Republic of the Philippines
Laguna State Polytechnic University
Province of Laguna

MACHINE PROBLEM 4			
Topic:	**Module 2.0: Feature Extraction and Object Detection**	Week No.:	**6-7**
Course Code:	**CSST106**	Term:	**1st Sem.**
Course Title:	**Perception and Computer Vision**	Academic Year:	**2024-2025**
Student Name:	**Lesly-Ann B. Victoria**	Section:	**BSCS-4B**
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:

PACKAGES:

```
#Unistall the current Package.
!pip uninstall -y opencv-python opencv-python-headless opencv-contrib-python
```

The command `!pip uninstall -y opencv-python opencv-python-headless opencv-contrib-python` removes all installed versions of OpenCV in the environment. The `-y` flag automatically confirms the uninstallation. This command clears out `opencv-python` (standard with GUI), `opencv-python-headless` (optimized for non-GUI environments), and `opencv-contrib-python` (which includes additional modules), allowing for a fresh setup if reinstalling specific OpenCV versions.

```
#Install necessary development tools and dependencies required to build OpenCV from source.
!apt-get install -y cmake
!apt-get install -y libopencv-dev build-essential cmake git pkg-config libgtk-3-dev \
   libavcodec-dev libavformat-dev libswscale-dev libtbb2 libtbb-dev libjpeg-dev \
   libpng-dev libtiff-dev libdc1394-22-dev libv4l-dev v4l-utils \
   libxvidcore-dev libx264-dev libxine2-dev gstreamer1.0-tools \
   libgstreamer-plugins-base1.0-dev libgstreamer-plugins-good1.0-dev \
   libgtk2.0-dev libtiff5-dev libopenexr-dev libatlas-base-dev \
   python3-dev python3-numpy libtbb-dev libeigen3-dev \
   libfaac-dev libmp3lame-dev libtheora-dev libvorbis-dev \
   libxvidcore-dev libx264-dev yasm libopencore-amrnb-dev \
   libopencore-amrwb-dev libv4l-dev libxine2-dev libtesseract-dev \
   liblapacke-dev libopenblas-dev checkinstall
```

This command installs the necessary development tools and dependencies required to build OpenCV from source. It first installs `cmake`, a build automation tool, and then a wide range of libraries essential for OpenCV's functionality, including those for image and video processing (like `libjpeg-dev`, `libpng-dev`, `libtiff-dev`), video streaming (like `libavcodec-dev`, `libavformat-dev`), graphical interfaces (`libgtk-3-dev`), and various mathematical and computational libraries (`libtbb-dev`, `libeigen3-dev`). Additionally, it includes Python support (`python3-dev`, `python3-numpy`) and other multimedia codecs (`libxvidcore-dev`, `libx264-dev`, `libmp3lame-dev`) to enable full multimedia processing capabilities in OpenCV.

```
#Clone the OpenCV repository from GitHub.
!git clone https://github.com/opencv/opencv.git
!git clone https://github.com/opencv/opencv_contrib.git
```

This command clones the OpenCV repositories from GitHub to the local environment. The first line, `!git clone https://github.com/opencv/opencv.git`, downloads the main OpenCV repository, which includes core functionalities and standard modules. The second line, `!git clone https://github.com/opencv/opencv_contrib.git`, downloads the "contrib" repository, which contains additional modules and experimental features not included in the main repository. Cloning both allows you to build OpenCV with extended functionalities if desired.

```
#Change directory to the cloned OpenCV directory.
%cd opencv
#Create a build directory for building the OpenCV source.
!mkdir build
#Move into the newly created build directory.
%cd build

#Run the CMake configuration for building OpenCV with specific options:
!cmake -D CMAKE_BUILD_TYPE=RELEASE \
       -D CMAKE_INSTALL_PREFIX=/usr/local \
       -D OPENCV_ENABLE_NONFREE=ON \
       -D OPENCV_EXTRA_MODULES_PATH=../../opencv_contrib/modules \
       -D BUILD_EXAMPLES=ON ..

#Compile OpenCV using 8 threads (parallel compilation) for faster build times.
!make -j8
#Install the compiled OpenCV library into the system.
!make install
```

This code sets up and compiles OpenCV from source with custom configurations. First, it changes to the OpenCV directory and creates a `build` directory for organizing the build files. Inside the `build` directory, it uses CMake to configure the build with specific options, such as enabling non-free modules and including extra modules from `opencv_contrib`. Then, it compiles OpenCV using 8 threads to speed up the process (`-j8`), and finally installs the compiled library to the system. This allows for a customized OpenCV installation with additional modules and optimized settings

```
#Import Libraries.
import cv2
import numpy as np
from matplotlib import pyplot as plt
from PIL import Image
from skimage.feature import hog
from skimage import exposure
```

Importing the necessary Libraries.

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):

CODE:

This code implements the Harris Corner Detection technique to identify corners in an image, which are points where the intensity changes in multiple directions. First, it loads an

image from the specified path in grayscale mode. Grayscale simplifies the process by using only intensity values, eliminating color information. The grayscale image is then converted to a `float32` format to prepare for the Harris Corner Detection, which requires high-precision data for effective corner computation. The Harris Corner Detection function, `cornerHarris`, takes the grayscale image and applies parameters such as `blockSize`, `ksize`, and `k` to define the region for corner detection. `blockSize` sets the neighborhood size, `ksize` defines the size of the Sobel operator, and `k` is a free parameter that influences the sensitivity to corner-like regions. After applying the detection, the corners are enhanced by dilation, which makes them visually more prominent.

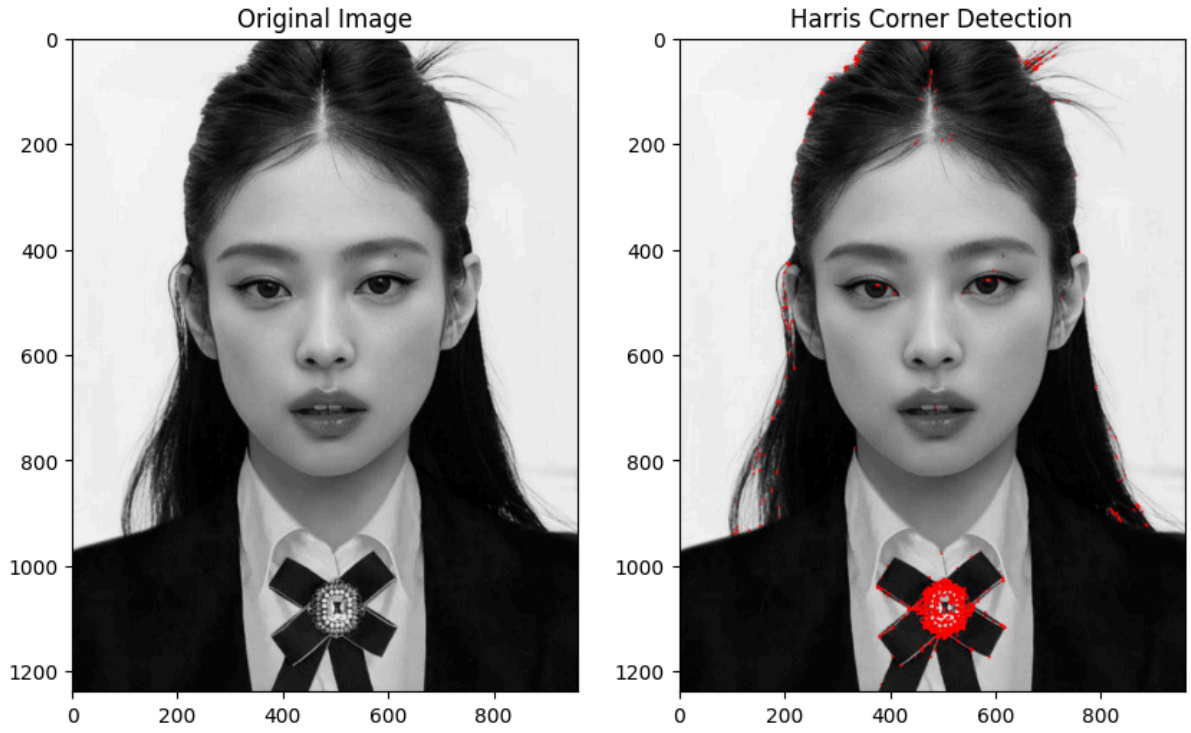
Next, the code creates a color version of the grayscale image to display the detected corners. Detected corner points are marked in red, using a condition that identifies strong corner responses based on a threshold of 1% of the maximum corner response. This step highlights the detected corners for easy visualization. The original grayscale image and the image with corners marked are then displayed side-by-side using Matplotlib, providing a clear comparison of the original image and the results of the Harris Corner Detection process.

```
def harris_corner_detection(image_path):
    #Load the image in grayscale.
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    #Convert to float32 for processing.
    img_float = np.float32(img)
    #Apply Harris Corner Detection.
    harris_corners = cv2.cornerHarris(img_float, blockSize=2, ksize=3, k=0.04)
    #Dilate the corner points to enhance them.
    harris_corners = cv2.dilate(harris_corners, None)
    #Create a color version of the grayscale image to display the red corner points.
    img_with_corners = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
    #Mark corners with red color.
    img_with_corners[harris_corners > 0.01 * harris_corners.max()] = [255, 0, 0]
    #Display the original and corner-detected images side by side.
    plt.figure(figsize=(10, 6))
    plt.subplot(1, 2, 1)
    plt.imshow(img, cmap='gray')
    plt.title('Original Image')

    plt.subplot(1, 2, 2)
    plt.imshow(img_with_corners)
    plt.title('Harris Corner Detection')
    plt.show()

#Load the image.
image_path = '/content/drive/MyDrive/jennie.jpg'
harris_corner_detection(image_path)
```

RESULT:



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):

CODE:

The code defines a function called `hog_feature_extraction`, which performs feature extraction using Histogram of Oriented Gradients (HOG) on an input image. First, the function reads the image in grayscale, making it suitable for processing by focusing on intensity rather than color. HOG is a technique commonly used in computer vision for detecting object features by analyzing the orientation and magnitude of gradients within localized sections of an image. The function uses the `hog` method with specific parameters: nine gradient orientations, an 8x8 pixels-per-cell size, and a 2x2 cells-per-block size. These parameters control how the gradients are computed and organized, ensuring a detailed yet compact representation of the image's structural features. The block normalization method is set to 'L2-Hys', which further refines the HOG descriptors for consistency across varying image contrasts. For visualization purposes, the code then adjusts the intensity of the HOG

visualization image, ensuring that the extracted features are easily visible. This visualization helps to interpret the HOG output by showing the edge patterns and gradient structures identified by the algorithm.

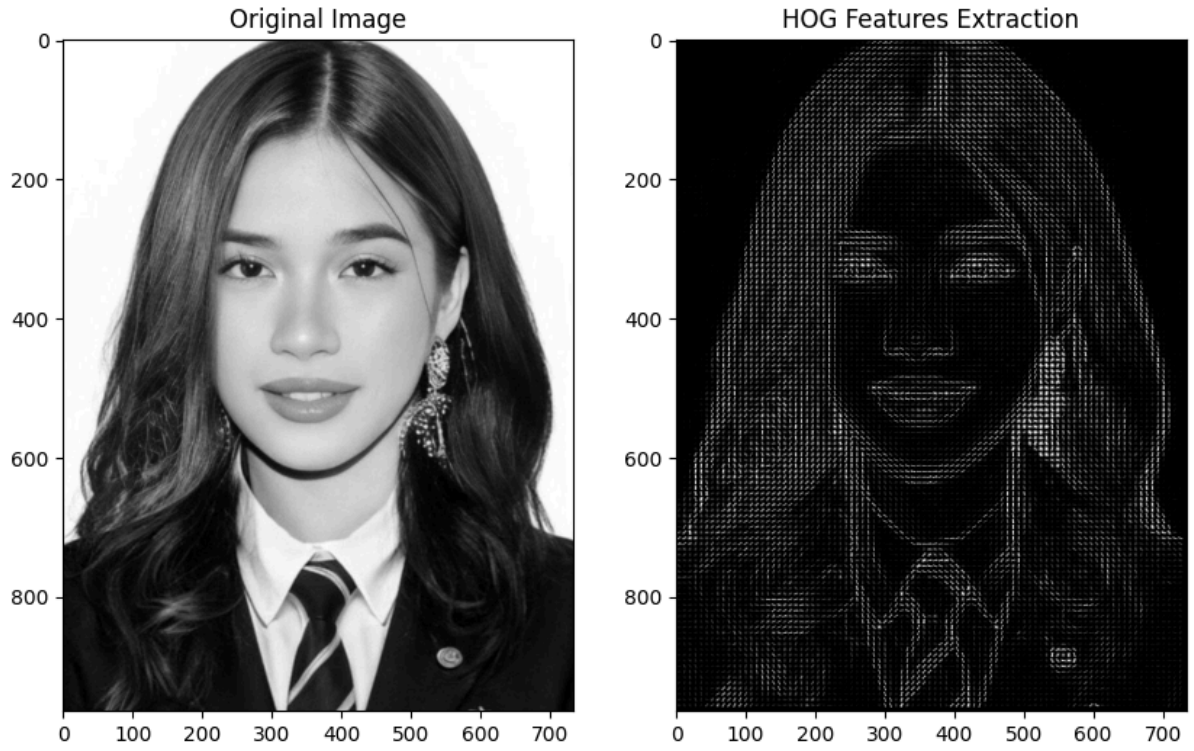
Finally, it displays the original grayscale image and the HOG features side by side in a plot. This dual display aids in understanding the correlation between the raw image and the extracted HOG features, providing insights into which aspects of the image are most prominent in terms of gradient orientation.

```
def hog_feature_extraction(image_path):
    #Load the image in grayscale.
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    #Extract HOG features.
    hog_features, hog_image = hog(img,
                                  orientations=9,
                                  pixels_per_cell=(8, 8),
                                  cells_per_block=(2, 2),
                                  block_norm='L2-Hys',
                                  visualize=True)
    #Adjust the intensity of the HOG image for better visualization.
    hog_image = exposure.rescale_intensity(hog_image, in_range=(0, 10))
    #Display the original and HOG visualized images side by side.
    plt.figure(figsize=(10, 6))
    plt.subplot(1, 2, 1)
    plt.imshow(img, cmap='gray')
    plt.title('Original Image')

    plt.subplot(1, 2, 2)
    plt.imshow(hog_image, cmap='gray')
    plt.title('HOG Features Extraction')
    plt.show()

#Load the image.
image_path = '/content/drive/MyDrive/aiah.jpg'
hog_feature_extraction(image_path)
```

RESULT:



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):

CODE:

This code defines a function, `orb_feature_matching`, which performs ORB-based feature extraction and matching on two input images. The function begins by loading two grayscale images from specified file paths, making them easier to process since ORB (Oriented FAST and Rotated BRIEF) is optimized for grayscale images. To ensure accurate matching, the images are resized to the same dimensions. The ORB detector is then initialized, which identifies keypoints (distinctive patterns or "features" in the images) and computes descriptors (compact representations of these features) for both images. ORB is a computationally efficient alternative to other feature detectors like SIFT and SURF, particularly suited for real-time applications.

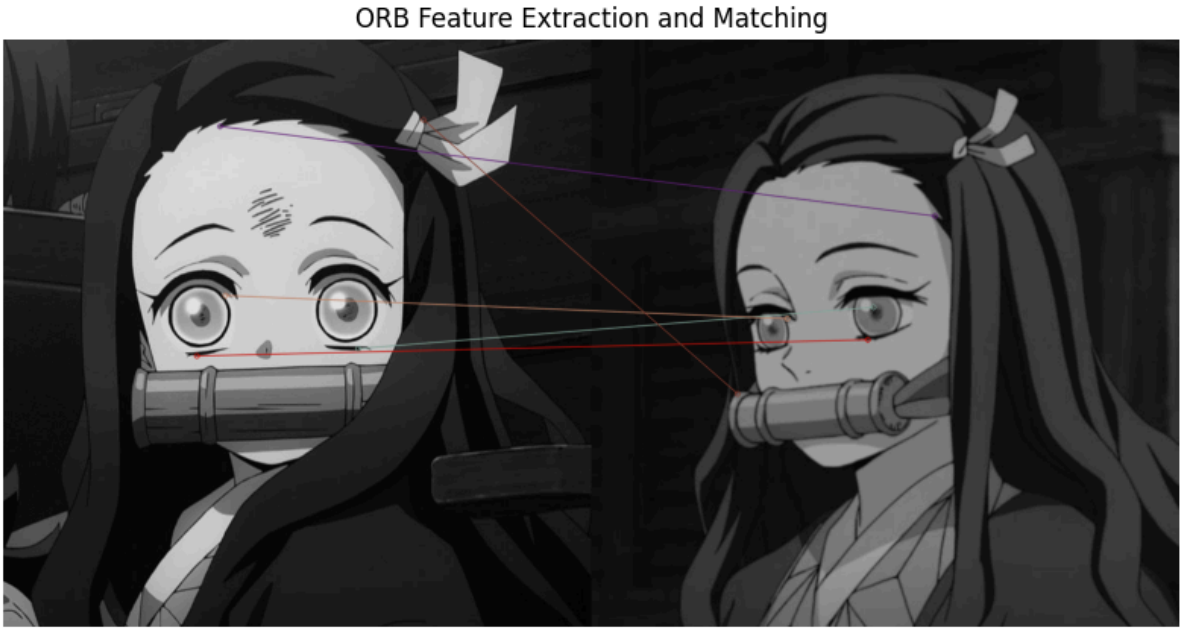
Next, the function sets up the FLANN-based (Fast Library for Approximate Nearest Neighbors) matcher, which is configured for the ORB descriptors by specifying parameters like the algorithm type and search parameters to optimize the matching process. The function performs a k-nearest neighbors (k=2) match, finding the two closest matches for each descriptor in the first image. To filter the results, Lowe's ratio test is applied to retain only high-quality matches, where the closest match is significantly closer than the second-closest match. This step helps to avoid mismatches by accepting only distinct matches. The function then visualizes the matches by drawing lines between matched keypoints in the two images, creating an overlay that highlights matched features.

Finally, the matched image is displayed using Matplotlib, showing the ORB keypoints and their connections between the two images. This visualization is useful for confirming the accuracy of the feature matching process and analyzing how well the ORB detector and FLANN matcher work together on the chosen images.

```python
def orb_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)
    #Resize the images to the same size.
    height, width = img1.shape[:2]
    img2 = cv2.resize(img2, (width, height))
    #Initialize the ORB detector.
    orb = cv2.ORB_create()
    #Detect keypoints and compute descriptors for both images.
    keypoints1, descriptors1 = orb.detectAndCompute(img1, None)
    keypoints2, descriptors2 = orb.detectAndCompute(img2, None)
    #Initialize FLANN-based matcher.
    index_params = dict(algorithm=6, table_number=6, key_size=12, multi_probe_level=1)
    search_params = dict(checks=50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    #Match descriptors using FLANN.
    matches = flann.knnMatch(descriptors1, descriptors2, k=2)
    #Filter matches using the Lowe's ratio test.
    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, keypoints1, img2, keypoints2, good_matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    #Display the matching keypoints.
    plt.figure(figsize=(10, 6))
    plt.imshow(img_matches)
    plt.title("ORB Feature Extraction and Matching")
    plt.axis('off')
    plt.show()

#Load the Images.
image_path1 = '/content/drive/MyDrive/nezuko1.jpg'
image_path2 = '/content/drive/MyDrive/nezuko2.jpg'
orb_feature_matching(image_path1, image_path2)
```

RESULT:



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):

CODE:

This code performs feature extraction on two images using SIFT (Scale-Invariant Feature Transform) and SURF (Speeded-Up Robust Features) techniques. It begins by loading two input images in grayscale mode to reduce complexity and ensure the feature detectors work efficiently. Then, it resizes the second image to match the dimensions of the first image, enabling a direct comparison between the two images. For feature extraction, the code first uses SIFT, which identifies key points and computes corresponding descriptors for both images. These descriptors represent unique features in the images, allowing for comparison and matching. Similarly, it uses SURF to extract features from both images. SURF is typically faster but requires additional modules due to its higher computational demand.

Once the SIFT and SURF features are computed, the code visualizes them by drawing key points on each image. For SIFT, it displays the detected key points for each image (one showing a side view of Zenitsu and the other a front view) in a side-by-side plot. This process repeats for SURF, creating a second side-by-side plot of key points. Each plot highlights key

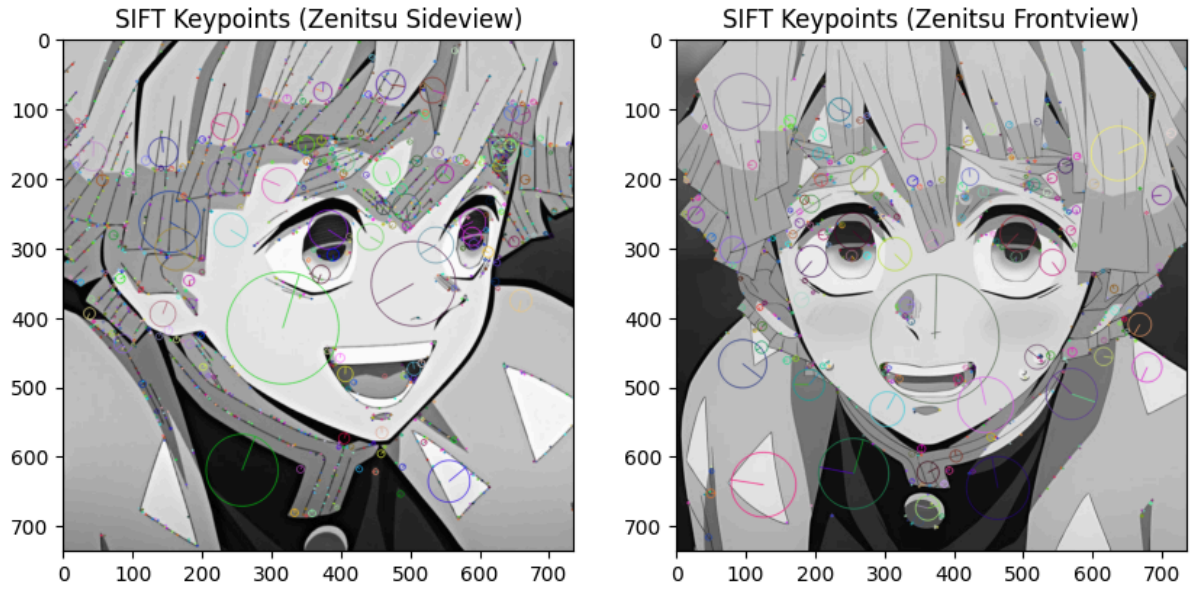
features of Zenitsu's face from both perspectives, allowing for visual analysis of how each feature detection method identifies unique aspects across different views. Finally, by displaying the plots, this function allows us to see and compare how well SIFT and SURF capture critical details in each perspective of Zenitsu's face, which is valuable for applications like object recognition, facial analysis, and computer vision tasks where detailed feature matching is essential.

```
def sift_and_surf_feature_extraction(image_path1, image_path2):
    #Load the two images in grayscale.
    img1 = cv2.imread(image_path1, cv2.IMREAD_GRAYSCALE)
    img2 = cv2.imread(image_path2, cv2.IMREAD_GRAYSCALE)
    #Resize the second image to match the size of the first.
    height, width = img1.shape[:2]
    img2 = cv2.resize(img2, (width, height))
    #SIFT Feature Extraction.
    sift = cv2.SIFT_create()
    keypoints1_sift, descriptors1_sift = sift.detectAndCompute(img1, None)
    keypoints2_sift, descriptors2_sift = sift.detectAndCompute(img2, None)
    #SURF Feature Extraction (requires xfeatures2d module).
    surf = cv2.xfeatures2d.SURF_create(hessianThreshold=400)
    keypoints1_surf, descriptors1_surf = surf.detectAndCompute(img1, None)
    keypoints2_surf, descriptors2_surf = surf.detectAndCompute(img2, None)
    #SIFT Keypoints.
    img1_sift = cv2.drawKeypoints(img1, keypoints1_sift, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    img2_sift = cv2.drawKeypoints(img2, keypoints2_sift, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    #SURF Keypoints.
    img1_surf = cv2.drawKeypoints(img1, keypoints1_surf, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    img2_surf = cv2.drawKeypoints(img2, keypoints2_surf, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    #Plot SIFT results.
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.imshow(img1_sift, cmap='gray')
    plt.title("SIFT Keypoints (Zenitsu Sideview)")
    plt.subplot(1, 2, 2)
    plt.imshow(img2_sift, cmap='gray')
    plt.title("SIFT Keypoints (Zenitsu Frontview)")
    plt.show()

    #Plot SURF results.
    plt.figure(figsize=(10, 6))
    plt.subplot(1, 2, 1)
    plt.imshow(img1_surf, cmap='gray')
    plt.title("SURF Keypoints (Zenitsu Sideview)")
    plt.subplot(1, 2, 2)
    plt.imshow(img2_surf, cmap='gray')
    plt.title("SURF Keypoints (Zenitsu Frontview)")
    plt.show()

#Load the Images.
image_path1 = '/content/drive/MyDrive/zenitsu1.jpg'
image_path2 = '/content/drive/MyDrive/zenitsu2.jpg'
sift_and_surf_feature_extraction(image_path1, image_path2)
```

RESULT:



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):

CODE:

This code performs feature matching between two images using ORB (Oriented FAST and Rotated BRIEF) features and a Brute-Force Matcher. It starts by loading the images in grayscale to simplify processing, as grayscale images are sufficient for detecting keypoints and computing descriptors in many computer vision applications. The code also resizes the second image to match the dimensions of the first, ensuring alignment between the images for better comparison. The ORB detector is initialized to identify keypoints and calculate feature descriptors for each image. ORB is chosen because it is efficient, robust to rotation, and ideal for real-time applications. Next, a Brute-Force Matcher is set up with Hamming distance as the similarity metric, which is well-suited for binary descriptors like those generated by ORB. The Brute-Force Matcher finds the best matches for each descriptor pair by minimizing the Hamming distance, and cross-checking is enabled to improve the match quality.

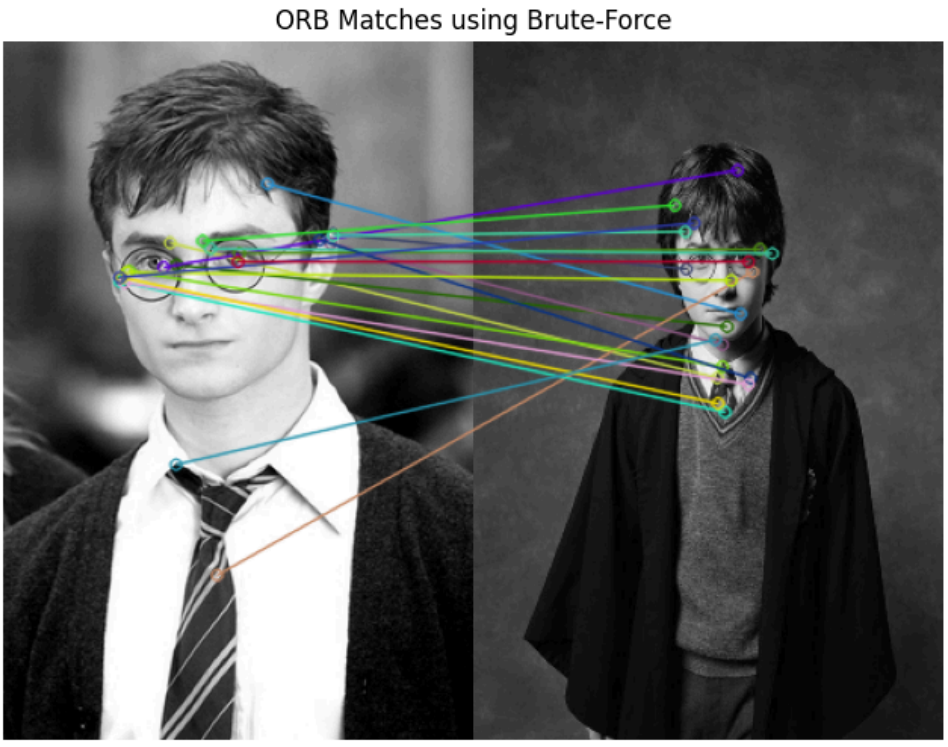
The matches are sorted by distance to prioritize the best matches, with the closest ones being the most relevant. Finally, the code uses OpenCV's `drawMatches` function to visually represent the top 20 matches between the two images, with lines connecting the

matching keypoints across the images. The result is displayed using Matplotlib, showing matched features and giving an intuitive overview of the feature similarity between the two images. This matching technique can be beneficial for tasks like image alignment, object recognition, or tracking.

```python
def brute_force_feature_matching(image_path1, image_path2):
    #Load the two images in grayscale.
    img1 = cv2.imread(image_path1, cv2.IMREAD_GRAYSCALE)
    img2 = cv2.imread(image_path2, cv2.IMREAD_GRAYSCALE)
    #Resize the second image to match the size of the first.
    height, width = img1.shape[:2]
    img2 = cv2.resize(img2, (width, height))
    #Initialize the ORB detector.
    orb = cv2.ORB_create()
    #Detect keypoints and compute descriptors for both images.
    keypoints1, descriptors1 = orb.detectAndCompute(img1, None)
    keypoints2, descriptors2 = orb.detectAndCompute(img2, None)
    #Initialize Brute-Force Matcher with Hamming distance (suitable for ORB).
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    #Match descriptors.
    matches = bf.match(descriptors1, descriptors2)
    #Sort matches based on distance (best matches first).
    matches = sorted(matches, key=lambda x: x.distance)
    #Draw matches.
    img_matches = cv2.drawMatches(img1, keypoints1, img2, keypoints2, matches[:20], None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    #Display the matching keypoints.
    plt.figure(figsize=(10, 6))
    plt.imshow(img_matches)
    plt.title("ORB Matches using Brute-Force")
    plt.axis('off')
    plt.show()

#Load the Images.
image_path1 = '/content/drive/MyDrive/HarryPotter1.jpg'
image_path2 = '/content/drive/MyDrive/HarryPotter2.jpg'
brute_force_feature_matching(image_path1, image_path2)
```

RESULT:



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):

CODE:

This code performs image segmentation using the Watershed algorithm, a popular method for isolating distinct regions within an image. The process begins by reading an image from a specified path, followed by converting it to grayscale to simplify further processing. To separate the foreground (areas of interest) from the background, the grayscale image undergoes thresholding, using Otsu's method, which adapts to the intensity distribution in the image for optimal separation.

Next, noise is removed through morphological "opening," a series of erosion followed by dilation, which helps smooth small irregularities in the binary image. The code then dilates the smoothed image to confidently mark the background regions. For identifying the foreground, it applies a distance transformation—calculating the distance of each pixel to the nearest zero pixel (background)—and thresholds it to pinpoint the most certain foreground areas.

To handle the ambiguous regions between foreground and background, the code computes the "unknown" areas, where neither background nor foreground is entirely clear. This creates a marker array for Watershed, in which background pixels are labeled as 1, the unknown region as 0, and distinct foreground areas with unique labels. The Watershed algorithm is then applied, which iteratively "floods" each marker from the center outwards, following boundaries until reaching other markers, thus segmenting the regions.

To enhance visibility of boundaries in the final result, the code thickens the watershed lines using a boundary mask, which is then dilated and applied to the original image. The boundaries between segmented regions appear as thick green lines. Finally, the segmented image is displayed, showing clear boundaries between distinct areas, demonstrating the effectiveness of the Watershed algorithm in separating complex image regions.

```python
def watershed_segmentation(image_path):
    img = cv2.imread(image_path)
    #Convert the image to grayscale.
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    #Apply thresholding to separate foreground and background.
    ret, binary = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
    #Remove noise using morphological operations (opening).
    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 and thresholding.
    dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
    ret, sure_fg = cv2.threshold(dist_transform, 0.7 * dist_transform.max(), 255, 0)
    #Identify unknown region (where foreground and background are unclear).
    sure_fg = np.uint8(sure_fg)
    unknown = cv2.subtract(sure_bg, sure_fg)
    #Label markers for watershed.
    ret, markers = cv2.connectedComponents(sure_fg)
    #Add one to all labels so that sure background is not zero, but 1.
    markers = markers + 1
    #Mark the unknown region as zero.
    markers[unknown == 255] = 0
    #Apply the Watershed algorithm.
    markers = cv2.watershed(img, markers)
    #Mark boundaries with a thicker line by dilating the boundary mask.
    img[markers == -1] = [0, 255, 0]
    boundary_mask = np.uint8(markers == -1) * 255
    thick_boundary = cv2.dilate(boundary_mask, kernel, iterations=2)
    #Apply the thickened boundary mask to the original image.
    img[thick_boundary == 255] = [0, 255, 0]
    #Display the segmented image.
    plt.figure(figsize=(10, 6))
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    plt.title("Segmented Image using Watershed Algorithm")
    plt.axis('off')
    plt.show()

#Load the Image.
image_path = '/content/drive/MyDrive/olaf.jpg'
watershed_segmentation(image_path)
```

RESULT: