Exercise 2 - Feature Extraction and Object Detection

This exercise will cover various methods of feature extraction:

- **SIFT Feature Extraction**
- **SURF Feature Extraction**
- **ORB Feature Extraction**

Also:

- **Feature Matching Using the SIFT Method**
- **Image Stitching using Homography**
- **Combining SIFT and ORB**

Task 1: SIFT Feature Extraction

SIFT (Scale-Invariant Feature Transform) detects important points, called key points, in an image. These key points represent distinct and unique features, such as corners or edges, that can be identified even if the image is resized, rotated, or transformed. SIFT generates a descriptor for each key point, which helps match these points across images.

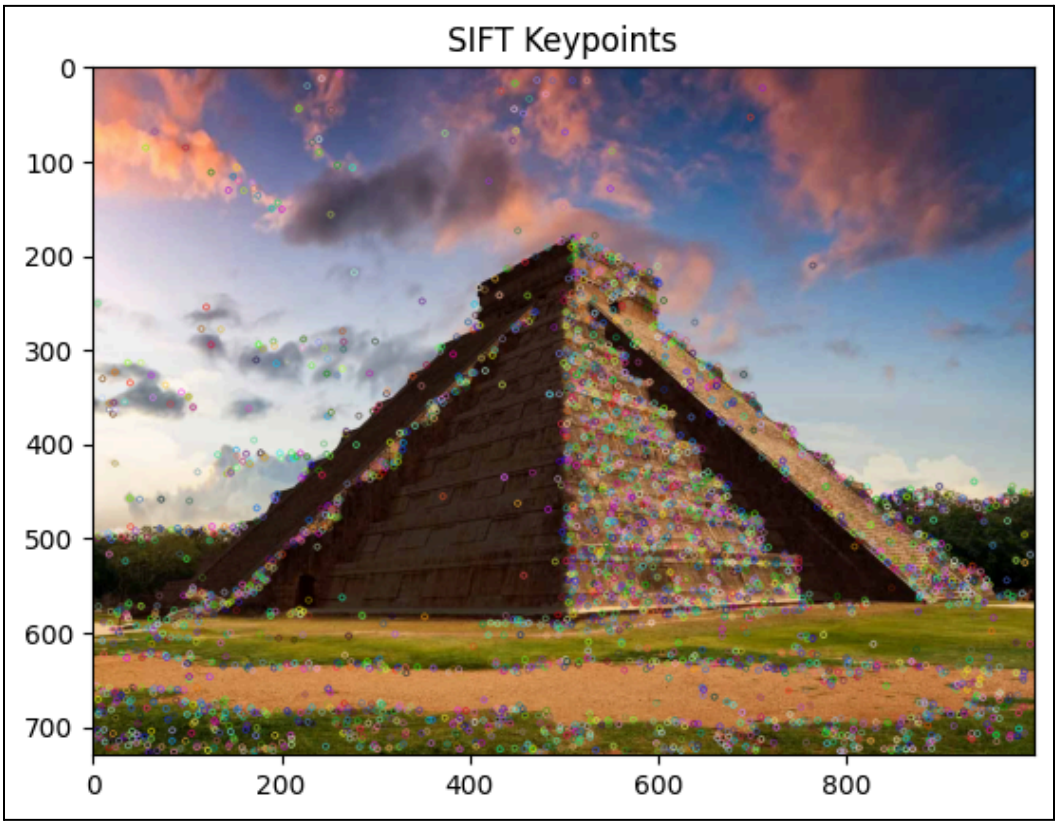
Code:

```
!pip install opencv-contrib-python
import cv2
import matplotlib.pyplot as plt
import numpy as np
```

```
#Load the Image
image = cv2.imread('mexico.png')
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

#Initialize SIFT detector
sift = cv2.SIFT_create()

#Detec keypoints and descriptors
keypoints, descriptors = sift.detectAndCompute(gray_image, None)
```

```
#Draw keypoints on the image
image_with_keypoints = cv2.drawKeypoints(image, keypoints, None)

#Display the image with keypoints
plt.imshow(cv2.cvtColor(image_with_keypoints, cv2.COLOR_BGR2RGB))
plt.title('SIFT Keypoints')
plt.show()
```

Output:



Explanation:

In this section, the code begins by installing the necessary OpenCV library that includes SIFT features. It reads an image, converts it to grayscale (which is essential for feature extraction), and initializes the SIFT detector. The SIFT algorithm detects keypoints and computes their descriptors, which are unique feature vectors that describe the keypoints. The code then draws these keypoints on the original image and displays the result using Matplotlib. This visualization helps to understand where the keypoints are located in the image, emphasizing the areas that are most distinctive.

Task 2: SURF Feature Extraction

SURF (Speeded-Up Robust Features) is similar to SIFT but is optimized for speed. SURF focuses on finding features faster, making it useful for real-time applications. It also detects keypoints and generates descriptors but uses a different mathematical approach to SIFT.

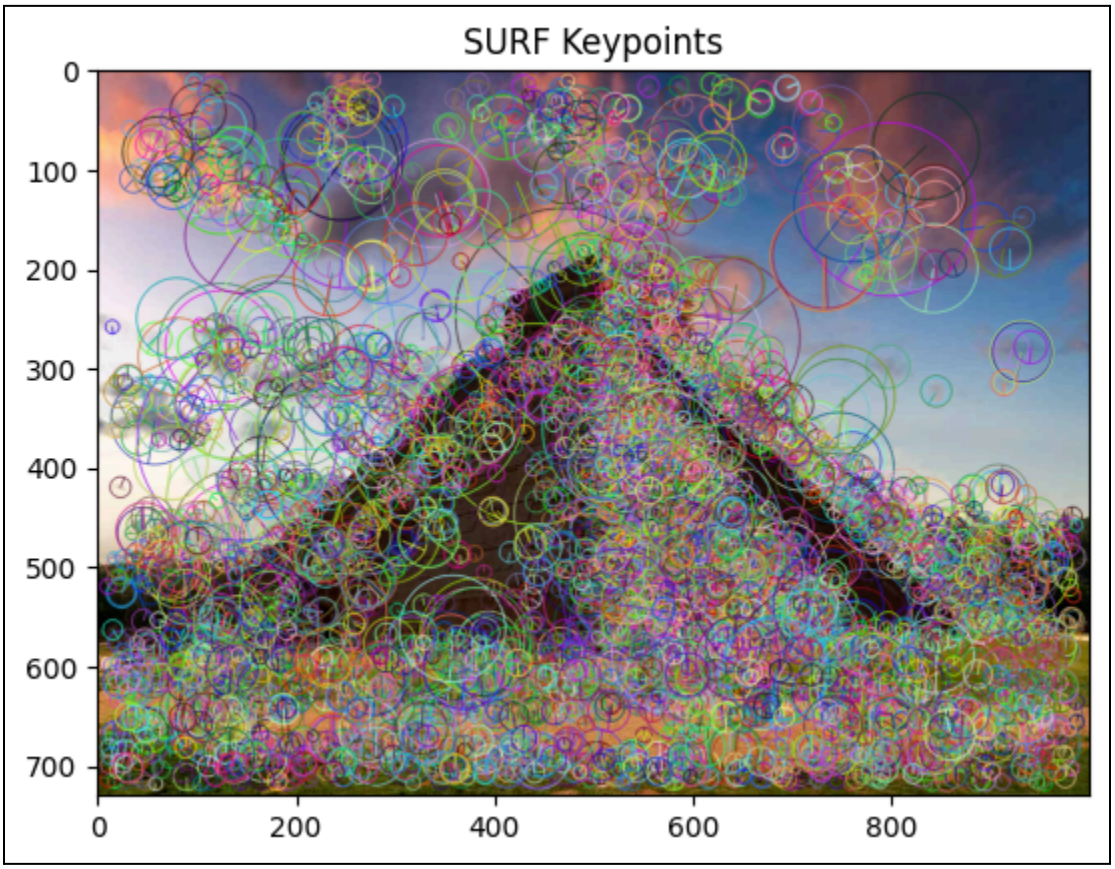
Code:

```
!apt-get update
!apt-get install -y cmake libjpeg-dev libpng-dev libtiff-dev
!apt-get install -y libavcodec-dev libavformat-dev libswscale-dev
!apt-get install -y libv4l-dev libxvidcore-dev libx264-dev
!apt-get install -y libgtk2.0-dev libatlas-base-dev gfortran
!apt-get install -y python3-dev

# Clone OpenCV and OpenCV Contrib
!git clone https://github.com/opencv/opencv.git
!git clone https://github.com/opencv/opencv_contrib.git
```

```
import cv2
try:
   surf = cv2.xfeatures2d.SURF_create()
   print("SURF is enabled and ready to use!")
except AttributeError:
   print("SURF is not available.")
Output:
SURF is enabled and ready to use!
#Load Image
image = cv2.imread('mexico.png')
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

#Initialize SURF detector
surf = cv2.xfeatures2d.SURF_create()

#Detect keypints and descriptors
keypoints, descriptors = surf.detectAndCompute(gray_image, None)

# Draw keypoints on the image
```

```
image_with_keypoints = cv2.drawKeypoints(image, keypoints, None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

# Convert BGR to RGB for displaying in matplotlib
plt.imshow(cv2.cvtColor(image_with_keypoints, cv2.COLOR_BGR2RGB))
plt.title('SURF Keypoints')
plt.show()
```

Output:



Explanation:

Similar to the SIFT method, this focuses on SURF (Speeded Up Robust Features). The code first installs the required packages and checks for the availability of the SURF detector. Once confirmed, it reads the same image and converts it to grayscale. The SURF detector is initialized, and it identifies keypoints and their descriptors, which can be used for matching features. The keypoints are drawn on the image, showing their locations, and the image is displayed. This task allows a comparison of the features

extracted by SURF versus SIFT, highlighting the differences in how these algorithms identify keypoints.

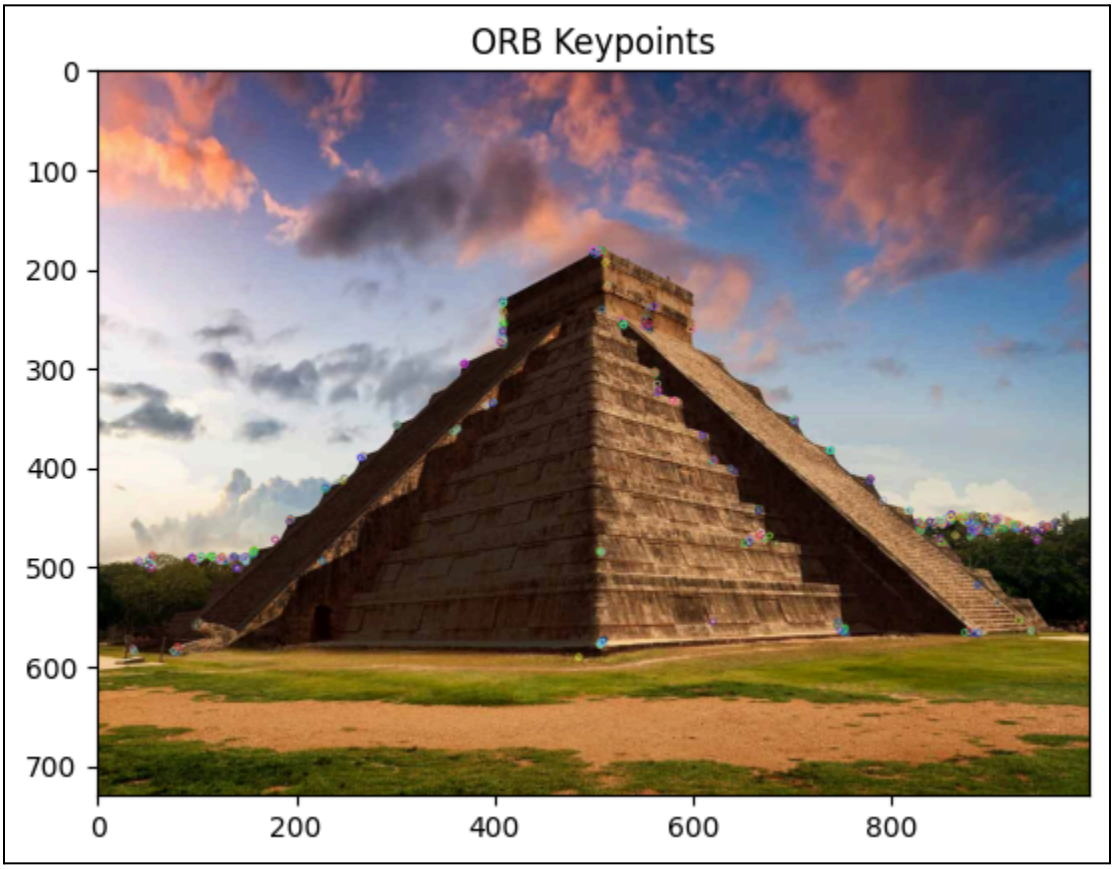
Task 3: ORB Feature Extraction

ORB (Oriented FAST and Rotated BRIEF) is a feature detection algorithm that is both fast and computationally less expensive than SIFT and SURF. It is ideal for real-time applications, particularly in mobile devices. ORB combines two methods: FAST (Features from Accelerated Segment Test) to detect keypoints and BRIEF (Binary Robust Independent Elementary Features) to compute descriptors.

Code:

```
#Load Image
image = cv2.imread('mexico.png')
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

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

#Detect keypints and descriptors
keypoints, descriptors = orb.detectAndCompute(gray_image, None)

##Draw keypoints on the image
image_with_keypoints = cv2.drawKeypoints(image, keypoints, None)

#Display the image with keypoints
plt.imshow(cv2.cvtColor(image_with_keypoints, cv2.COLOR_BGR2RGB))
plt.title('ORB Keypoints')
plt.show()
```

Output:



Explanation:

The third task employs the ORB (Oriented FAST and Rotated BRIEF) method for feature extraction. Like the previous tasks, the image is loaded and converted to grayscale. The ORB detector is initialized, and it identifies keypoints and descriptors. After processing, the keypoints are visualized on the original image. ORB is a fast alternative to SIFT and SURF, which makes it suitable for real-time applications. The results illustrate how ORB captures keypoints effectively, although with potentially fewer keypoints than SIFT or SURF due to its speed-optimized approach.

Task 4: Feature Matching

Feature matching is used to find similar points between two images. After detecting keypoints using SIFT, the algorithm uses a Brute-Force Matcher to find matching keypoints between two images. The matcher compares the descriptors of the keypoints and finds pairs that are similar. In the code, we load two images, detect their

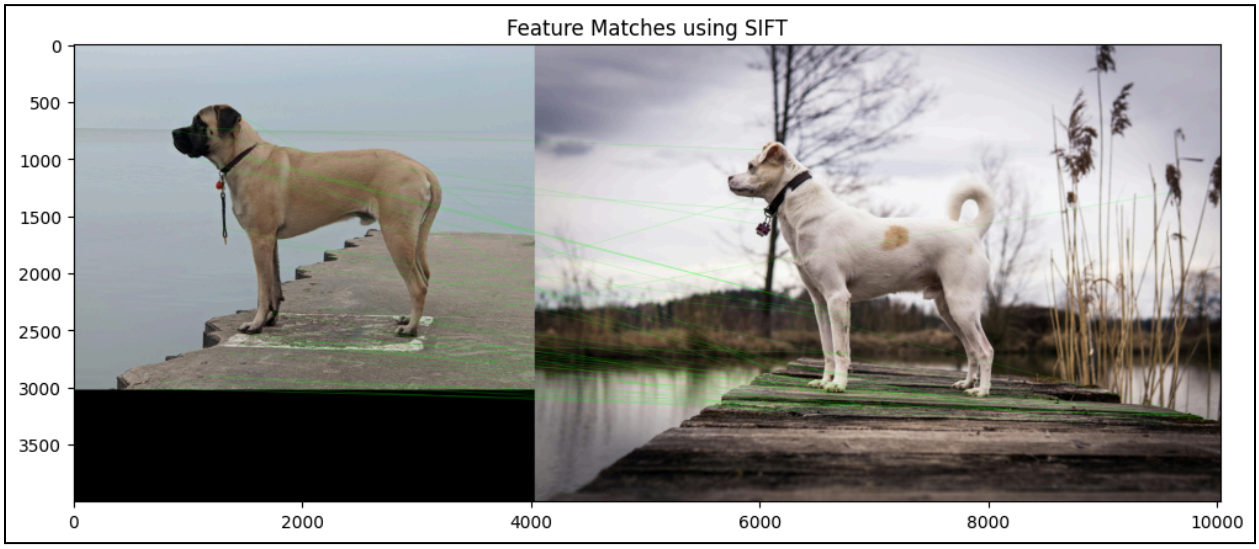
keypoints and descriptors using SIFT, and then use the matcher to draw lines between matching keypoints. The lines show which points in the first image correspond to points in the second image.

Code:

```
# Load two images
image1 = cv2.imread('dog1.jpg')
image2 = cv2.imread('dog2.jpg')

# Convert images to grayscale
gray1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

# Initialize SIFT detector
sift = cv2.SIFT_create()

# Detect keypoints and descriptors
keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

# FLANN parameters
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)  # The number of checks

# Initialize the FLANN-based matcher
flann = cv2.FlannBasedMatcher(index_params, search_params)

# Perform the matching between descriptors
matches = flann.knnMatch(descriptors1, descriptors2, k=2)

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

# Draw the matches with thicker, colored lines
matched_image = cv2.drawMatches(
```

```
    image1, keypoints1,
    image2, keypoints2,
    good_matches, None,
    matchColor=(0, 260, 0),   # Green color for matches
    singlePointColor=(270, 0, 0),   # Blue color for keypoints
    flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the image with matches
plt.figure(figsize=(12, 6))
plt.imshow(cv2.cvtColor(matched_image, cv2.COLOR_BGR2RGB))
plt.title('Feature Matches using SIFT')
plt.show()
```

Output:



Explanation:

In this task, the code matches features between two images (dog1 and dog2) using the SIFT algorithm. The images are loaded and converted to grayscale before keypoints and descriptors are extracted. The code uses the FLANN (Fast Library for Approximate Nearest Neighbors) matcher to perform matching between the descriptors of both images. A ratio test is applied to filter out poor matches, retaining only the good ones. The matched features are visualized, demonstrating how well the features from the two images correspond to each other. This task highlights the effectiveness of SIFT in establishing correspondences between images.

Task 5: Application of Feature Matching | Image Stitching using HOMOGRAPHY

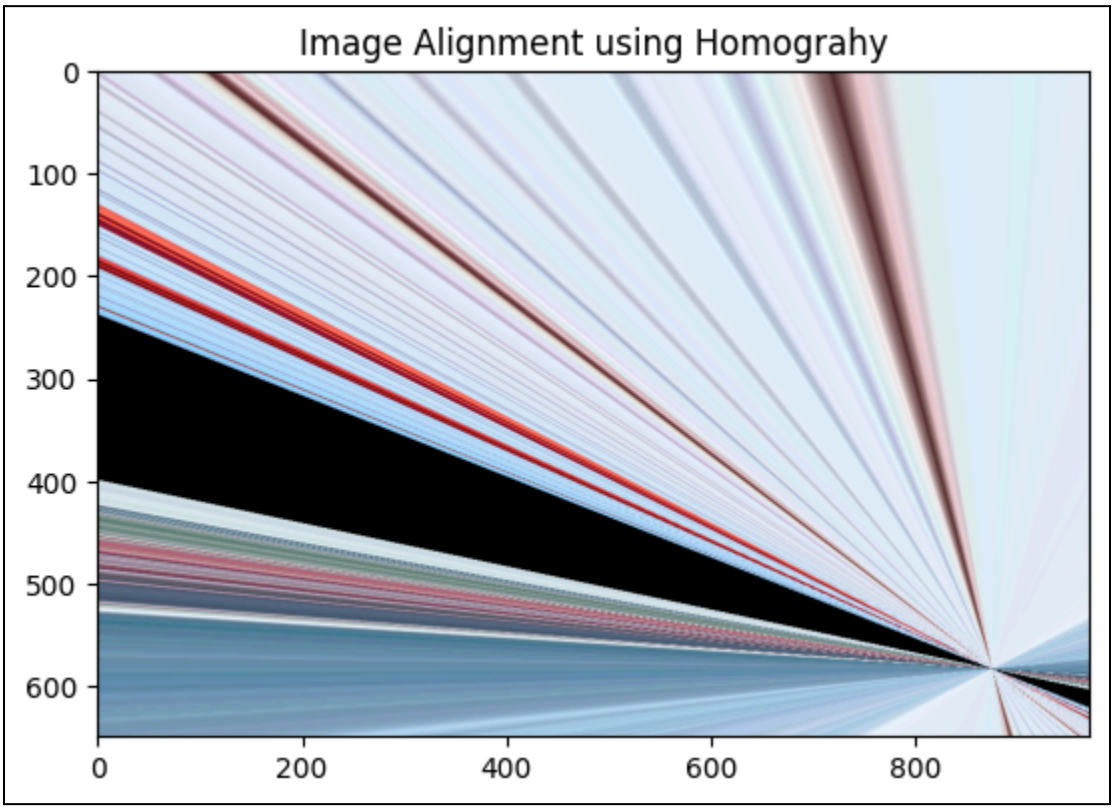
Homography is a mathematical transformation that maps points from one image to another, which is useful for aligning images taken from different angles or perspectives. This process is used in image stitching (e.g., creating panoramas), where you align and merge images to form a larger one. The code uses the keypoints matched between two images and calculates the homography matrix. This matrix is then used to warp one image to align it with the other.

Code:

```
#Loading two images
image1 = cv2.imread('bridge1.jpeg')
image2 = cv2.imread('bridge2.jpg')

#Convert to grayscale
gray1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

#Detect keypoints and descriptors using SIFT
sift = cv2.SIFT_create()
keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

#Match features using BFMatcher
bf = cv2.BFMatcher(cv2.NORM_L2)
matches = bf.knnMatch(descriptors1, descriptors2, k=2)

#Apply ratio test (Lowe's ratio test)
good_matches = []
for m, n in matches:
   if m.distance < 0.75 * n.distance:
       good_matches.append(m)

   #Extract location of good matches
src_pts     =     np.float32([keypoints1[m.queryIdx].pt     for     m     in
good_matches]).reshape(-1, 1, 2)
dst_pts     =     np.float32([keypoints2[m.trainIdx].pt     for     m     in
good_matches]).reshape(-1, 1, 2)
```

```
   #Find homography matrix
   #Corrected function name from findingHomography to findHomography
M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

   #Warp one image to align with the other
h,w, _ = image1.shape
result = cv2.warpPerspective(image1, M, (w, h))

   #Display the result
plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
plt.title('Image Alignment using Homograhy')
plt.show()
```

Output:



Explanation:

This task shows a practical application of feature matching by stitching two images together. After loading and converting the images to grayscale, keypoints and

descriptors are extracted using SIFT. The code matches these features using the BFMatcher (Brute-Force Matcher). The good matches are filtered using a ratio test, and the source and destination points for the homography matrix are extracted. The homography matrix is computed, allowing one image to be warped and aligned with the other. The resulting stitched image is displayed, illustrating how feature matching can facilitate seamless image alignment and blending.

Task 6: Combining Feature Extraction Methods | ORB and SIFT Feature Extraction

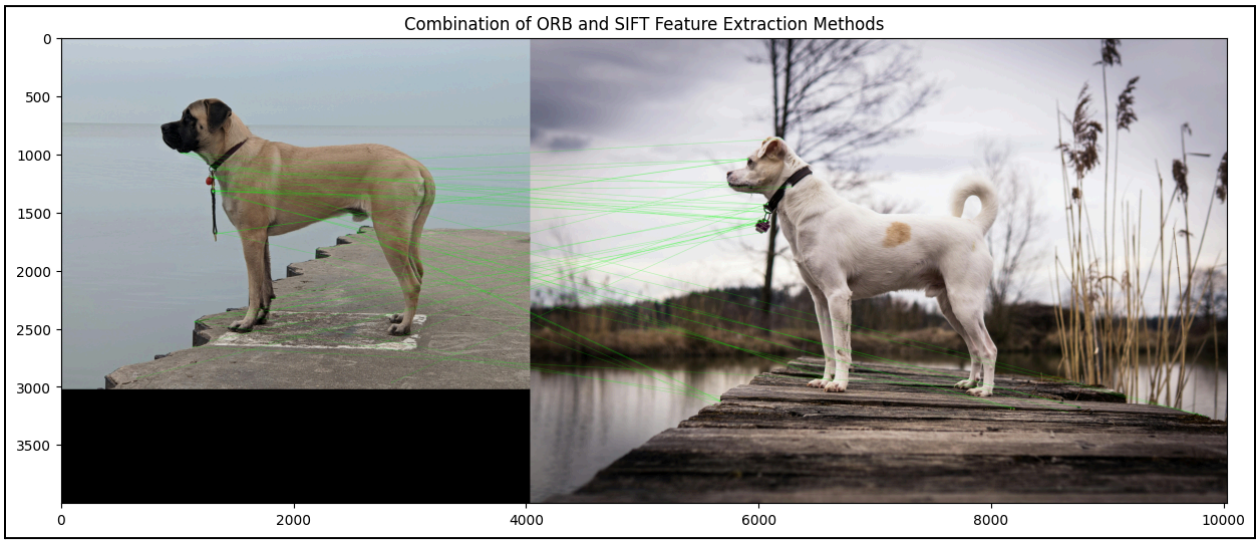
By combining two feature extraction methods (SIFT and ORB), you can take advantage of the strengths of both. For example, SIFT is more accurate, but ORB is faster. By detecting keypoints using both methods, you can compare how they perform on different types of images and possibly combine their outputs for more strong feature detection and matching.

Code:

```
# Load Images
image1 = cv2.imread('dog1.jpg')
image2 = cv2.imread('dog2.jpg')

# SIFT detector
sift = cv2.SIFT_create()
keypoints_sift1, descriptors_sift1 = sift.detectAndCompute(image1, None)
keypoints_sift2, descriptors_sift2 = sift.detectAndCompute(image2, None)

# ORB detector
orb = cv2.ORB_create()
keypoints_orb1, descriptors_orb1 = orb.detectAndCompute(image1, None)
keypoints_orb2, descriptors_orb2 = orb.detectAndCompute(image2, None)

# Cross-Check Matching (BFMatcher with crossCheck=True)
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(descriptors_orb1, descriptors_orb2)

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

# Draw matches with thicker, bolder lines
```

```
match_img = cv2.drawMatches(
    image1, keypoints_orb1, image2, keypoints_orb2, matches[:50], None,
    matchColor=(0, 255, 0),  # Green color for matches
    singlePointColor=(255, 0, 0),  # Red color for keypoints
    flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
)

# Display the matches
plt.figure(figsize=(15, 10))
plt.imshow(cv2.cvtColor(match_img, cv2.COLOR_BGR2RGB))
plt.title('Combination of ORB and SIFT Feature Extraction Methods')
plt.show()
```

Output:



Explanation:

The final task combines the results of SIFT and ORB for feature extraction and matching between two dog images. Keypoints and descriptors are extracted using both methods, and matches are found using the BFMatcher with cross-checking enabled. This approach ensures that only consistent matches between the two sets of descriptors are retained. The matches are sorted by distance, and the top matches are visualized. This combination demonstrates how using multiple feature extraction techniques can enhance the overall matching process by manipulating the strengths of each method.

Observations

Throughout the exercises, it is observed that SIFT and SURF provide the best feature extraction with good invariance to scaling and rotation, making them suitable for complex tasks. ORB, while faster, may not capture as many distinctive features but excels in real-time scenarios. The application of feature-matching techniques, especially in image stitching, illustrates the power of these algorithms in practical image-processing tasks. Each technique has its strengths and weaknesses, and the choice of which to use depends on the specific requirements of the task, such as speed versus accuracy.

- End of Documentation -