Republic of the Philippines
Laguna State Polytechnic University
Province of Laguna

EXERCISE NO. 3			
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:	

Advanced Feature Extraction and Image Processing

Install necessary libraries

```
!pip install opencv-python matplotlib Pillow
!pip install opencv-python matplotlib scikit-image
```

Import necessary libraries

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from skimage import feature
```

Load image from the Google Drive

```
image_path = '/content/drive/MyDrive/2X2.jpg'
image = Image.open(image_path)
```

Convert the image to a NumPy array and then to a BGR format (OpenCV format) and grayscale

```
image = np.array(image)
image_bgr = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
gray_image = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)
plt.figure(figsize=(10, 6))
plt.imshow(gray_image, cmap='gray')
plt.title('Grayscale Image')
plt.axis('off')
plt.show()
```

RESULT:

Exercise 1: Harris Corner Detection

Task: Harris Corner Detection is a classic corner detection algorithm. Use the Harris Corner Detection algorithm to detect corners in an image.

- Load an image of your choice.
- Convert it to grayscale.
- Apply the Harris Corner Detection method to detect corners.
- Visualize the corners on the image and display the result.

Key Points:

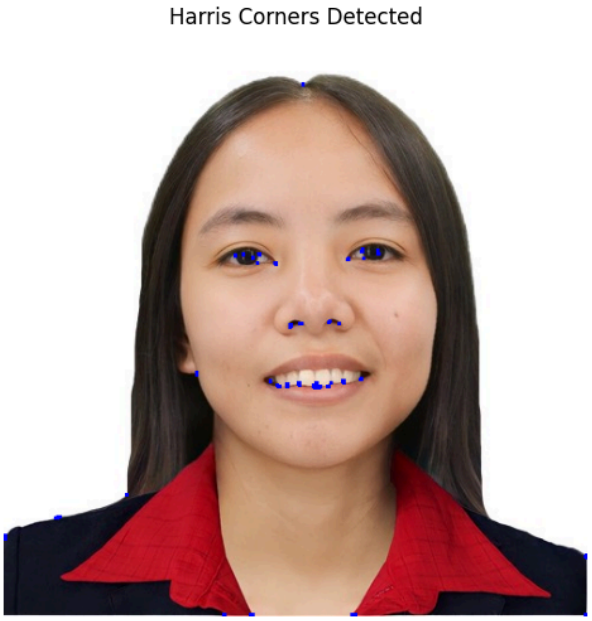
- Harris Corner Detection is used to find corners, which are points of interest.
- It's particularly useful for corner detection in images where object edges intersect.

Harris Corner Detection

```
#Apply Harris Corner Detection.
gray_image_float = np.float32(gray_image)
corners = cv2.cornerHarris(gray_image_float, blockSize=2, ksize=3, k=0.04)

#Dilate the corners to enhance the corner points.
corners = cv2.dilate(corners, None)

#Create a mask for corners.
threshold = 0.01 * corners.max()
image_bgr[corners > threshold] = [255, 0, 0]

#Visualize the results.
plt.figure(figsize=(10, 6))
plt.imshow(cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB))
plt.title('Harris Corners Detected')
plt.axis('off')
plt.show()
```

This code applies Harris Corner Detection, a technique used to identify key points in an image where there is a significant change in intensity. First, the grayscale image (`gray_image`) is converted to a float type to prepare for corner detection. Using OpenCV's `cornerHarris` function, the image is analyzed to find corner points, with parameters like `blockSize`, `ksize`, and `k` controlling the window size, Sobel operator, and sensitivity to corners, respectively. The detected corners are then dilated to make them more prominent. A threshold mask is created to highlight strong corner points by setting pixels above the threshold to red (`[255, 0, 0]`) in the color image (`image_bgr`). Finally, the results are visualized with `matplotlib`, showing the detected corners overlaid on the original image.

RESULT:



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

Task: The HOG descriptor is widely used for object detection, especially in human detection.

- Load an image of a person or any object.
- Convert the image to grayscale.
- Apply the HOG descriptor to extract features.
- Visualize the gradient orientations on the image.

Key Points:

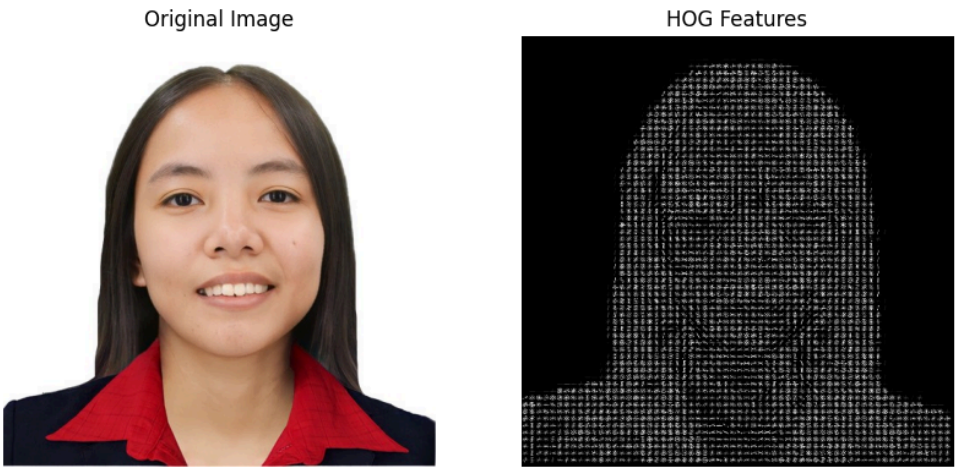
- HOG focuses on the structure of objects through gradients.
- Useful for human detection and general object recognition.

HOG (Histogram of Oriented Gradients) Feature Extraction

```
#Apply HOG descriptor.
hog_features, hog_image = feature.hog(
    gray_image,
    orientations=9,       #Number of orientation bins.
    pixels_per_cell=(8, 8),  #Size of each cell.
    cells_per_block=(2, 2),   #Size of blocks.
    visualize=True        #Visualize the HOG image.
)

#Rescale HOG image to 0-255 for visualization.
hog_image_rescaled = (hog_image * 255).astype(np.uint8)

#Visualize the original image and HOG features.
plt.figure(figsize=(10, 6))

#Original Image.
plt.subplot(1, 2, 1)
plt.imshow(image)
plt.title('Original Image')
plt.axis('off')

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

This code applies the Histogram of Oriented Gradients (HOG) descriptor to an image, which is a technique used in computer vision to extract texture and shape features. First, the `hog` function is called with parameters for orientations (direction bins), cell size, and block size. Setting `visualize=True` generates both the feature descriptor (`hog_features`) and a visual representation (`hog_image`) of the HOG features. The HOG image is then rescaled to a 0-255 range for better visualization. Finally, a plot displays two side-by-side images: the original image and the HOG feature visualization, allowing comparison of the raw image with its edge-like, gradient-based feature representation.

RESULT:



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

Task: FAST is another keypoint detector known for its speed.

- Load an image.
- Convert the image to grayscale.
- Apply the FAST algorithm to detect keypoints.
- Visualize the keypoints on the image and display the result.

Key Points:

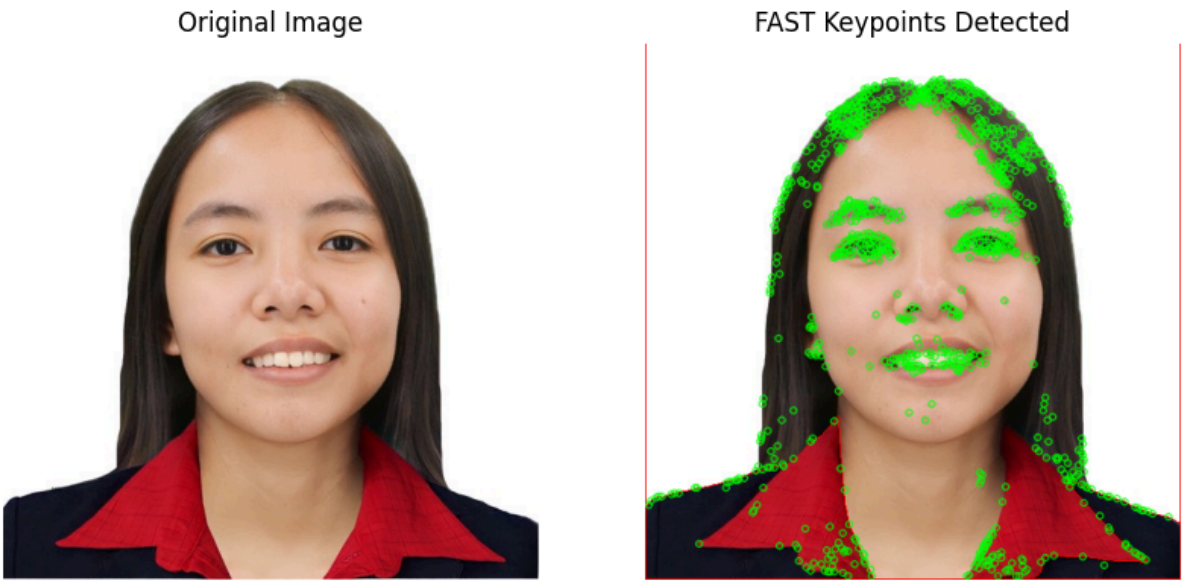
- FAST is designed to be computationally efficient and quick in detecting keypoints.
- It is often used in real-time applications like robotics and mobile vision.

FAST (Features from Accelerated Segment Test) Keypoint Detection

```
#Initialize the FAST detector.
fast = cv2.FastFeatureDetector_create()

#Detect keypoints.
keypoints = fast.detect(gray_image, None)

#Draw the keypoints on the image.
image_with_keypoints = cv2.drawKeypoints(image_bgr, keypoints, None, color=(0, 255, 0))

#Visualize the original image and the keypoints.
plt.figure(figsize=(10, 6))

#Original Image.
plt.subplot(1, 2, 1)
plt.imshow(image)
plt.title('Original Image')
plt.axis('off')

#Image with Keypoints.
plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(image_with_keypoints, cv2.COLOR_BGR2RGB))
plt.title('FAST Keypoints Detected')
plt.axis('off')
plt.show()
```

This code initializes and uses the FAST (Features from Accelerated Segment Test) detector to identify key points in an image. First, the FAST detector is created with `cv2.FastFeatureDetector_create()`. Then, it detects keypoints in a grayscale version of the image (`gray_image`) and draws them onto the original image (`image_bgr`) in green. Finally, it uses `matplotlib` to visualize the results by displaying two side-by-side images: the original image and the image with detected keypoints, which highlight prominent features in the image suitable for tasks like object recognition or tracking.

RESULT:



Exercise 4: Feature Matching using ORB and FLANN

Task: Use ORB descriptors to find and match features between two images using FLANN-based matching.

• Load two images of your choice.
• Extract keypoints and descriptors using ORB.
• Match features between the two images using the FLANN matcher.
• Display the matched features.

Key Points:

• ORB is fast and efficient, making it suitable for resource-constrained environments.
• FLANN (Fast Library for Approximate Nearest Neighbors) speeds up the matching process, making it ideal for large datasets.

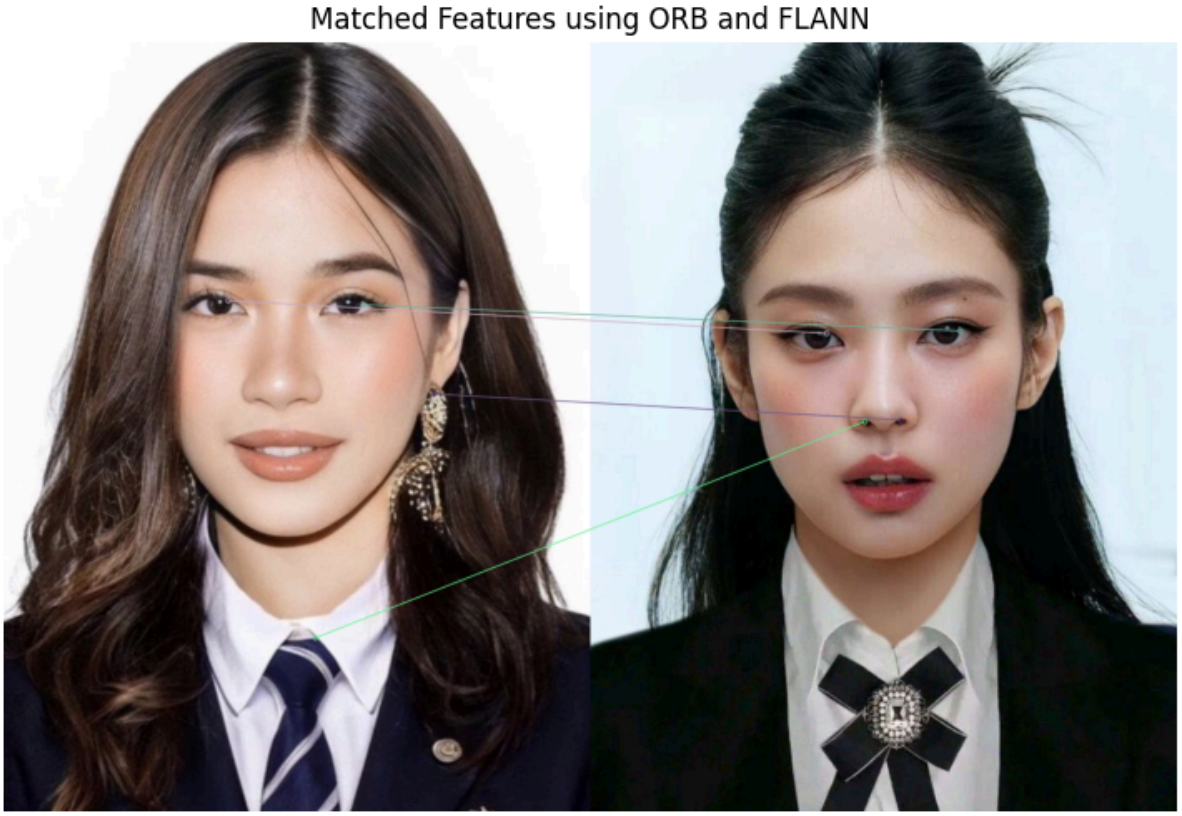
Feature Matching using ORB and FLANN

```python
#Load the two images using PIL.
image_path1 = '/content/drive/MyDrive/aiah.jpg'
image_path2 = '/content/drive/MyDrive/jennie.jpg'
image1 = Image.open(image_path1)
image2 = Image.open(image_path2)

#Convert the images to NumPy arrays and then to BGR format (OpenCV format).
image1 = np.array(image1)
image2 = np.array(image2)
image1_bgr = cv2.cvtColor(image1, cv2.COLOR_RGB2BGR)
image2_bgr = cv2.cvtColor(image2, cv2.COLOR_RGB2BGR)

#Resize the images to the same size.
height, width = image1_bgr.shape[:2]
image2_bgr = cv2.resize(image2_bgr, (width, height))

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

#Detect keypoints and compute descriptors for both images.
keypoints1, descriptors1 = orb.detectAndCompute(image1_bgr, None)
keypoints2, descriptors2 = orb.detectAndCompute(image2_bgr, None)

#Create FLANN matcher.
FLANN_INDEX_LSH = 6
index_params = dict(algorithm=FLANN_INDEX_LSH, table_number=6, key_size=12, multi_probe_level=1)
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)

#Match features between the two images.
matches = flann.knnMatch(descriptors1, descriptors2, k=2)

#Store good 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 on the images.
matched_image = cv2.drawMatches(image1_bgr, keypoints1, image2_bgr, keypoints2, good_matches, None,
                                flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

#Visualize the matched features.
plt.figure(figsize=(10, 6))
plt.imshow(cv2.cvtColor(matched_image, cv2.COLOR_BGR2RGB))
plt.title('Matched Features using ORB and FLANN')
plt.axis('off')
plt.show()
```

This code uses Python to perform feature matching between two images with OpenCV. First, it loads two images using the `PIL` library, converts them to NumPy arrays, and then switches their color format from RGB (used by PIL) to BGR (used by OpenCV). Next, it resizes one image to match the dimensions of the other. An ORB (Oriented FAST and Rotated BRIEF) detector, which identifies keypoints and computes descriptors, is then initialized for both images. A FLANN-based (Fast Library for Approximate Nearest Neighbors) matcher is used to find similar features between the images. Good matches are

selected based on Lowe's ratio test to filter out weak matches, and the matched features are drawn and visualized using Matplotlib. This allows for visual inspection of similarities between the two images based on keypoint features.

RESULT:



Exercise 5: Image Segmentation using Watershed Algorithm

Task: The Watershed algorithm segments an image into distinct regions.

- Load an image.
- Apply a threshold to convert the image to binary.
- Apply the Watershed algorithm to segment the image into regions.
- Visualize and display the segmented regions.

Key Points:

- Image segmentation is crucial for object detection and recognition.
- The Watershed algorithm is especially useful for separating overlapping objects.

Image Segmentation using Watershed Algorithm

```
#Apply thresholding to create a binary image.
_, binary_image = cv2.threshold(gray_image, 127, 255, cv2.THRESH_BINARY_INV)

#Remove noise and prepare for Watershed.
kernel = np.ones((3, 3), np.uint8)
dilated_image = cv2.dilate(binary_image, kernel, iterations=2)
dist_transform = cv2.distanceTransform(dilated_image, cv2.DIST_L2, 5)
_, sure_fg = cv2.threshold(dist_transform, 0.7 * dist_transform.max(), 255, 0)

#Find unknown region.
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(dilated_image, sure_fg)

#Label markers.
ret, markers = cv2.connectedComponents(sure_fg)
markers = markers + 1

#Mark the unknown regions with zero.
markers[unknown == 300] = 0

#Apply the Watershed algorithm.
markers = cv2.watershed(image_bgr, markers)
image_bgr[markers == -1] = [0, 0, 255]

#Visualize the segmented regions.
plt.figure(figsize=(10, 6))
plt.imshow(cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB))
plt.title('Segmented Image using Watershed Algorithm')
plt.axis('off')
plt.show()
```

This code applies the Watershed algorithm to segment regions in an image. It begins by thresholding a grayscale image (`gray_image`) to create a binary image where pixels are either black or white. Then, it dilates the binary image to remove noise and enhance areas for segmentation. A distance transform identifies the sure foreground regions, and by subtracting it from the dilated image, it identifies unknown regions.

Next, connected components are labeled to prepare markers for Watershed, with unknown regions marked as zero. When the Watershed algorithm is applied, it treats pixel boundaries as watershed lines, separating distinct objects. The code then highlights these boundaries in red on the original color image (`image_bgr`). Finally, it displays the segmented image with matplotlib, visually emphasizing the regions detected by Watershed.

RESULT:

CONCLUSION:

In summary, this series of exercises demonstrates essential techniques in advanced feature extraction and image processing, using methods commonly applied in computer vision tasks:

1. ***Harris Corner Detection*** - identifies corners in an image, which are useful for locating points where object edges intersect, enhancing feature detection for tracking and mapping.
2. ***Histogram of Oriented Gradients (HOG)*** - emphasizes texture and shape through gradient-based features, which are particularly useful for object detection tasks such as human recognition.
3. ***FAST Keypoint Detection*** - offers a quick and computationally efficient way to detect keypoints, especially suited for real-time applications like robotics and mobile vision.
4. ***ORB and FLANN-Based Feature Matching*** - enables feature comparison between two images, leveraging ORB's efficiency and FLANN's speed to find similar keypoints. This technique is ideal for matching objects across frames or identifying objects in different views.
5. ***Watershed Algorithm for Image Segmentation*** - segments an image into distinct regions by separating overlapping objects, which is crucial for object detection, image analysis, and recognition tasks.

Overall, these techniques cover a spectrum of feature extraction and image processing strategies, from corner detection to segmentation. Each method addresses specific needs in computer vision, enabling tasks like object recognition, real-time keypoint detection, and image segmentation. Combining these methods can significantly improve the accuracy and robustness of applications in fields like autonomous driving, surveillance, and augmented reality.