Republic of the Philippines
Laguna State Polytechnic University
Province of Laguna

EXERCISE NO. 2			
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:	

Instructions:
Complete the following tasks using OpenCV and relevant libraries in Python (e.g., OpenCV, scikit-image). Each task should be implemented in a separate code block. Submit your Python script or notebook with all cells executed, outputs displayed, and brief explanations describing your approach, observations, and results.

SIFT Feature Extraction
SIFT (Scale-Invariant Feature Transform) detects important points, called keypoints, in an image. These keypoints 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 keypoint, which helps in matching these points across images.
The code first loads the image, converts it to grayscale (because many feature detectors work better on grayscale images), and then uses the SIFT algorithm to detect keypoints. The keypoints are visualized on the image.

Key Points:
- Keypoints are important image features.
- Descriptors are used to describe and match these keypoints.

Task 1: SIFT Feature Extraction
1. Load an image of your choice.
2. Use the SIFT (Scale-Invariant Feature Transform) algorithm to detect and compute keypoints and descriptors in the image.
3. Visualize the keypoints on the image and display the result.

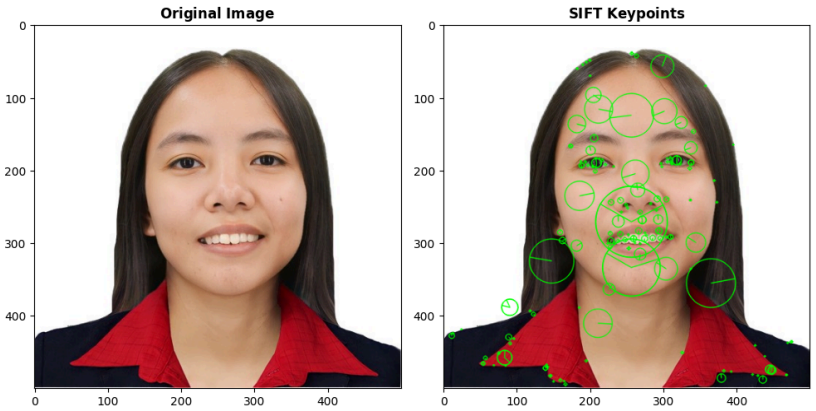
SIFT Feature Extraction

```
#Install the OpenCV headless version for environments where display (GUI) functionality is not required.
!pip install opencv-python-headless
```

```python
#Import Libraries.
import cv2
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

#Load the Original Image.
image_path = '/content/drive/MyDrive/2X2.jpg'
image = Image.open(image_path)

#Convert the PIL image to a NumPy array (already in RGB format).
image_np = np.array(image)

#Convert to grayscale for keypoint detection.
gray_image = cv2.cvtColor(image_np, cv2.COLOR_RGB2GRAY)

#Initialize the SIFT (Scale-Invariant Feature Transform).
sift = cv2.SIFT_create()

#Detect keypoints and compute descriptors.
keypoints, descriptors = sift.detectAndCompute(gray_image, None)

#Draw the detected keypoints on the original image (in RGB format).
image_with_keypoints = cv2.drawKeypoints(image_np, keypoints, None, (0, 255, 0), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

#Display Original Image.
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
plt.imshow(image_np)
plt.title(r'$\bf{Original\ Image}$')

#Display SIFT Keypoints.
plt.subplot(1, 2, 2)
plt.imshow(image_with_keypoints)
plt.title(r'$\bf{SIFT\ Keypoints}$')
plt.tight_layout()
plt.show()
```

This code performs SIFT (Scale-Invariant Feature Transform) keypoint detection on an image and displays both the original image and the image with detected keypoints side by side. First, it imports the necessary libraries: OpenCV for image processing, Matplotlib for visualization, NumPy for array handling, and PIL for image loading. The image is loaded from a specified path and converted to a NumPy array for OpenCV compatibility. The code then converts the image to grayscale (required for SIFT), initializes the SIFT detector, and detects keypoints—distinctive features in the image—while also generating descriptors that describe these features. The detected keypoints are overlaid on the original image in green, highlighting areas where SIFT identified unique structures. Finally, Matplotlib displays the original image alongside the keypoint-enhanced version, allowing for a visual comparison between the two.

RESULT:



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.

In the code, SURF is used to detect keypoints in a grayscale image, and the keypoints are visualized similarly to SIFT. The performance of SURF is usually faster than SIFT, but it might miss certain keypoints that SIFT would detect.

Key Points:

- SURF is faster than SIFT.
- It can be a good choice for real-time applications.

Task 2: SURF Feature Extraction

1. Load a different image (or the same one).
2. Apply the SURF (Speeded-Up Robust Features) algorithm to detect and compute keypoints and descriptors.
3. Visualize and display the keypoints.

SURF Feature Extraction

```
#Unistall the current Package.
!pip uninstall -y opencv-python opencv-python-headless opencv-contrib-python
```

The command `!pip uninstall -y opencv-python opencv-python-headless opencv-contrib-python` removes all installed versions of OpenCV in the environment. The `-y` flag automatically confirms the uninstallation. This command clears out `opencv-python` (standard with GUI), `opencv-python-headless` (optimized for non-GUI environments), and `opencv-contrib-python` (which includes additional modules), allowing for a fresh setup if reinstalling specific OpenCV versions.

```
#Install necessary development tools and dependencies required to build OpenCV from source.
!apt-get install -y cmake
!apt-get install -y libopencv-dev build-essential cmake git pkg-config libgtk-3-dev \
   libavcodec-dev libavformat-dev libswscale-dev libtbb2 libtbb-dev libjpeg-dev \
   libpng-dev libtiff-dev libdc1394-22-dev libv4l-dev v4l-utils \
   libxvidcore-dev libx264-dev libxine2-dev gstreamer1.0-tools \
   libgstreamer-plugins-base1.0-dev libgstreamer-plugins-good1.0-dev \
   libgtk2.0-dev libtiff5-dev libopenexr-dev libatlas-base-dev \
   python3-dev python3-numpy libtbb-dev libeigen3-dev \
   libfaac-dev libmp3lame-dev libtheora-dev libvorbis-dev \
   libxvidcore-dev libx264-dev yasm libopencore-amrnb-dev \
   libopencore-amrwb-dev libv4l-dev libxine2-dev libtesseract-dev \
   liblapacke-dev libopenblas-dev checkinstall
```

This command installs the necessary development tools and dependencies required to build OpenCV from source. It first installs `cmake`, a build automation tool, and then a wide range of libraries essential for OpenCV's functionality, including those for image and video processing (like `libjpeg-dev`, `libpng-dev`, `libtiff-dev`), video streaming (like `libavcodec-dev`, `libavformat-dev`), graphical interfaces (`libgtk-3-dev`), and various mathematical and computational libraries (`libtbb-dev`, `libeigen3-dev`). Additionally, it includes Python support (`python3-dev`, `python3-numpy`) and other multimedia codecs (`libxvidcore-dev`, `libx264-dev`, `libmp3lame-dev`) to enable full multimedia processing capabilities in OpenCV.

```
#Clone the OpenCV repository from GitHub.
!git clone https://github.com/opencv/opencv.git
!git clone https://github.com/opencv/opencv_contrib.git
```

This command clones the OpenCV repositories from GitHub to the local environment. The first line, `!git clone https://github.com/opencv/opencv.git`, downloads the main OpenCV repository, which includes core functionalities and standard modules. The second line, `!git clone https://github.com/opencv/opencv_contrib.git`, downloads the "contrib" repository, which contains additional modules and experimental features not included in the main repository. Cloning both allows you to build OpenCV with extended functionalities if desired.

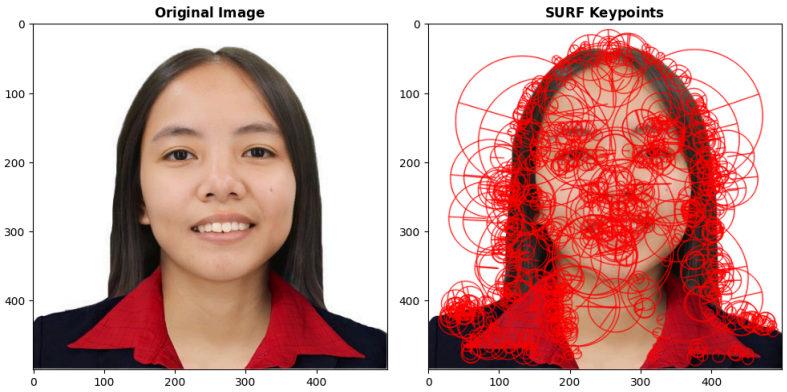
```
#Change directory to the cloned OpenCV directory.
%cd opencv
#Create a build directory for building the OpenCV source.
!mkdir build
#Move into the newly created build directory.
%cd build

#Run the CMake configuration for building OpenCV with specific options:
!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=ON ..

#Compile OpenCV using 8 threads (parallel compilation) for faster build times.
!make -j8
#Install the compiled OpenCV library into the system.
!make install
```

This code sets up and compiles OpenCV from source with custom configurations. First, it changes to the OpenCV directory and creates a `build` directory for organizing the build files. Inside the `build` directory, it uses CMake to configure the build with specific options, such as enabling non-free modules and including extra modules from `opencv_contrib`. Then, it compiles OpenCV using 8 threads to speed up the process (`-j8`), and finally installs the compiled library to the system. This allows for a customized OpenCV installation with additional modules and optimized settings.

```
#Import Libraries.
import cv2
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

#Load the Original Image.
image_path = '/content/drive/MyDrive/2X2.jpg'
image = Image.open(image_path)

#Convert the PIL image to a NumPy array (already in RGB format).
image_np = np.array(image)

#Convert to grayscale for keypoint detection.
gray_image = cv2.cvtColor(image_np, cv2.COLOR_RGB2GRAY)

#Initialize the SURF (Speeded-Up Robust Features).
surf = cv2.xfeatures2d.SURF_create()

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

#Draw the detected keypoints on the original image (in RGB format).
image_with_keypoints = cv2.drawKeypoints(image_np, keypoints, None, (255, 0, 0), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

#Display Original Image.
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
plt.imshow(image_np)
plt.title(r'$\bf{Original\ Image}$')

#Display SURF Keypoints.
plt.subplot(1, 2, 2)
plt.imshow(image_with_keypoints)
plt.title(r'$\bf{SURF\ Keypoints}$')
plt.tight_layout()
plt.show()
```

This code demonstrates how to use the SURF (Speeded-Up Robust Features) algorithm to detect keypoints in an image and then display these keypoints alongside the original image. First, it imports the necessary libraries for image processing, plotting, and array manipulation. An image is loaded from a specified path and converted to a NumPy array for OpenCV compatibility. The image is then converted to grayscale, as the SURF algorithm requires a single-channel image to detect keypoints efficiently. Next, the SURF detector is initialized to identify keypoints and compute descriptors, which describe the detected points. These keypoints are then drawn as blue circles on the original RGB image, with each circle showing both the position and scale of the detected features. Finally, the code uses Matplotlib to display the original image on the left and the image with SURF keypoints on the right, providing a visual comparison between the original and feature-detected versions. Note that this code requires the `opencv-contrib-python` package since SURF is part of OpenCV's extra modules (`xfeatures2d`).

RESULT:



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.

The code uses ORB to detect keypoints and display them on the image. Unlike SIFT and SURF, ORB is more focused on speed and efficiency, which makes it suitable for applications that need to process images quickly.

Key Points:

- ORB is a fast alternative to SIFT and SURF.
- It's suitable for real-time and resource-constrained environments.

Task 3: ORB Feature Extraction (20 points)

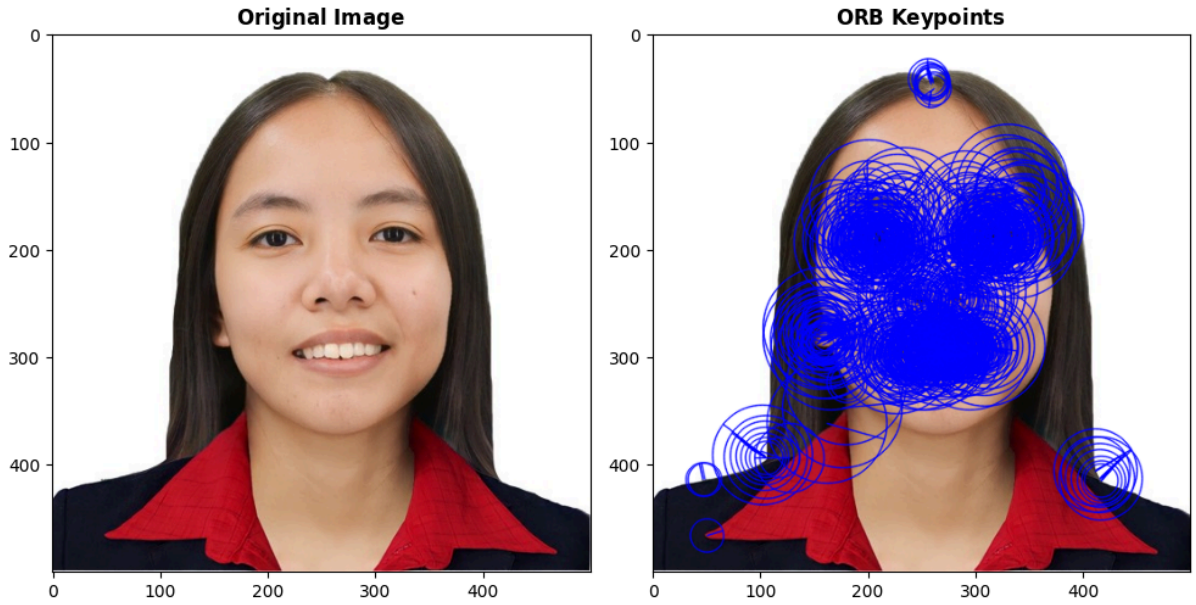
1. Apply the ORB (Oriented FAST and Rotated BRIEF) algorithm to detect keypoints and compute descriptors on another image.
2. Visualize and display the keypoints.

ORB Feature Extraction

```python
#Import Libraries.
import cv2
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

#Load the Original Image.
image_path = '/content/drive/MyDrive/2X2.jpg'
image = Image.open(image_path)

#Convert the PIL image to a NumPy array (already in RGB format).
image_np = np.array(image)

#Convert to grayscale for keypoint detection.
gray_image = cv2.cvtColor(image_np, cv2.COLOR_RGB2GRAY)

#Initialize ORB (Oriented FAST and Rotated BRIEF).
orb = cv2.ORB_create()

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

#Draw the detected keypoints on the original image (in RGB format).
image_with_keypoints = cv2.drawKeypoints(image_np, keypoints, None, (0, 0, 255), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

#Display Original Image.
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
plt.imshow(image_np)
plt.title(r'$\bf{Original\ Image}$')

#Display ORB Keypoints.
plt.subplot(1, 2, 2)
plt.imshow(image_with_keypoints)
plt.title(r'$\bf{ORB\ Keypoints}$')
plt.tight_layout()
plt.show()
```

This code performs keypoint detection on an image using the ORB (Oriented FAST and Rotated BRIEF) algorithm, which is efficient for detecting and describing keypoints. First, it loads an image and converts it into a NumPy array for processing. The image is then converted to grayscale, as ORB requires a single-channel image for analysis. After initializing ORB, the code detects keypoints (distinctive points) in the grayscale image and computes descriptors for each keypoint. These detected keypoints are then drawn on the original RGB image in red. Finally, using Matplotlib, the original image and the image with ORB-detected keypoints are displayed side by side, allowing for a visual comparison of the keypoint detection results.

RESULT:



Feature Matching using SIFT

In this exercise, 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.

Key Points:

- Feature matching helps compare and find similarities between two images.
- The Brute-Force Matcher finds the closest matching descriptors.

Task 4: Feature Matching

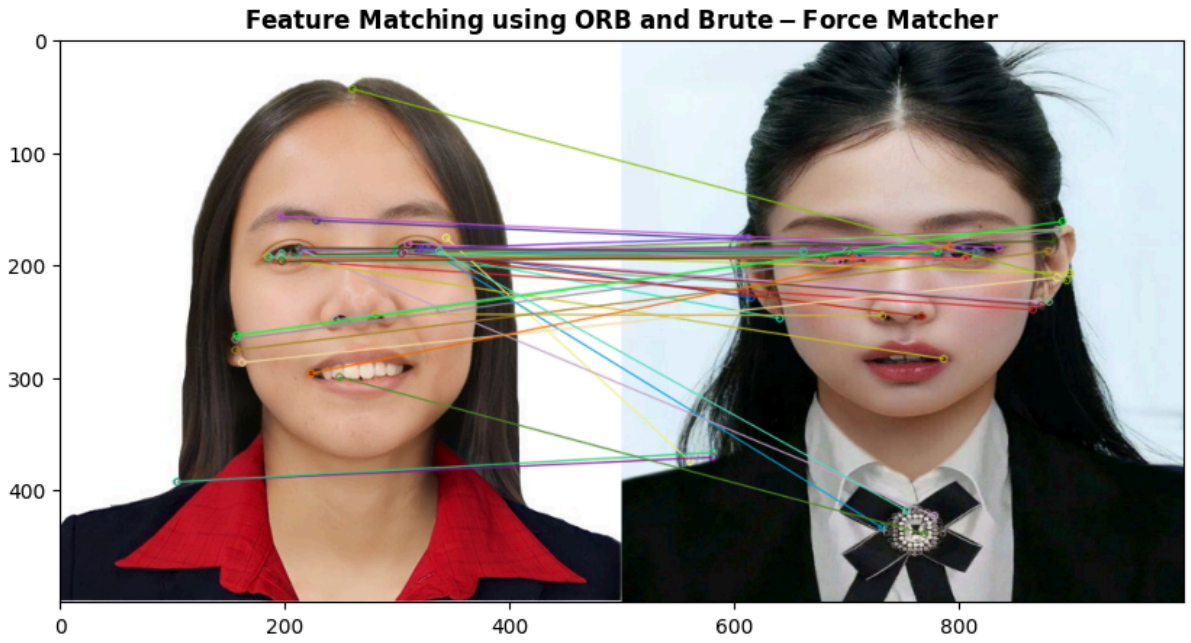
1. Using the keypoints and descriptors obtained from the previous tasks (e.g., SIFT, SURF, or ORB), match the features between two different images using Brute-Force Matching or FLANN (Fast Library for Approximate Nearest Neighbors).
2. Display the matched keypoints on both images.

Feature Matching using SIFT

```python
#Import Libraries.
import cv2
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

#Load the two images.
image1_path = '/content/drive/MyDrive/2X2.jpg'
image2_path = '/content/drive/MyDrive/jennie.jpg'
image1 = Image.open(image1_path)
image2 = Image.open(image2_path)

#Convert the images to NumPy arrays.
image1_np = np.array(image1)
image2_np = np.array(image2)

#Resize the images to the same size.
height, width = image1_np.shape[:2]
image2_np = cv2.resize(image2_np, (width, height))

#Convert to grayscale for keypoint detection.
gray_image1 = cv2.cvtColor(image1_np, cv2.COLOR_RGB2GRAY)
gray_image2 = cv2.cvtColor(image2_np, cv2.COLOR_RGB2GRAY)

#Initialize ORB (Oriented FAST and Rotated BRIEF).
orb = cv2.ORB_create(nfeatures=500)

#Detect keypoints and compute descriptors for both images.
keypoints1, descriptors1 = orb.detectAndCompute(gray_image1, None)
keypoints2, descriptors2 = orb.detectAndCompute(gray_image2, None)

#Initialize Brute-Force Matcher.
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

#Match descriptors between the two images.
matches = bf.match(descriptors1, descriptors2)

#Sort matches based on distance (the lower the distance, the better the match).
matches = sorted(matches, key=lambda x: x.distance)

#Draw the top matches (you can adjust the number of matches to display).
matched_image = cv2.drawMatches(image1_np, keypoints1, image2_np, keypoints2, matches[:50], None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

#Display the feature matching keypoints.
plt.figure(figsize=(10, 6))
plt.imshow(matched_image)
plt.title(r'$\bf{Feature\ Matching\ using\ ORB\ and\ Brute-Force\ Matcher}$')
plt.show()
```

This code performs feature matching between two images using the ORB (Oriented FAST and Rotated BRIEF) algorithm and a Brute-Force Matcher. First, the images are loaded, converted to NumPy arrays, and resized to ensure they are the same dimensions. They are then converted to grayscale for keypoint detection. ORB, a fast and efficient feature detection algorithm, is initialized to identify keypoints and compute descriptors for each image. The Brute-Force Matcher is set up to compare descriptors and find matches, where each match represents a point in one image that corresponds to a similar point in the other. The matches are sorted based on distance, with the closest matches considered the best. Finally, the top 50 matches are drawn between the two images, and the result is displayed using Matplotlib to visualize the keypoints that best align between the two images.

RESULT:



Real-World Applications (Image Stitching using Homography)
In this task, you use matched keypoints from two images to align or "stitch" them together. 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.

Key Points:

- Homography is used to align images.
- This is useful in applications like panoramic image creation or object recognition.

Task 5: Applications of Feature Matching

1. Apply feature matching to two images of the same scene taken from different angles or perspectives.
2. Use the matched features to align the images (e.g., using homography to warp one image onto another).

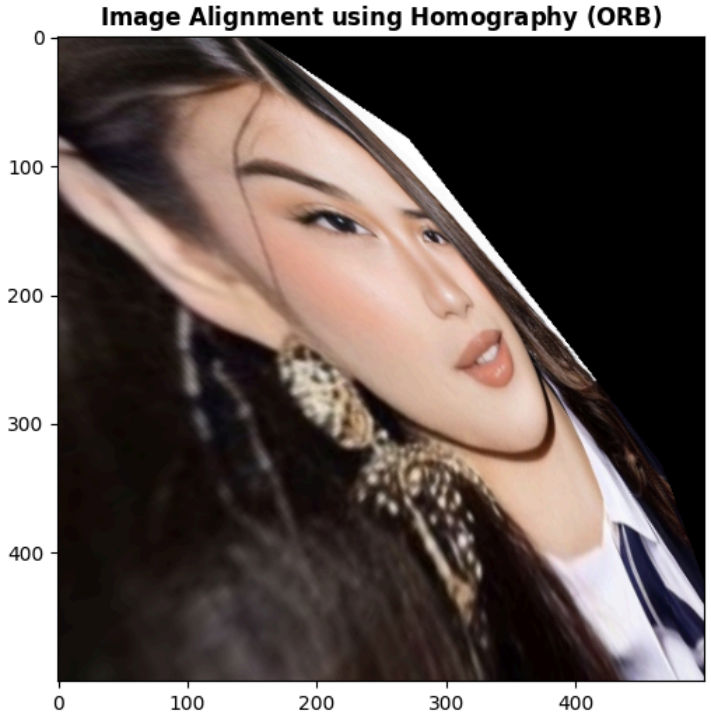
Real-World Applications (Image Stitching using Homography)

```python
#Import Libraries.
import cv2
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

#Load the two images.
image1_path = '/content/drive/MyDrive/2X2.jpg'
image2_path = '/content/drive/MyDrive/aiah.jpg'
image1 = Image.open(image1_path)
image2 = Image.open(image2_path)

#Convert the images to NumPy arrays.
image1_np = np.array(image1)
image2_np = np.array(image2)

#Resize the second image to match the first image's size.
height, width = image1_np.shape[:2]
image2_np = cv2.resize(image2_np, (width, height))

#Convert to grayscale for keypoint detection.
gray_image1 = cv2.cvtColor(image1_np, cv2.COLOR_RGB2GRAY)
gray_image2 = cv2.cvtColor(image2_np, cv2.COLOR_RGB2GRAY)

#Initialize ORB (Oriented FAST and Rotated BRIEF).
orb = cv2.ORB_create()

#Detect keypoints and compute descriptors using ORB.
keypoints1, descriptors1 = orb.detectAndCompute(gray_image1, None)
keypoints2, descriptors2 = orb.detectAndCompute(gray_image2, None)

#Match features using BFMatcher.
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(descriptors1, descriptors2)

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

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

#Find the homography matrix.
M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC)

#Warp the second image to align with the first image.
h, w = image1_np.shape[:2]
result = cv2.warpPerspective(image2_np, M, (w, h))

#Display the applications of feature matching.
plt.figure(figsize=(10, 6))
plt.imshow(result)
plt.title(r'$\bf{Image\ Alignment\ using\ Homography\ (ORB)}$')
plt.show()
```

This code aligns two images using feature matching and homography. First, it loads two images and converts them into NumPy arrays. The second image is resized to match the

dimensions of the first. Both images are converted to grayscale for feature detection. The ORB (Oriented FAST and Rotated BRIEF) algorithm detects keypoints and computes descriptors for both images. These descriptors are then matched using BFMatcher, a brute-force matcher that pairs features from both images based on similarity. The matches are sorted by distance to prioritize stronger matches. The coordinates of the matched points are used to compute a homography matrix, which describes the transformation needed to align the second image with the first. Finally, the `warpPerspective` function uses this homography matrix to adjust the second image's perspective, and the result is displayed, showing the aligned version of the second image overlaid onto the first.

RESULT:



Combining SIFT and ORB

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 robust feature detection and matching.
In the code, we extract keypoints from two images using both SIFT and ORB, and then you can use a matcher to compare and match the features detected by both methods.

Key Points:

- Combining methods can improve performance in some applications.

- SIFT is accurate, while ORB is fast, making them complementary in certain tasks.

Task 6: Combining Feature Extraction Methods (10 points)

1. Combine multiple feature extraction methods (e.g., SIFT + ORB) to extract features and match them between two images.
2. Display the combined result.

Combining SIFT and ORB

```
#Import Libraries.
import cv2
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

#Load the two images in color.
image1_path = '/content/drive/MyDrive/jennie.jpg'
image2_path = '/content/drive/MyDrive/aiah.jpg'
image1 = cv2.imread(image1_path)
image2 = cv2.imread(image2_path)

#Resize images.
height, width = 900, 800
image1 = cv2.resize(image1, (width, height))
image2 = cv2.resize(image2, (width, height))

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

#SIFT detector.
sift = cv2.SIFT_create()
keypoints1_sift, descriptors1_sift = sift.detectAndCompute(gray1, None)
keypoints2_sift, descriptors2_sift = sift.detectAndCompute(gray2, None)

#ORB detector.
orb = cv2.ORB_create()
keypoints1_orb, descriptors1_orb = orb.detectAndCompute(gray1, None)
keypoints2_orb, descriptors2_orb = orb.detectAndCompute(gray2, None)

#Brute Force Matcher for SIFT (since SIFT uses 128-dim descriptors).
bf_sift = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
matches_sift = bf_sift.match(descriptors1_sift, descriptors2_sift)

#Brute Force Matcher for ORB (since ORB uses 32-dim descriptors).
bf_orb = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches_orb = bf_orb.match(descriptors1_orb, descriptors2_orb)

#Sort matches by distance (optional, for better visualization).
matches_sift = sorted(matches_sift, key=lambda x: x.distance)
matches_orb = sorted(matches_orb, key=lambda x: x.distance)
```

```
#Draw top 50 matches for SIFT and ORB (using original BGR images).
img_matches_sift = cv2.drawMatches(image1, keypoints1_sift, image2, keypoints2_sift, matches_sift[:50], None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
img_matches_orb = cv2.drawMatches(image1, keypoints1_orb, image2, keypoints2_orb, matches_orb[:50], None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

#Combine SIFT and ORB matches.
combined_matches = matches_sift[:25] + matches_orb[:25]

#Combine keypoints for display.
combined_keypoints1 = keypoints1_sift + keypoints1_orb
combined_keypoints2 = keypoints2_sift + keypoints2_orb

#Draw matches for combined SIFT + ORB (using original BGR images).
combined_img_matches = cv2.drawMatches(
    image1, combined_keypoints1, image2, combined_keypoints2, combined_matches, None,
    flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
)
plt.figure(figsize=(12, 10))

#Display SIFT Feature Matching.
plt.subplot(2, 2, 1)
plt.imshow(cv2.cvtColor(img_matches_sift, cv2.COLOR_BGR2RGB))
plt.title('SIFT Feature Matching')
plt.axis('off')

#Display ORB Feature Matching.
plt.subplot(2, 2, 2)
plt.imshow(cv2.cvtColor(img_matches_orb, cv2.COLOR_BGR2RGB))
plt.title('ORB Feature Matching')
plt.axis('off')

# Display combined SIFT + ORB matches.
plt.subplot(2, 1, 2)
plt.imshow(cv2.cvtColor(combined_img_matches, cv2.COLOR_BGR2RGB))
plt.title('Combined SIFT + ORB Feature Matching')
plt.axis('off')
plt.tight_layout()
plt.show()
```

This code demonstrates feature detection and matching between two images using both SIFT (Scale-Invariant Feature Transform) and ORB (Oriented FAST and Rotated BRIEF) algorithms. It then displays the matches detected by each algorithm individually as well as a combined result.

First, the necessary libraries for image processing, plotting, and array manipulation are imported. Two images are loaded in color from specified paths and resized to a fixed dimension of 900 by 800 pixels for consistency. The resized images are then converted to grayscale, as feature detection algorithms like SIFT and ORB typically work better with single-channel images. The SIFT and ORB detectors are created to identify keypoints—distinctive, easily recognizable points—in each grayscale image. For SIFT, the code detects keypoints and computes 128-dimensional descriptors for these points, which capture the surrounding region's characteristics. Similarly, ORB detects keypoints and computes 32-dimensional descriptors for each, making it faster and more efficient than SIFT, though potentially less precise.

Next, two brute-force matchers are created to find matching descriptors between the two images. For SIFT, a brute-force matcher with the L2 norm is used to match its 128-dimensional descriptors, while ORB uses the Hamming norm for its binary descriptors. The matches are then sorted by distance, bringing the closest matches (those with the smallest distance between descriptors) to the top. The code draws the top 50 matches for each detector on the original color images, creating visualizations for SIFT and ORB matches. It also combines the top 25 matches from each detector and combines their keypoints to create a blended display showing both SIFT and ORB matches together.

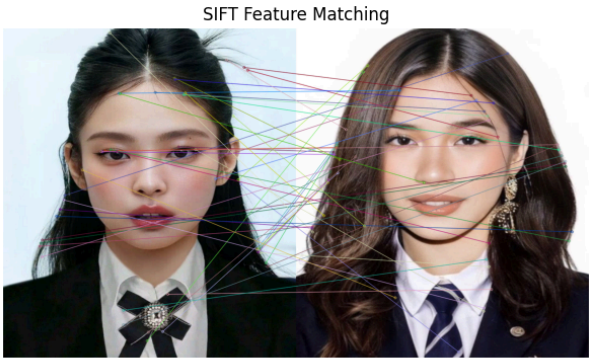
Finally, Matplotlib is used to display the results in a 2x2 grid:

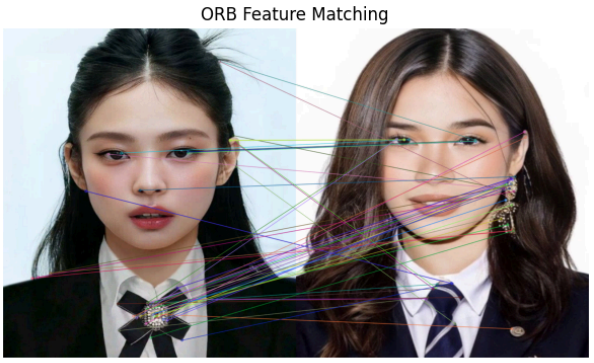
- The first plot shows the matches from SIFT.

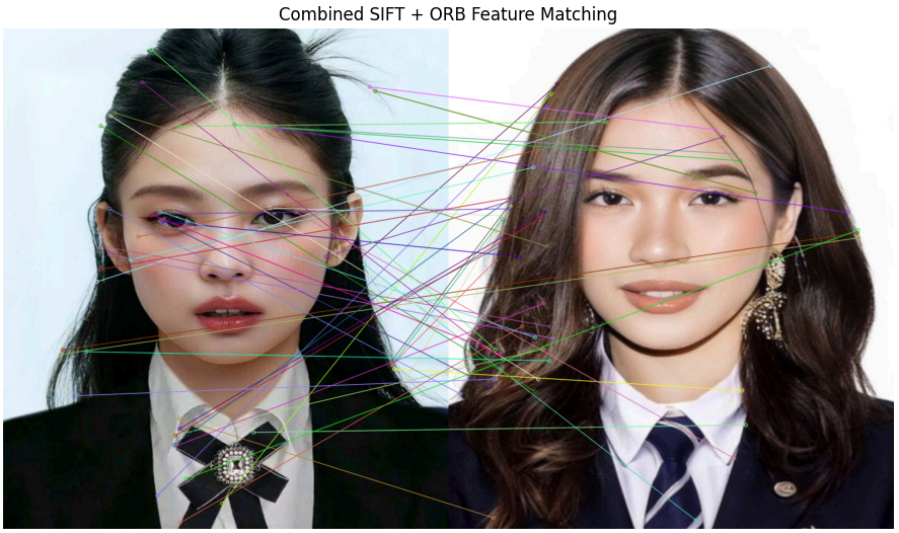
- The second plot shows the matches from ORB.
- The third plot, spanning the bottom row, displays the combined matches from both algorithms.

This provides a side-by-side comparison of each algorithm's effectiveness in detecting and matching features between the two images.

RESULT:







Overall Understanding:

- SIFT is accurate for detecting and matching keypoints even in transformed images (scaled, rotated).
- SURF is faster than SIFT but still effective for keypoint detection.
- ORB is highly efficient and suited for real-time applications.
- Feature Matching is essential in comparing different images to find common objects or align them.
- Homography is used in aligning images, such as stitching images together to form a panorama.

These methods are foundational in computer vision tasks, such as object detection, image stitching, and image recognition.

CONCLUSION:

In summary, SIFT, SURF, and ORB are powerful techniques for detecting and matching keypoints across images, each with unique strengths tailored to different applications. SIFT provides high accuracy in detecting keypoints, even under transformations, making it ideal for complex image comparisons. SURF offers a balanced approach, providing speed without sacrificing too much accuracy, while ORB excels in efficiency, making it well-suited for real-time applications. Feature matching, a critical aspect of these methods, enables identifying common elements across images, supporting tasks like object recognition and alignment. Homography plays a vital role in aligning matched images, facilitating applications like image stitching for panoramic views. Together, these methods form the backbone of many computer vision applications, from image recognition to panorama creation, and are instrumental in advancing the field of visual analysis.