Exercise 3 - Advanced Feature Extraction and Image Processing

---

This exercise will cover different advanced feature extraction and image processing:

- **Harris Corner Detection**
- **HOG (Histogram of Oriented Gradients) Feature Extraction**
- **FAST (Features from Accelerated Segment Test) Keypoint Detection**
- **Feature Matching using ORB and FLANN**
- **Image Segmentation using Watershed Algorithm**

Task 1: Harris Corner Detection

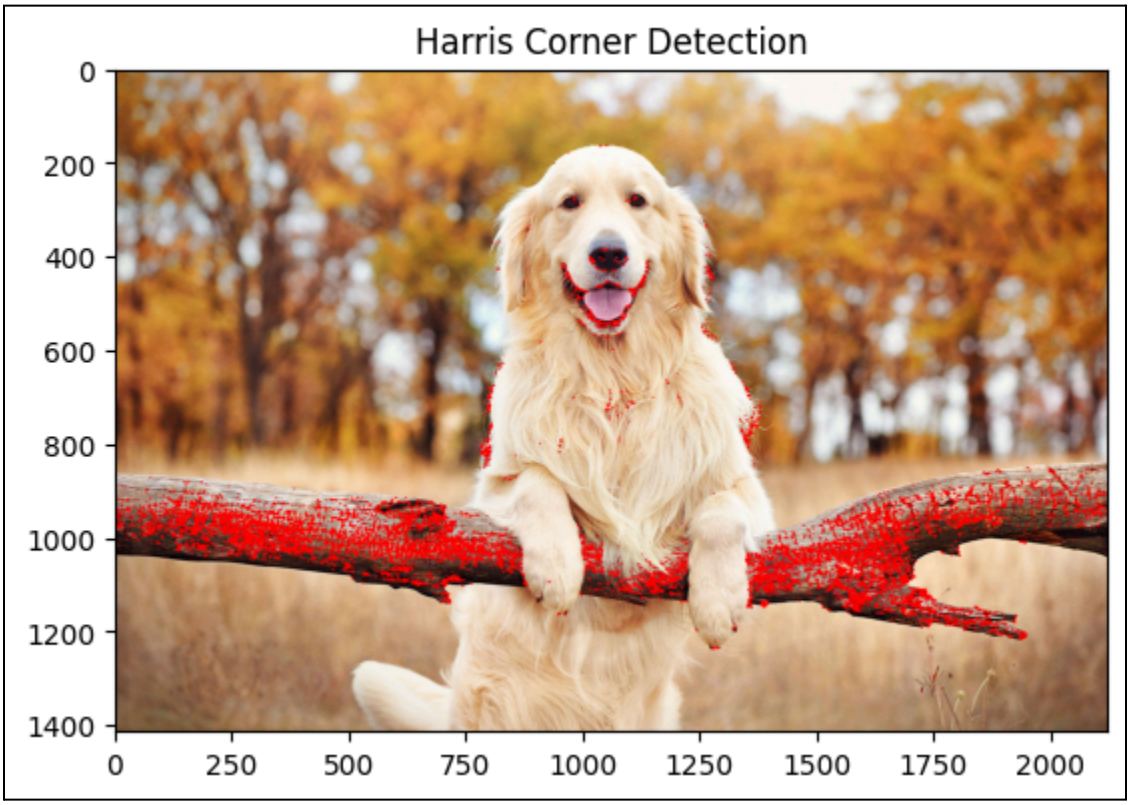
Code:

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

```
# Load the image
img = cv2.imread('dog.jpg')
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()
```

Output:



Explanation:

The Harris Corner Detection technique identifies corners in an image, where there are sudden changes in intensity in all directions. It works by converting the grayscale image to a floating-point format and detecting the points where the image gradients differ significantly. The detected corners are then enhanced by dilating them, making them more visible. Finally, the corners are marked in red on the original image to highlight areas of interest, showing where unique edges meet or intersect.

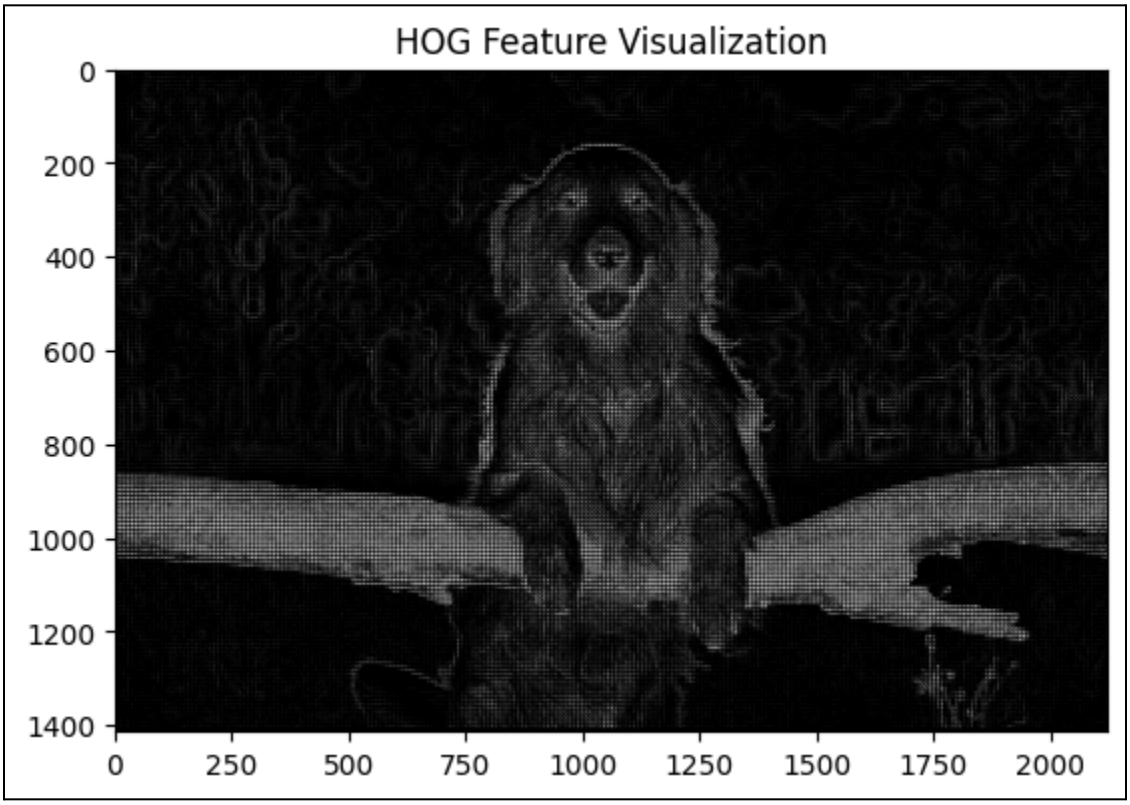
Task 2: HOG (Histogram of Oriented Gradients) Feature Extraction

Code:

```
# Load Image
```

```
img = cv2.imread('dog.jpg')
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()
```

Output:



Explanation:

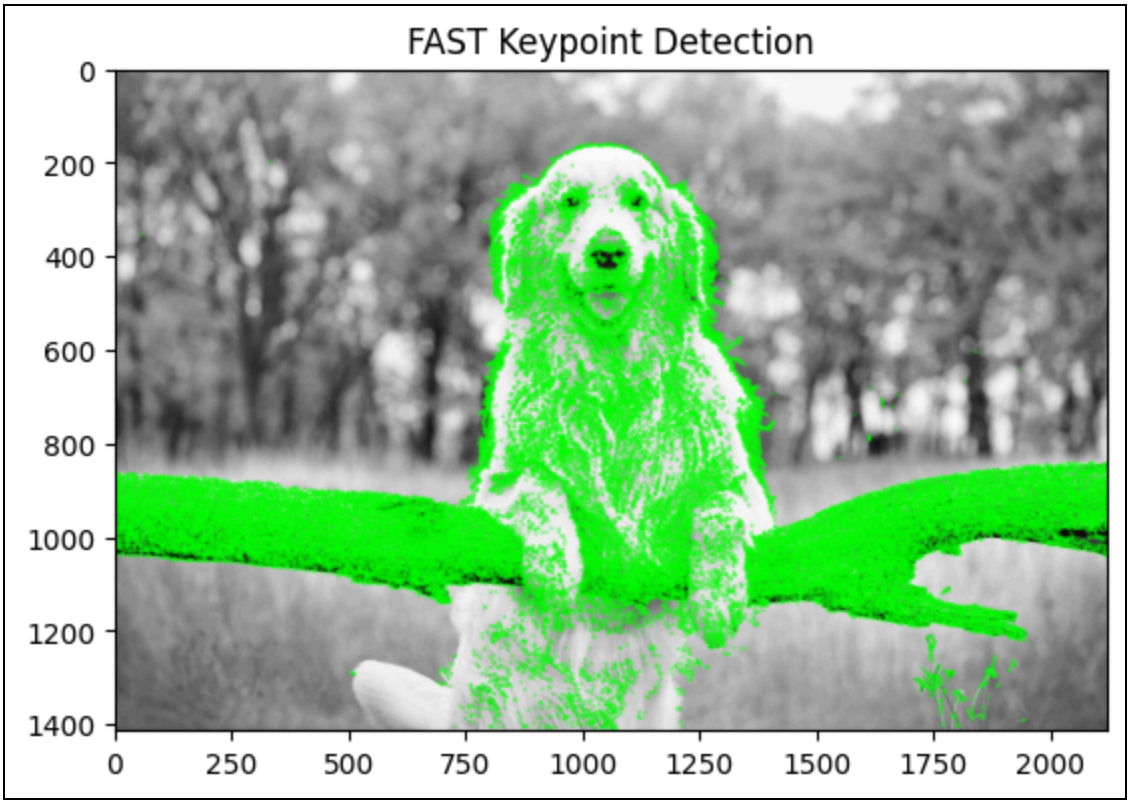
The HOG feature extraction technique helps capture the shapes and textures in an image by analyzing gradient orientations. First, the image is converted to grayscale. Then, gradients are calculated within small cells, and the orientations are grouped into bins. The HOG method creates a feature vector representing the gradient patterns, which is particularly helpful for object detection. The output includes a visual representation of the gradient directions, which can reveal the structural outline of objects within the image.

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

Code:

```
img = cv2.imread('dog.jpg')
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()
```

Output:



Explanation:

FAST (Features from Accelerated Segment Test) is a quick method for detecting keypoints in an image by comparing the intensity of each pixel with its surrounding pixels. This process detects points that stand out as unique features. The identified keypoints are marked on the grayscale image with green dots. FAST is particularly useful for its speed, allowing it to detect keypoints efficiently, even in images with many details.

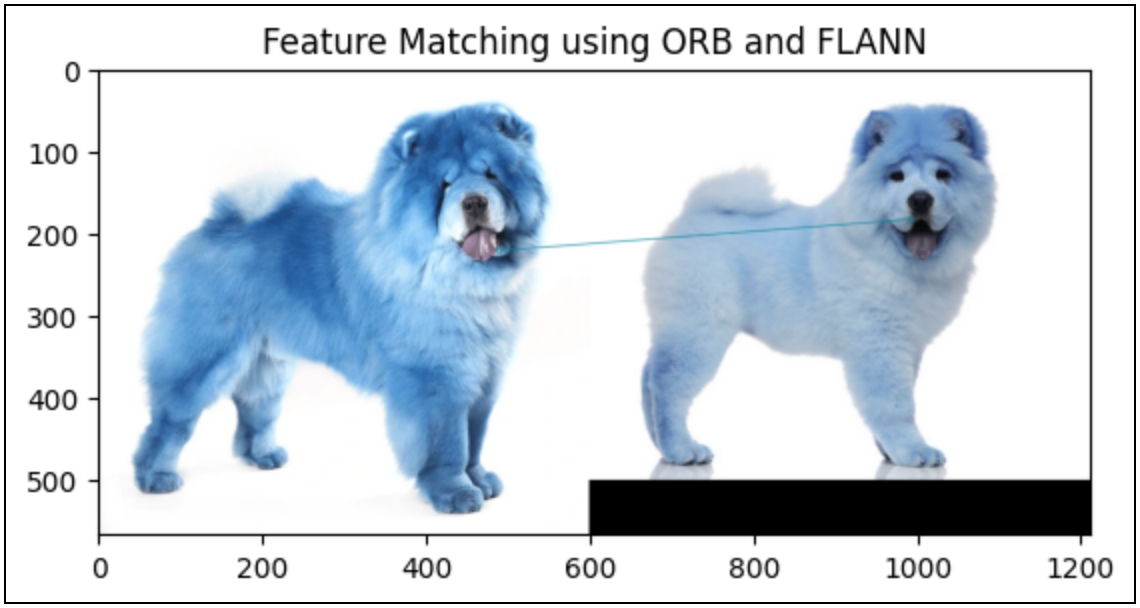
Task 4: Feature Matching using ORB and FLANN

Code:

```
# Load Images
img1 = cv2.imread('dog1.png')
img2 = cv2.imread('dog2.png')

# Convert images to grayscale
img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
```

```
img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

# ORB detector
orb = cv2.ORB_create()

# Find the keypoints and descriptors with ORB
kp1, des1 = orb.detectAndCompute(img1_gray, None)
kp2, des2 = orb.detectAndCompute(img2_gray, 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 descriptors
matches = flann.knnMatch(des1, des2, k=2)

# Apply ratio test with a check for two items in matches
good_matches = []
for match_pair in matches:
   if len(match_pair) == 2:  # Ensure there are two matches
       m, n = match_pair
       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=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)


# Display result
plt.imshow(result)
plt.title('Feature Matching using ORB and FLANN')
plt.show()
```

Output:



Explanation:

Feature matching involves finding common features between two images. The ORB (Oriented FAST and Rotated BRIEF) algorithm detects keypoints and computes their descriptors, which describe the features in each image. A FLANN (Fast Library for Approximate Nearest Neighbors) matcher then compares these descriptors to find matches between the images. Using a ratio test, only the best matches are retained, highlighting similar areas or objects between the two images. The result shows lines connecting matching features, making it easier to identify overlapping or related parts.

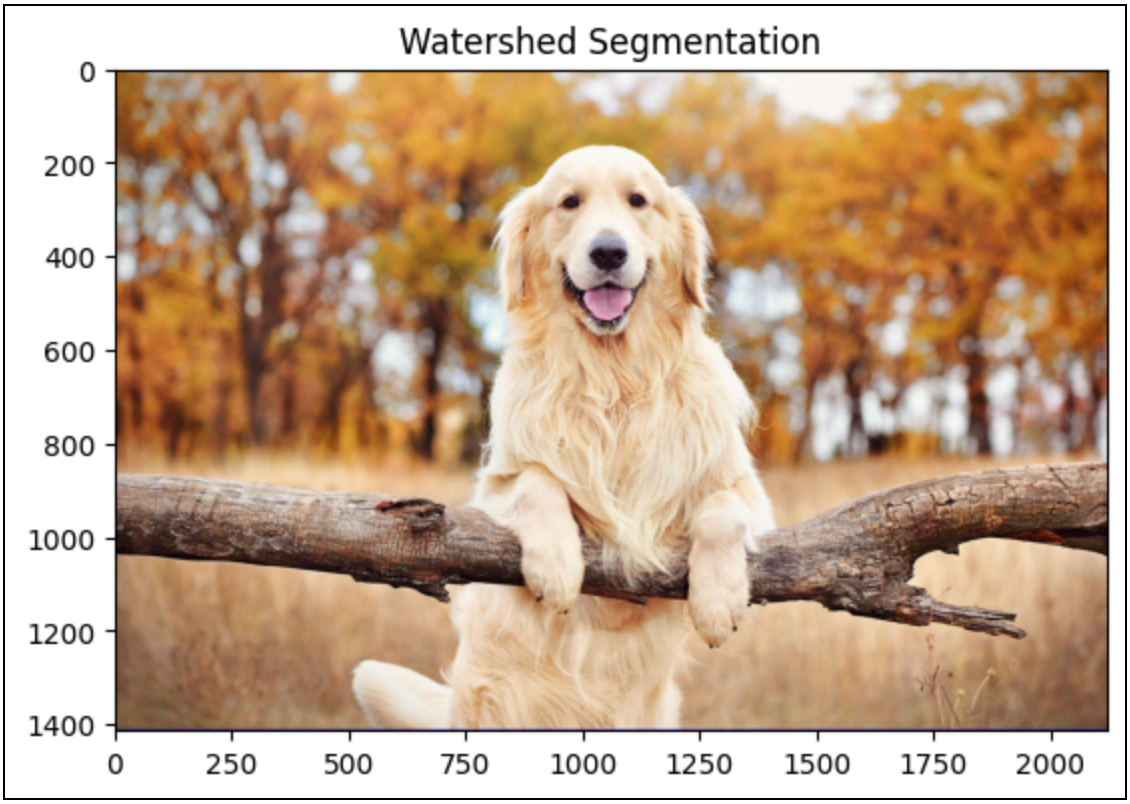
Task 5: Image Segmentation using Watershed Algorithm

Code:

```
# Load Image
img = cv2.imread('dog.jpg')
gray_image = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Apply threshold to convert the image to binary
ret , binary_image = cv2.threshold(gray_image, 0, 255,
cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

# Noise removal
kernel = np.ones((3, 3), np.uint8)
```

```
opening = cv2.morphologyEx(binary_image, cv2.MORPH_OPEN, kernel,
iterations=2)

# Sure background
sure_bg = cv2.dilate(opening, kernel, iterations=3)

# Finding sure foreground area
dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
ret, sure_fg = cv2.threshold(dist_transform, 0.7 * dist_transform.max(),
255, 0)

# Unknown region
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(sure_bg, sure_fg)

# Marker labelling
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
markers [unknown == 0] = 0

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

# Display the result
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.title('Watershed Segmentation')
plt.show()
```

Output:



Explanation:

The Watershed algorithm segments an image by marking regions based on intensity levels. Initially, the image is converted to a binary format, followed by a noise-removal step to improve clarity. The algorithm identifies "sure" background and foreground areas, and then marks regions where objects are likely to be present. Using these markers, it applies the Watershed algorithm to separate distinct regions. The boundaries are then highlighted in blue, providing a clear segmentation of objects within the image.

Observation:

These tasks demonstrate different ways to analyze and manipulate images by identifying corners, gradients, keypoints, matching features, and segmenting objects. Harris Corner Detection and FAST focus on detecting unique points, while HOG captures detailed textures. ORB and FLANN are used together for effective feature matching, making it possible to align or compare images. The Watershed algorithm effectively segments images, distinguishing different regions. Each technique offers a unique approach, making it useful for specific image analysis goals.

- *End of Documentation* -