a binary image that separates objects from the background. Next, the function performs morphological operations to remove noise and uses dilation to define a clear "sure background" area.

- ➔ To separate foreground objects, it applies a distance transform and identifies regions that are confidently in the foreground. The difference between the foreground and background regions is used to mark "unknown" areas, which could contain boundary regions between objects. Finally, the function labels markers for each identified region and applies the Watershed algorithm. This marks the boundaries of each segmented region in red on the original image, making it easy to see the separation between objects or areas within the image. The result is displayed, showing distinct regions clearly outlined, which is useful in applications like medical imaging, object detection, and more.

Instructions:

1. **Input**:
   - o You will be provided with two or more images for each task, or you can select images of your own.
   - o Each image should be loaded, processed, and then displayed with appropriate markings (e.g., keypoints, segments, or matches).
2. **Output**:
   - o Display the original images and results for each task (e.g., corner detection, feature extraction, matching, segmentation).
   - o Ensure that each result is appropriately labeled (title and axis for the plot if needed).
3. **Implementation Guidelines**:
   - o Implement each task as a separate function in Python.
   - o Use OpenCV for image processing and feature extraction tasks.
   - o Use Matplotlib to visualize and display the results.
   - o Include proper comments to explain your code.

Submission:

- A PDF or markdown document (performance_analysis.pdf or performance_analysis.md).

Submission Guidelines:

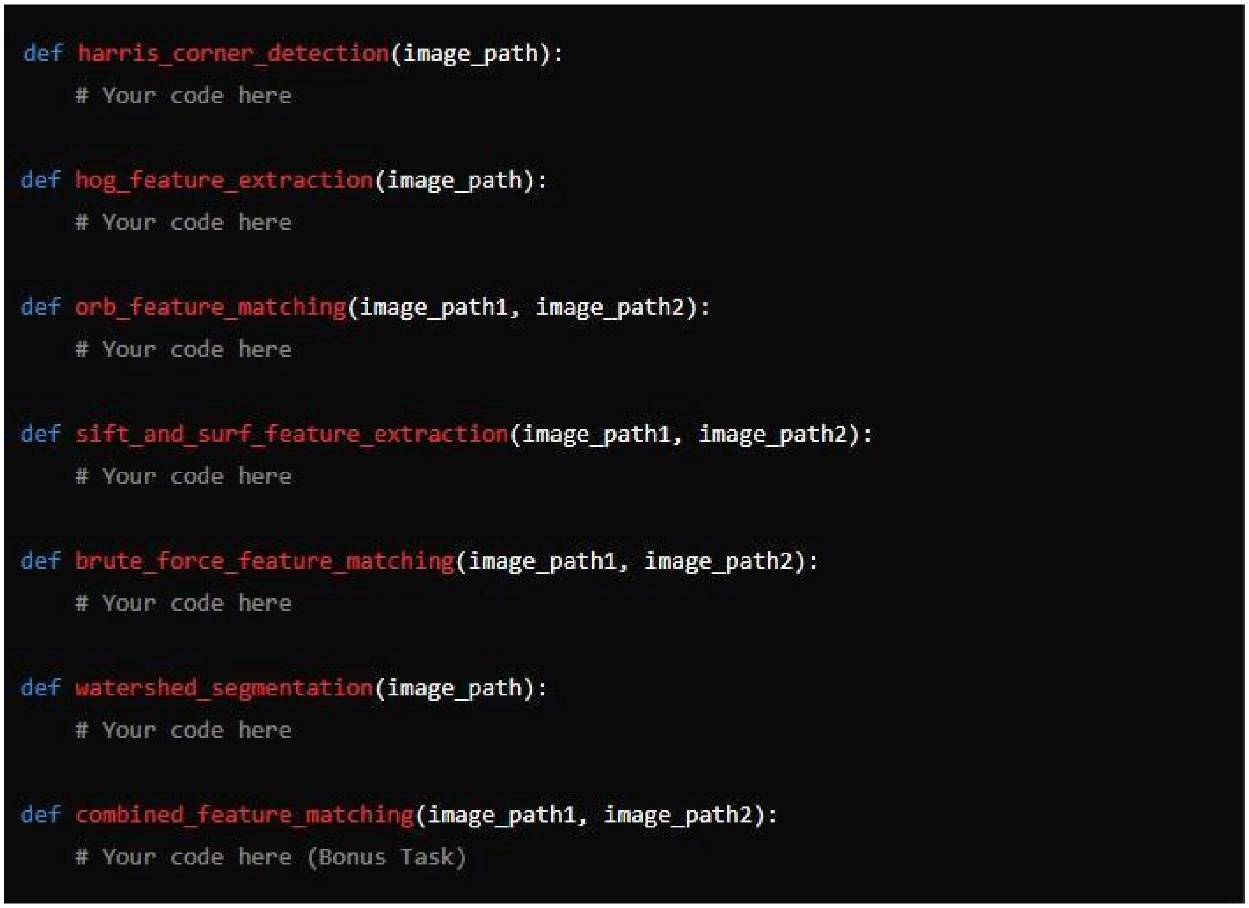
- **GitHub Repository**:
  - o Create a folder in your CSST106-Perception and Computer Vision repository named Feature-Extraction-Machine-Problem.
  - o Upload all code, images, and reports to this folder.

- **File Naming Format**: **[SECTION-LASTNAME-MP4]** 4D-LASTNAME-MP4
  - o 4D-BERNARDINO-SIFT.py
  - o 4D-BERNARDINO-Matching.jpg

Additional Penalties:

- **Incorrect Filename Format**: -5 points
- **Late Submission**: -5 points per day
- **Cheating/Plagiarism**: Zero points for the entire task

Sample Function Signatures



```python
def harris_corner_detection(image_path):
    # Your code here

def hog_feature_extraction(image_path):
    # Your code here

def orb_feature_matching(image_path1, image_path2):
    # Your code here

def sift_and_surf_feature_extraction(image_path1, image_path2):
    # Your code here

def brute_force_feature_matching(image_path1, image_path2):
    # Your code here

def watershed_segmentation(image_path):
    # Your code here

def combined_feature_matching(image_path1, image_path2):
    # Your code here (Bonus Task)
```

Rubric for Advanced Feature Extraction and Image Processing

Criteria	Excellent (90-100%)	Good (75-89%)	Satisfactory (60-74%)	Needs Improvement (0-59%)
Task 1: Harris Corner Detection	Clear detection and visualization	Minor issues in visualization	Basic implementation	Incorrect or no solution
Task 2: HOG Feature Extraction	Clear visualization and explanation	Minor issues in feature extraction	Basic explanation	Incorrect or no feature extraction
Task 3: ORB Feature Matching	Correct matching and visualization	Minor issues with matching	Basic attempt	Incorrect matching
Task 4: SIFT and SURF Extraction	Accurate feature detection	Minor issues with feature comparison	Basic implementation	Incorrect or no comparison
Task 5: Brute-Force Feature Matching	Clear visualization and comparison	Minor issues in visualization	Basic attempt	Incorrect matching
Task 6: Watershed Segmentation	Correct segmentation and result	Minor issues in segmentation	Basic segmentation	Incorrect or no segmentation