Machine Problem 3 - Feature Extraction and Object Detection

Objective:

The objective of this machine problem is to implement and compare the three feature extraction methods (SIFT, SURF, and ORB) in a single task. You will use these methods for feature matching between two images, then perform image alignment using Homography to warp one image onto the other.

Problem Description:

1. You are tasked with loading two images and performing the following steps:
2. Extract keypoints and descriptors from both images using SIFT, SURF, and ORB.
3. Perform feature matching between the two images using both Brute-Force Matcher and FLANN Matcher.
4. Use the matched keypoints to calculate a Homography matrix and align the two images.
5. Compare the performance of SIFT, SURF, and ORB in terms of feature matching accuracy and speed.

Important libraries and repo

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
from google.colab import files
from PIL import Image
```

Cloning OpenCV repo to enable SURF feature extraction

```
# Install dependencies required for building OpenCV
!apt update && apt install -y build-essential cmake git libgtk2.0-dev
pkg-config libavcodec-dev libavformat-dev libswscale-dev

# Clone the OpenCV and OpenCV Contrib repositories
!rm -rf opencv opencv_contrib  # Clear any previous installations
!git clone https://github.com/opencv/opencv.git
!git clone https://github.com/opencv/opencv_contrib.git
```

```
# Create a build directory and navigate to it
!mkdir -p opencv/build
%cd opencv/build

# Run CMake configuration with OPENCV_ENABLE_NONFREE enabled
!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=OFF ..

# Compile OpenCV (this will take a while)
!make -j8

# Install the compiled OpenCV library
!make install
!ldconfig
```

Explanation:

This code installs all necessary dependencies, clones the OpenCV repositories, configures the build, compiles the library, and installs it on the system.

Step 1: Load Images

Code:

```
# Load the images
img1_path = '/content/dog1.png'
img2_path = '/content/dog2.png'

img1 = cv2.imread(img1_path, cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread(img2_path, cv2.IMREAD_GRAYSCALE)
```

Explanation:

The code sets the file paths for two images, dog1.png and dog2.png. It then uses the cv2.imread() function to read these images from the specified paths. By using the option cv2.IMREAD_GRAYSCALE, it ensures that the images are loaded in grayscale, which means they will be in black and white instead of color. In summary, this code loads two images as grayscale images for further processing.

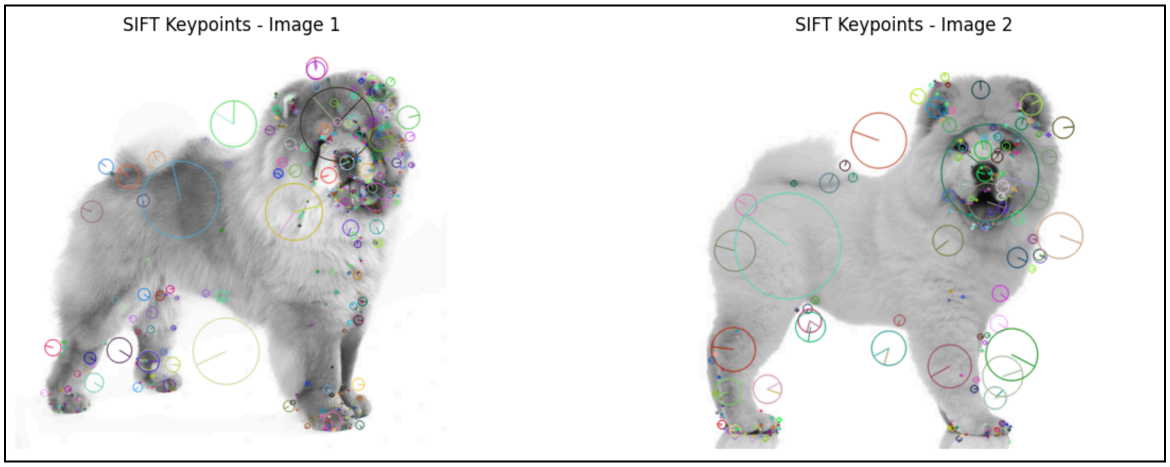
Step 2: Extract Keypoints and Descriptors Using SIFT, SURF, and ORB

- **SIFT FEATURE EXTRACTION**

Code:

```
# SIFT Feature Extraction
sift = cv2.SIFT_create()
kp1_sift, des1_sift = sift.detectAndCompute(img1, None)
kp2_sift, des2_sift = sift.detectAndCompute(img2, None)

# Print the number of keypoints detected for SIFT and display keypoints
print(f"SIFT - Keypoints in Image 1: {len(kp1_sift)}, Keypoints in Image
2: {len(kp2_sift)}")

# Display SIFT Keypoints
img1_sift       =       cv2.drawKeypoints(img1,       kp1_sift,       None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
img2_sift       =       cv2.drawKeypoints(img2,       kp2_sift,       None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.imshow(img1_sift, cmap='gray')
plt.title("SIFT Keypoints - Image 1")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(img2_sift, cmap='gray')
plt.title("SIFT Keypoints - Image 2")
plt.axis('off')

plt.show()
```

Output:



Explanation:

This code performs **SIFT (Scale-Invariant Feature Transform**) feature extraction on two grayscale images.

First, it creates a SIFT object using cv2.SIFT_create(). Then, it detects keypoints and computes their descriptors for both images using sift.detectAndCompute(). Keypoints are specific points of interest in the images that can be used for matching and analysis.

Next, the code prints the number of keypoints found in each image. After that, it visualizes the detected keypoints by drawing them on the original images using cv2.drawKeypoints().

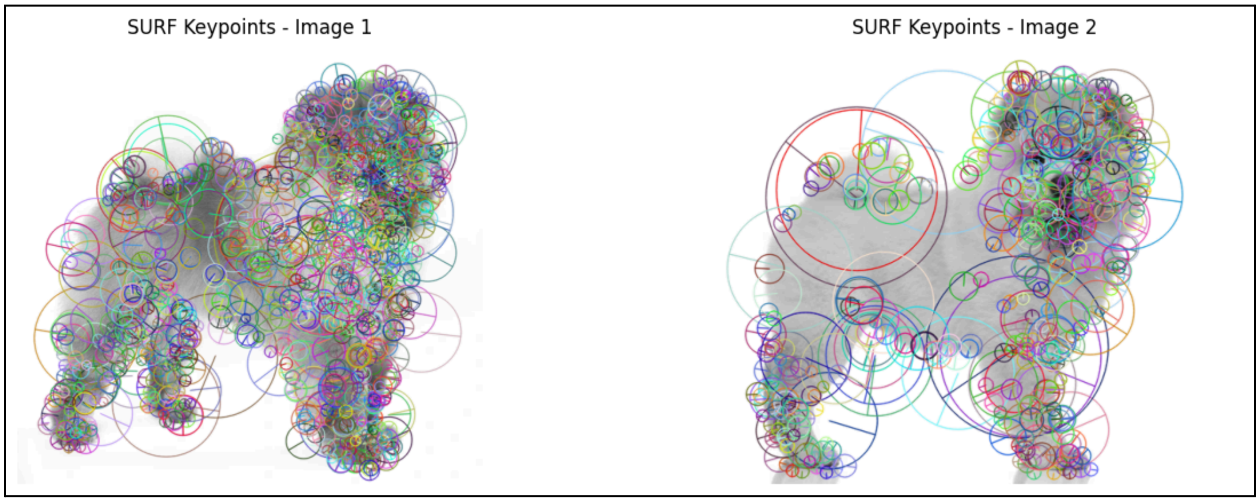
Finally, it displays both images with their keypoints in a side-by-side format using matplotlib. Each subplot shows one image with its detected SIFT keypoints highlighted, and the axes are turned off for a cleaner look.

- **SURF FEATURE EXTRACTION**

Code:

```
# SURF Feature Extraction
surf = cv2.xfeatures2d.SURF_create()
kp1_surf, des1_surf = surf.detectAndCompute(img1, None)
kp2_surf, des2_surf = surf.detectAndCompute(img2, None)


# Print the number of keypoints detected for SURF and display keypoints
print(f"SURF - Keypoints in Image 1: {len(kp1_surf)}, Keypoints in Image 2: {len(kp2_surf)}")


# Display SURF Keypoints
img1_surf = cv2.drawKeypoints(img1, kp1_surf, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
img2_surf = cv2.drawKeypoints(img2, kp2_surf, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.imshow(img1_surf, cmap='gray')
plt.title("SURF Keypoints - Image 1")
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(img2_surf, cmap='gray')
plt.title("SURF Keypoints - Image 2")
plt.axis('off')

plt.show()
```

Output:



Explanation:

This code snippet performs **SURF (Speeded-Up Robust Features**) feature extraction on two grayscale images. It initializes a SURF detector and uses the detectAndCompute() method to identify keypoints and compute their descriptors for both images. The number of detected keypoints is printed to the console. To visualize the results, the code uses cv2.drawKeypoints() to highlight the keypoints on each image. Finally, it displays the images side by side using matplotlib, providing a clear comparison of the keypoint distributions while keeping the axes off for a cleaner look. Overall, the code effectively extracts and visualizes SURF features from the images.

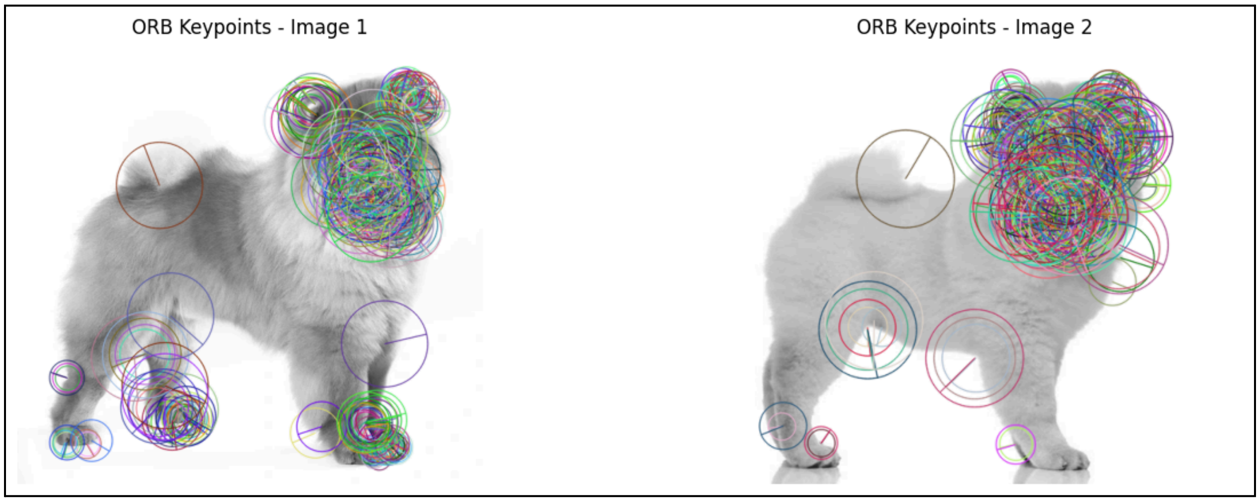
- **ORB FEATURE EXTRACTION**

Code:

```
# ORB Feature Extraction
orb = cv2.ORB_create()
kp1_orb, des1_orb = orb.detectAndCompute(img1, None)
kp2_orb, des2_orb = orb.detectAndCompute(img2, None)
```

```
# Print the number of keypoints detected for ORB and display keypoints
print(f"ORB - Keypoints in Image 1: {len(kp1_orb)}, Keypoints in Image 2:
{len(kp2_orb)}")

# Display ORB Keypoints
img1_orb        =        cv2.drawKeypoints(img1,        kp1_orb,        None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
img2_orb        =        cv2.drawKeypoints(img2,        kp2_orb,        None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.imshow(img1_orb, cmap='gray')
plt.title("ORB Keypoints - Image 1")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(img2_orb, cmap='gray')
plt.title("ORB Keypoints - Image 2")
plt.axis('off')

plt.show()
```

Output:



Explanation:

ORB (Oriented FAST and Rotated BRIEF) feature extraction on two grayscale images is what this code snippet is all about. It initializes the ORB detector and uses the detectAndCompute() method to find keypoints and their descriptors for both images. The number of detected keypoints is printed to the console. To visualize the results, the code highlights the keypoints on each image using cv2.drawKeypoints() and displays them side by side with matplotlib. This allows for a clear comparison of the ORB keypoints in the two images.

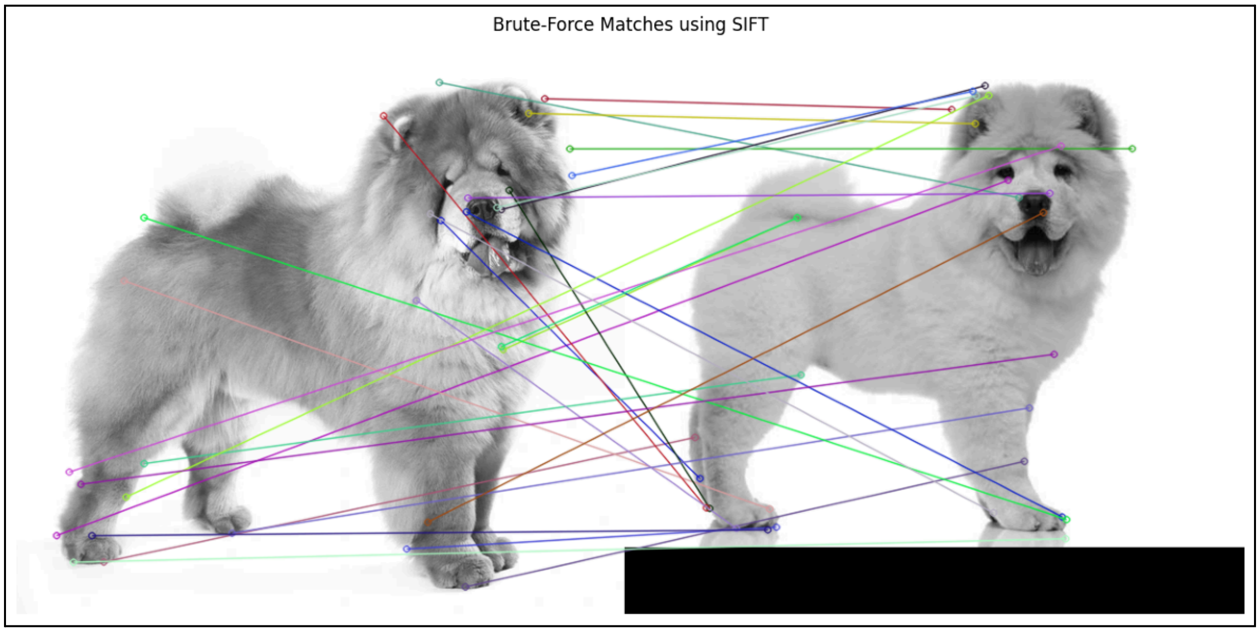
Step 3: Feature Matching with Brute-Force and FLANN

- **Feature Matching with Brute-Force Matcher**

Code:

```
# Brute-Force Matcher
# Using SIFT descriptors for matching
bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
matches_sift = bf.match(des1_sift, des2_sift)

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

# Draw matches
img_matches_sift  =  cv2.drawMatches(img1,  kp1_sift,  img2,  kp2_sift,
matches_sift[:30],                                                None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the matches
plt.figure(figsize=(15, 10))
plt.imshow(img_matches_sift)
plt.title("Brute-Force Matches using SIFT")
plt.axis('off')
plt.show()
```

Output:



[redacted]

Explanation:

This code snippet implements a **Brute-Force Matcher** to find correspondences between keypoints in two images using SIFT (Scale-Invariant Feature Transform) descriptors. It initializes the matcher with cv2.BFMatcher() using L2 norm and cross-checking for more reliable matches. The match() method is then called to find matches between the SIFT descriptors of the two images.

The matches are sorted by distance, allowing the closest matches to be prioritized. The top 30 matches are then visualized using cv2.drawMatches(), which draws lines between corresponding keypoints in the two images. Finally, the matched image is displayed using matplotlib, providing a clear visual representation of the matched features. This approach effectively demonstrates the matching of keypoints between the two images based on their SIFT descriptors.

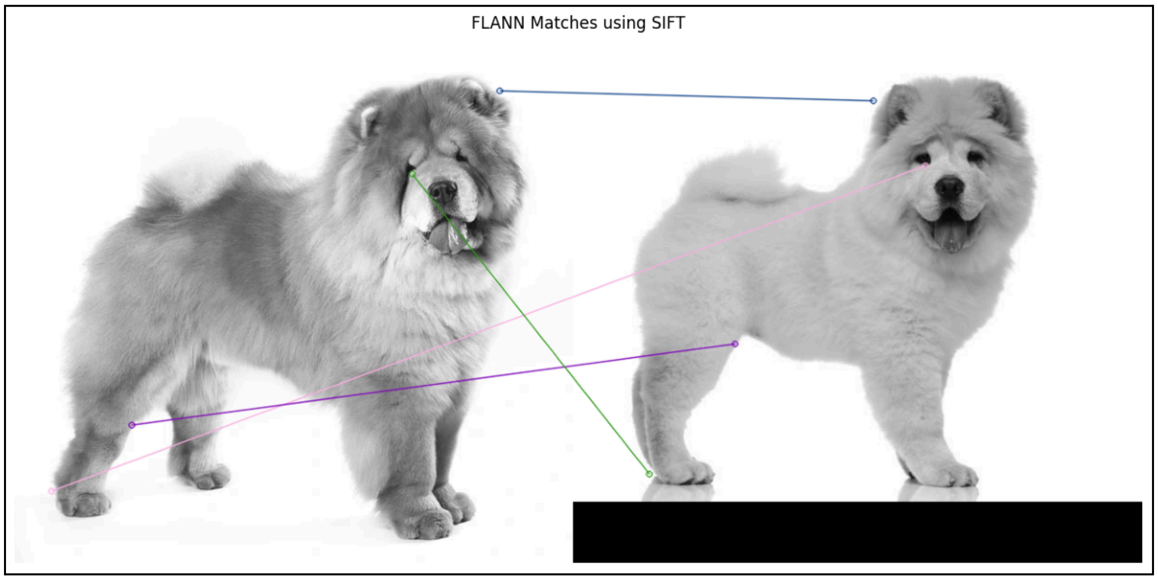
- **Feature Matching with FLANN Matcher**

Code:

```
# FLANN Matcher
# Prepare FLANN parameters
```

```
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)  # or pass empty dictionary

# Using SIFT descriptors for matching
flann = cv2.FlannBasedMatcher(index_params, search_params)
matches_flann = flann.knnMatch(des1_sift, des2_sift, k=2)

# Need to draw only good matches, so we need to draw first 'k' matches
good_matches_flann = []
for m, n in matches_flann:
   if m.distance < 0.7 * n.distance:
       good_matches_flann.append(m)

# Draw matches
img_matches_flann   =   cv2.drawMatches(img1,   kp1_sift,   img2,   kp2_sift,
good_matches_flann[:30],                                                 None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the matches
plt.figure(figsize=(15, 10))
plt.imshow(img_matches_flann)
plt.title("FLANN Matches using SIFT")
plt.axis('off')
plt.show()
```

Output:



Explanation:

This code snippet implements a **FLANN (Fast Library for Approximate Nearest Neighbors) Matcher** to find correspondences between keypoints in two images using SIFT (Scale-Invariant Feature Transform) descriptors. It prepares the FLANN parameters by specifying the KDTREE algorithm with five trees and sets the search parameters to check 50 times. A FlannBasedMatcher is created with these parameters, and the knnMatch method is used to find the two nearest neighbors for each descriptor in the first image. Matches are filtered to retain only those where the distance to the first nearest neighbor is less than 0.7 times that of the second nearest neighbor, effectively eliminating weak matches. The resulting good matches are visualized using cv2.drawMatches(), which connects corresponding keypoints in the two images, and the matched image is displayed with matplotlib, providing a clear visual representation of the matched features based on SIFT descriptors using the FLANN algorithm.

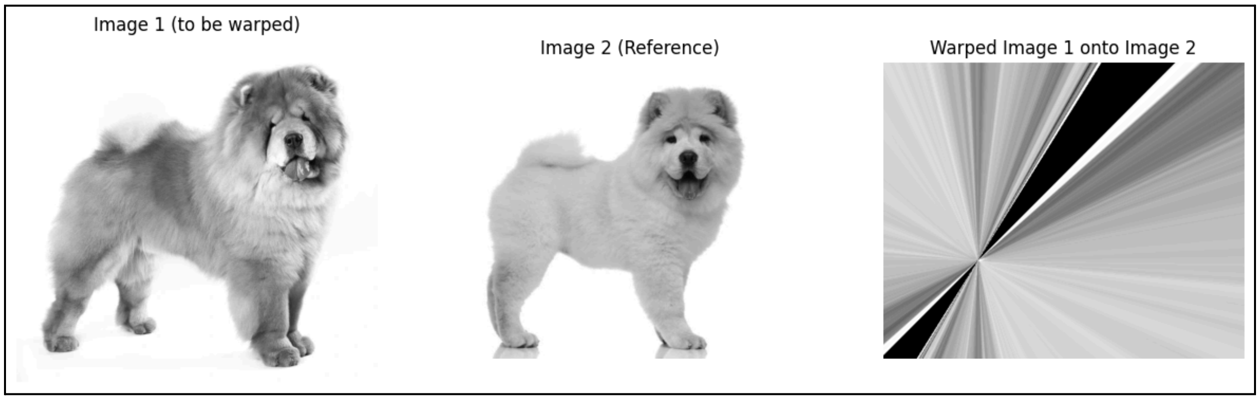
Step 4: Image Alignment Using Homography

Code:

```
# Check if there are enough good matches to compute a homography (at least
4 matches are needed)
if len(good_matches_flann) >= 4:
   # Extract the matched keypoints' coordinates
         src_pts = np.float32([kp1_sift[m.queryIdx].pt for m in
good_matches_flann]).reshape(-1, 1, 2)
         dst_pts = np.float32([kp2_sift[m.trainIdx].pt for m in
good_matches_flann]).reshape(-1, 1, 2)

   # Compute the homography matrix using RANSAC
   H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

   # If the homography is found, warp img1 onto img2's perspective
   if H is not None:
       # Warp img1 to img2's perspective using the computed homography
       height, width = img2.shape
       warped_image = cv2.warpPerspective(img1, H, (width, height))

       # Display the original, reference, and aligned images
```

```
        plt.figure(figsize=(15, 5))

        plt.subplot(1, 3, 1)
        plt.imshow(img1, cmap='gray')
        plt.title("Image 1 (to be warped)")
        plt.axis('off')

        plt.subplot(1, 3, 2)
        plt.imshow(img2, cmap='gray')
        plt.title("Image 2 (Reference)")
        plt.axis('off')

        plt.subplot(1, 3, 3)
        plt.imshow(warped_image, cmap='gray')
        plt.title("Warped Image 1 onto Image 2")
        plt.axis('off')

        plt.show()
    else:
        print("Homography could not be computed.")
else:
     print("Not enough good matches to compute a homography. At least 4
matches are required.")
```

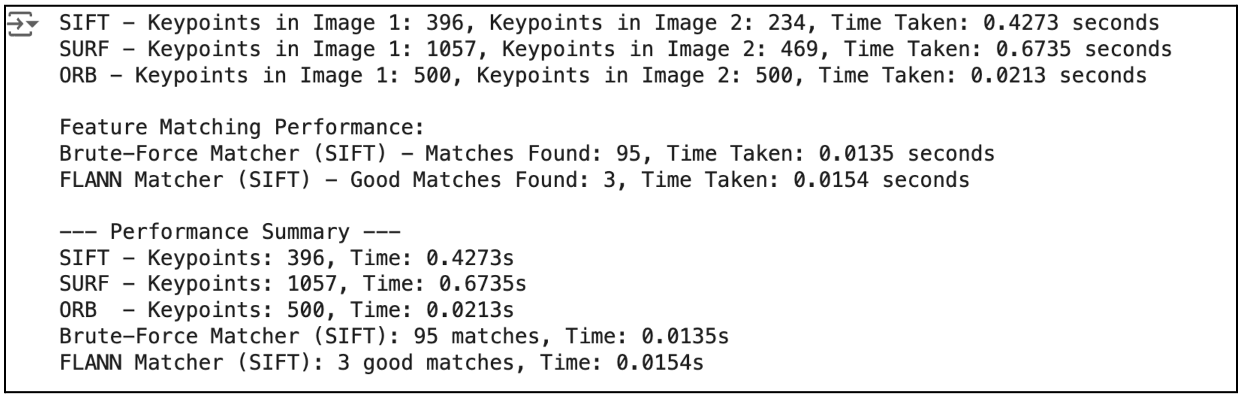
Output:



Explanation:

This code section checks if there are enough good matches (at least four) to compute a homography between two images using the matched keypoints from the FLANN Matcher. If sufficient matches are found, it extracts the coordinates of the matched keypoints and reshapes them for processing. The homography matrix is computed using the RANSAC algorithm, which is strong to outliers. If the homography is successfully computed, the first image is warped to align with the perspective of the second image using cv2.warpPerspective. The original image, the reference image, and the warped image are then displayed side by side using matplotlib for visual comparison. If the homography cannot be computed or there are not enough good matches, appropriate messages are printed to indicate the issue.

Performance Analysis Summary



```
SIFT - Keypoints in Image 1: 396, Keypoints in Image 2: 234, Time Taken: 0.4273 seconds
SURF - Keypoints in Image 1: 1057, Keypoints in Image 2: 469, Time Taken: 0.6735 seconds
ORB - Keypoints in Image 1: 500, Keypoints in Image 2: 500, Time Taken: 0.0213 seconds

Feature Matching Performance:
Brute-Force Matcher (SIFT) - Matches Found: 95, Time Taken: 0.0135 seconds
FLANN Matcher (SIFT) - Good Matches Found: 3, Time Taken: 0.0154 seconds

--- Performance Summary ---
SIFT - Keypoints: 396, Time: 0.4273s
SURF - Keypoints: 1057, Time: 0.6735s
ORB  - Keypoints: 500, Time: 0.0213s
Brute-Force Matcher (SIFT): 95 matches, Time: 0.0135s
FLANN Matcher (SIFT): 3 good matches, Time: 0.0154s
```

This performance summary provides a performance comparison of three different feature detection algorithms—SIFT (Scale-Invariant Feature Transform), SURF (Speeded-Up Robust Features), and ORB (Oriented FAST and Rotated BRIEF)—in terms of the number of keypoints detected and the time taken for detection.

- SIFT detected 396 keypoints in Image 1 and 234 keypoints in Image 2, taking approximately 0.4273 seconds. This method is known for its strength against scale and rotation changes but is computationally intensive, which is reflected in its relatively longer processing time.
-
- SURF detected significantly more keypoints, with 1,057 in Image 1 and 469 in Image 2, but took longer at 0.6735 seconds. SURF is generally faster than SIFT due

to its use of integral images and approximations, yet it still exhibits higher computational demands compared to ORB.

- ORB, on the other hand, detected 500 keypoints in both images and performed the detection in just 0.0213 seconds. ORB is a more lightweight alternative, designed to be efficient and fast while still providing good performance, particularly in real-time applications.
- In terms of feature matching performance, the Brute-Force Matcher using SIFT found 95 matches in a very short time of 0.0135 seconds, indicating its effectiveness in matching keypoints despite the computational cost of SIFT. Conversely, the FLANN Matcher with SIFT yielded only 3 good matches in 0.0154 seconds. This result suggests that while FLANN is generally faster for larger datasets, it may not always yield a sufficient number of good matches compared to the Brute-Force method, particularly when the number of keypoints is lower or the matches are less distinct.
- Overall, the performance summary indicates that while SIFT and SURF provide strong keypoint detection, they are slower than ORB, which offers a good balance of speed and performance. The choice of matcher also affects the number of matches found, with Brute-Force being more effective in this case than FLANN for SIFT descriptors.

- *End of Documentation -*