4B BAUTISTA EXER 3

Advanced Feature Extraction and Image Processing.  
  
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  
  
# Load the image

img = cv2.imread('great-dane.png')

gray\_image = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Apply Harris Corner Detection

gray = np.float32(gray\_image)

corners = cv2.cornerHarris(gray, 2, 3, 0.04)

# Dilate the corners for better visibility

corners = cv2.dilate(corners, None)

# Mark corners on the original image

img[corners > 0.01 \* corners.max()] = [0, 0, 255]

# Display the result

plt.imshow(cv2.cvtColor(img, cv2.COLOR\_BGR2RGB))

plt.title('Harris Corner Detection')

plt.show()

Explanation : mark area that will be shown as the key feature of the sample.

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.

# Load Image

img = cv2.imread('great-dane.png')

gray\_image = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Compute HOG feature and visualize the gradient orientations

hog\_fearture, hog\_image = hog(gray\_image, orientations=8, pixels\_per\_cell=(8, 8), cells\_per\_block=(2, 2),

                              visualize=True, feature\_vector=True)

# Rescale the intensity of the HOG image for visualization

hog\_image\_rescaled = exposure.rescale\_intensity(hog\_image, in\_range=(0, 10))

# Display the HOG image

plt.imshow(hog\_image\_rescaled, cmap='gray')

plt.title('HOG Feature Visualization')

plt.show()

Explanation : make a Pixelate copy of the input sample .

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.

img = cv2.imread('great-dane.png')

gray\_image = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Apply FAST algorithm to detect keypoints

fast = cv2.FastFeatureDetector\_create()

keypoints = fast.detect(gray\_image, None)

# Draw Keypoints on the image

img\_keypoints = cv2.drawKeypoints(gray\_image, keypoints, None, color=(0, 255, 0))

# Display the image with keypoints

plt.imshow(cv2.cvtColor(img\_keypoints, cv2.COLOR\_BGR2RGB))

plt.title('FAST Keypoint Detection')

plt.show()

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.

# Load Image

img1 = cv2.imread('great-dane.png')

img2 = cv2.imread('great-dane-2.png')

# 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)

# FLANN based 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 description

matches = flann.knnMatch(des1, des2, k=2)

# Apply ratio test

good\_matches = []

for m, n in matches:

    if m.distance < 0.7 \* n.distance:

        good\_matches.append(m)

# Draw the matches

result = cv2.drawMatches(img1, kp1, img2, kp2, good\_matches, None, flags=2)

# Display result

plt.imshow(result)

plt.title('Feature Matching')

plt.show()

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.

# Load Image

img = cv2.imread('great\_dane.png')

gray\_image = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Apply threshold to convert the image to binary using Otsu's method

ret , binary\_image = cv2.threshold(gray\_image, 0, 255, cv2.THRESH\_BINARY\_INV + cv2.THRESH\_OTSU)

# Noise removal (morphological opening)

kernel = np.ones((3, 3), np.uint8)

opening = cv2.morphologyEx(binary\_image, cv2.MORPH\_OPEN, kernel, iterations=2)

# Sure background (dilate the binary image)

sure\_bg = cv2.dilate(opening, kernel, iterations=3)

# Finding sure foreground area using distance transform

dist\_transform = cv2.distanceTransform(opening, cv2.DIST\_L2, 5)

ret, sure\_fg = cv2.threshold(dist\_transform, 0.7 \* dist\_transform.max(), 255, 0)

# Convert sure foreground to uint8 and find unknown region

sure\_fg = np.uint8(sure\_fg)

unknown = cv2.subtract(sure\_bg, sure\_fg)

# Marker labelling using connected components

ret, markers = cv2.connectedComponents(sure\_fg)

# Add one to all labels so that sure background is not 0, but 1

markers = markers + 1

# Mark the region Unknown with 0

markers[unknown == 0] = 0

# Apply watershed algorithm

cv2.watershed(img, markers)

# Mark boundaries of the segmented regions (red boundaries)

img[markers == -1] = [255, 0, 0]

# Display the result

plt.imshow(cv2.cvtColor(img, cv2.COLOR\_BGR2RGB))

plt.title('Watershed Segmentation')

plt.axis('off')  # Hide axis for better visualization

plt.show()