Republic of the Philippines
Laguna State Polytechnic University
Province of Laguna

Machine Problem No. 3			
Topic:	**Module 2.0: Feature Extraction and Object Detection**	**Week No.**	6-7
Course Code:	**CSST106**	**Term:**	1st Semester
Course Title:	**Perception and Computer Vision**	**Academic Year:**	2024-2025
Student Name	**Lesly-Ann B. Victoria**	**Section**	BSCS-4B
Due date		**Points**	

Machine Problem No. 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:
You are tasked with loading two images and performing the following steps:

1. Extract keypoints and descriptors from both images using **SIFT**, **SURF**, and **ORB**.
2. Perform feature matching between the two images using both **Brute-Force Matcher** and **FLANN Matcher**.
3. Use the matched keypoints to calculate a **Homography matrix** and align the two images.
4. Compare the performance of SIFT, SURF, and ORB in terms of feature matching accuracy and speed.

You will submit your code, processed images, and a short report comparing the results.

Task Breakdown:

PACKAGES:

```
#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.

Step 1: Load Images

- Load two images of your choice that depict the same scene or object but from different angles.

CODE:

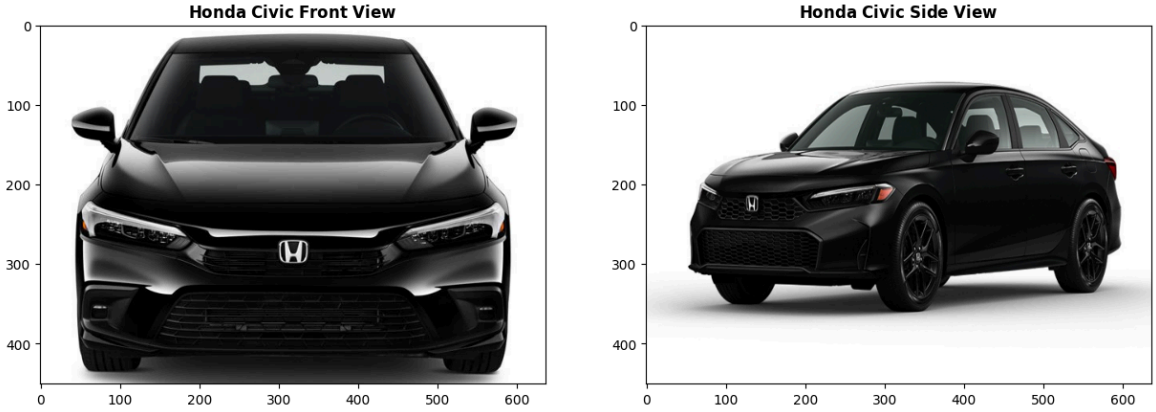
This code begins by importing several libraries essential for image processing and visualization: OpenCV, Matplotlib, NumPy, and the Python Imaging Library (PIL). These libraries allow the user to load, manipulate, and display images.

First, two images of a car are loaded from specified file paths. `image1_path` and `image2_path` hold the paths to a front view and a side view of a Honda Civic, respectively. The PIL library opens these images, which are then converted to NumPy arrays, enabling easy manipulation and access to pixel-level data. The code resizes the second image to match the dimensions of the first one to ensure consistency in display. It does this by extracting the height and width of `image1` and resizing `image2` to those dimensions using OpenCV's `resize` function. This step ensures that both images are of equal size, making them visually comparable.

Finally, the code uses Matplotlib to display both images side by side. Each image is placed in a subplot within a larger figure. The titles above each subplot—"Honda Civic Front View" and "Honda Civic Side View"—use LaTeX formatting for a bold appearance, enhancing the presentation of the images. The `plt.show()` command at the end renders the figure, allowing viewers to observe the two perspectives of the Honda Civic side by side.

```
#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/car_frontview.jpg'
image2_path = '/content/drive/MyDrive/car_sideview.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))

#Display the two original color images.
plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
plt.title(r'$\bf{Honda\ Civic\ Front\ View}$')
plt.imshow(image1_np)
plt.subplot(1, 2, 2)
plt.title(r'$\bf{Honda\ Civic\ Side\ View}$')
plt.imshow(image2_np)
plt.show()
```

RESULT:



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

- Apply the **SIFT** algorithm to detect keypoints and compute descriptors for both images.
- Apply the **SURF** algorithm to do the same.
- Finally, apply **ORB** to extract keypoints and descriptors.

CODE:

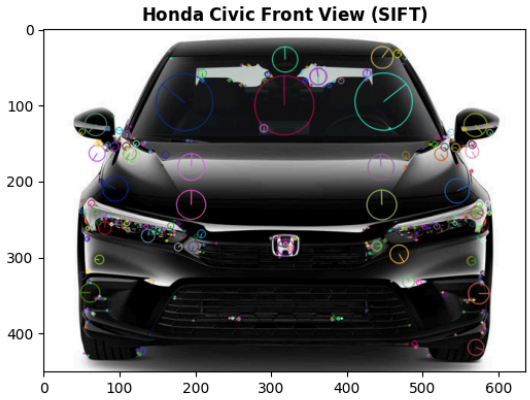
The code provides extracts and visualizes keypoints from two images of a Honda Civic car using three different feature detection algorithms: SIFT, SURF, and ORB. The `extract_features` function begins by applying each algorithm to both images. For SIFT (Scale-Invariant Feature Transform), it detects keypoints and descriptors that capture distinctive features within each image, storing them in `kp1_sift` and `kp2_sift` for the two images, along with their descriptors `des1_sift` and `des2_sift`. SURF (Speeded-Up Robust Features), a similar algorithm known for its speed and accuracy, detects keypoints and descriptors as well, which are saved in `kp1_surf`, `kp2_surf`, `des1_surf`, and `des2_surf`. Finally, the ORB (Oriented FAST and Rotated BRIEF) algorithm, known for being computationally efficient, captures its keypoints and descriptors in a similar manner, assigning them to `kp1_orb`, `kp2_orb`, `des1_orb`, and `des2_orb`.

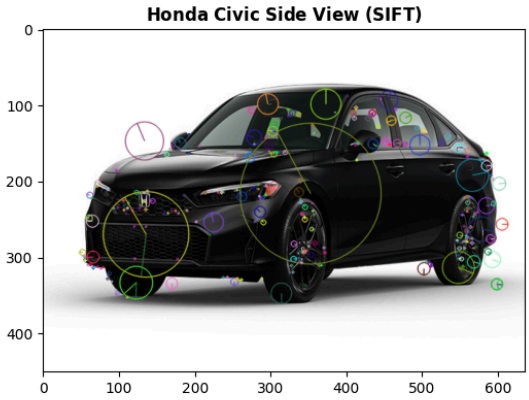
The function `draw_keypoints` is then used to visualize these keypoints on the images. Each algorithm's keypoints are drawn on both the front and side views of the Honda Civic image. These images are displayed in a grid format using Matplotlib. Each row corresponds to a different algorithm, with SIFT in the first row, SURF in the second, and ORB in the third. The front and side views of the Honda Civic are displayed side by side in each row, enabling a comparison of the keypoints detected by each algorithm. The titles in the plot provide context, specifying the algorithm used and the view angle of the image. This visualization helps in comparing how each algorithm performs on similar images, illustrating the distinct patterns of keypoint distribution and density achieved by each feature detector.

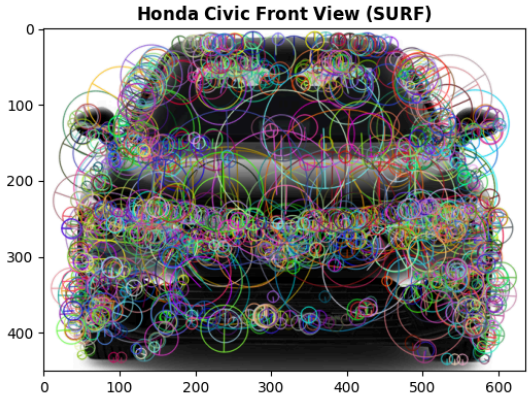
```python
def extract_features(image1, image2):
    #SIFT.
    sift = cv2.SIFT_create()
    kp1_sift, des1_sift = sift.detectAndCompute(image1_np, None)
    kp2_sift, des2_sift = sift.detectAndCompute(image2_np, None)

    #SURF.
    surf = cv2.xfeatures2d.SURF_create()
    kp1_surf, des1_surf = surf.detectAndCompute(image1_np, None)
    kp2_surf, des2_surf = surf.detectAndCompute(image2_np, None)

    #ORB.
    orb = cv2.ORB_create()
    kp1_orb, des1_orb = orb.detectAndCompute(image1_np, None)
    kp2_orb, des2_orb = orb.detectAndCompute(image2_np, None)

    return (kp1_sift, des1_sift, kp2_sift, des2_sift), (kp1_surf, des1_surf, kp2_surf, des2_surf), (kp1_orb, des1_orb, kp2_orb, des2_orb)

#Extract features.
sift_features, surf_features, orb_features = extract_features(image1_np, image2_np)

#Function to draw keypoints.
def draw_keypoints(image, keypoints):
    return cv2.drawKeypoints(image, keypoints, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

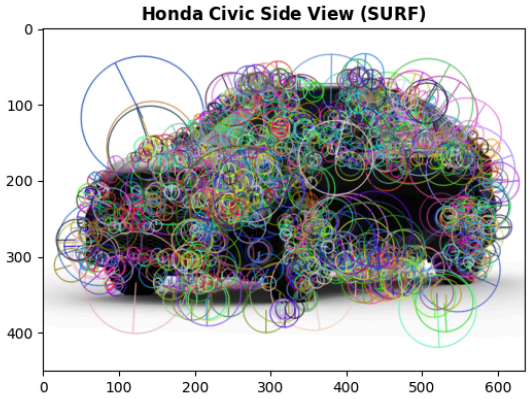
```python
#Draw keypoints for SIFT.
image1_sift_kp = draw_keypoints(image1_np, sift_features[0])
image2_sift_kp = draw_keypoints(image2_np, sift_features[2])

#Draw keypoints for SURF.
image1_surf_kp = draw_keypoints(image1_np, surf_features[0])
image2_surf_kp = draw_keypoints(image2_np, surf_features[2])

#Draw keypoints for ORB.
image1_orb_kp = draw_keypoints(image1_np, orb_features[0])
image2_orb_kp = draw_keypoints(image2_np, orb_features[2])

#Display the keypoints for each method.
plt.figure(figsize=(15, 12))
plt.subplot(3, 2, 1)
plt.title(r'$\bf{Honda\ Civic\ Front\ View\ (SIFT)}$')
plt.imshow(image1_sift_kp)
plt.subplot(3, 2, 2)
plt.title(r'$\bf{Honda\ Civic\ Side\ View\ (SIFT)}$')
plt.imshow(image2_sift_kp)
plt.subplot(3, 2, 3)
plt.title(r'$\bf{Honda\ Civic\ Front\ View\ (SURF)}$')
plt.imshow(image1_surf_kp)
plt.subplot(3, 2, 4)
plt.title(r'$\bf{Honda\ Civic\ Side\ View\ (SURF)}$')
plt.imshow(image2_surf_kp)
plt.subplot(3, 2, 5)
plt.title(r'$\bf{Honda\ Civic\ Front\ View\ (ORB)}$')
plt.imshow(image1_orb_kp)
plt.subplot(3, 2, 6)
plt.title(r'$\bf{Honda\ Civic\ Side\ View\ (ORB)}$')
plt.imshow(image2_orb_kp)
plt.tight_layout()
plt.show()
```

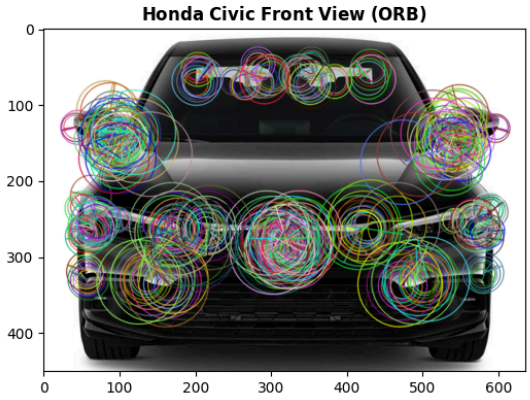
RESULT:

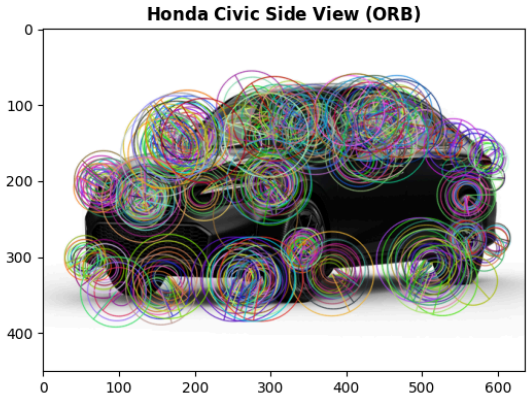












Step 3: Feature Matching with Brute-Force and FLANN

- Match the descriptors between the two images using **Brute-Force Matcher**.
- Repeat the process using the **FLANN Matcher**.
- For each matching method, display the matches with lines connecting corresponding keypoints between the two images.

CODE:

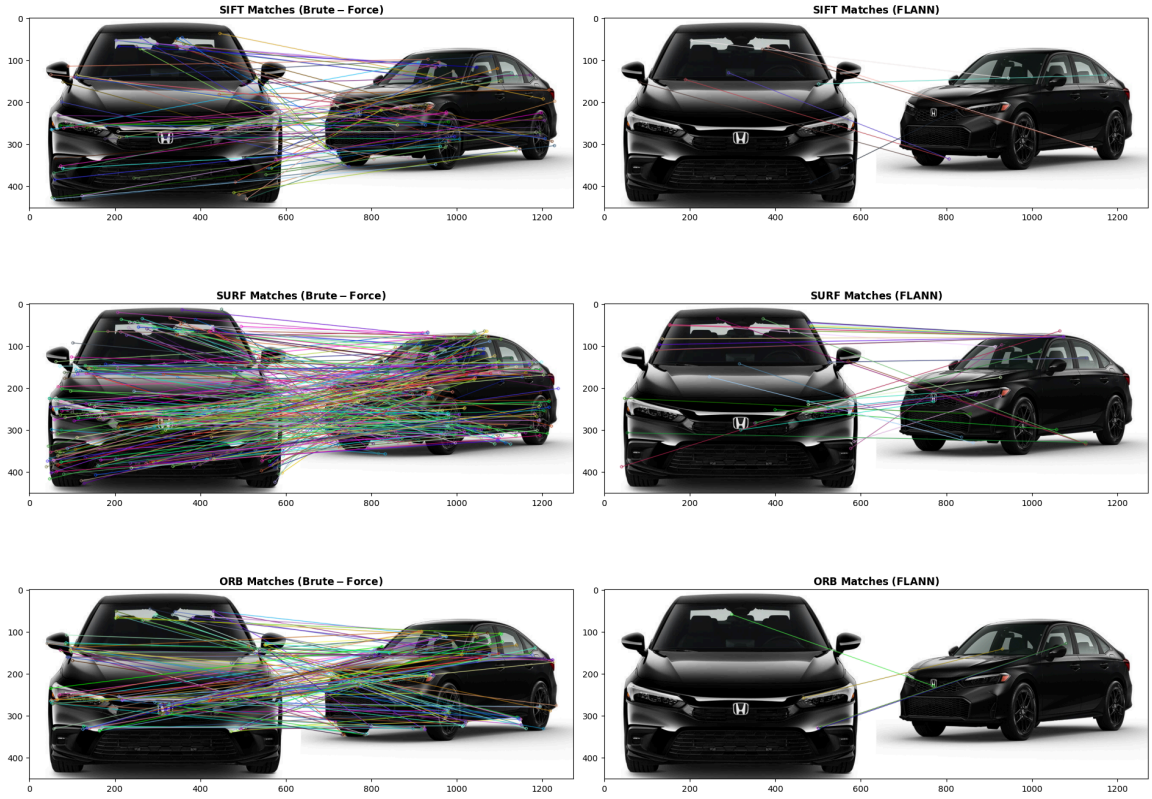
This code performs feature matching between two images using three popular feature extraction methods—SIFT, SURF, and ORB—and two matching techniques: Brute-Force Matcher and FLANN Matcher. The process begins by defining two functions for matching descriptors. The `match_descriptors_bf` function uses the Brute-Force Matcher to compare feature descriptors and returns matches sorted by distance. The `match_descriptors_flann` function employs the FLANN (Fast Library for Approximate Nearest Neighbors) Matcher, using parameters for faster matching. It also includes Lowe's ratio test to retain only the best matches, reducing false matches by setting a threshold on distance. Next, the code performs matching for each feature extraction method. For SIFT and SURF, it calls both the Brute-Force and FLANN matchers, while for ORB, the descriptors are first converted to `float32` to ensure compatibility with FLANN. The matching results for each feature and matcher combination are stored in variables like `sift_matches_bf` and `sift_matches_flann`.

A `draw_matches` function is then used to visualize the matches on the images, showing matched keypoints between the two images for each method and matcher type. Finally, the code organizes the results into a 3x2 grid using `matplotlib` and displays them in six subplots. Each subplot title indicates the matching type and method used: SIFT (Brute-Force and FLANN), SURF (Brute-Force and FLANN), and ORB (Brute-Force and FLANN), allowing a clear comparison of the matching performance across different methods and techniques.

```python
#Function for matching descriptors using Brute-Force Matcher.
def match_descriptors_bf(des1, des2):
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(des1, des2)
    matches = sorted(matches, key=lambda x: x.distance)
    return matches

#Function for matching descriptors using FLANN Matcher.
def match_descriptors_flann(des1, des2):
    index_params = dict(algorithm=1, trees=5)
    search_params = dict(checks=50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des1, des2, k=2)

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

    return good_matches

#Function to draw matches.
def draw_matches(image1, kp1, image2, kp2, matches):
    matched_image = cv2.drawMatches(image1, kp1, image2, kp2, matches, None,
                                    flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    return matched_image

#Perform matching for SIFT.
sift_matches_bf = match_descriptors_bf(sift_features[1], sift_features[3])
sift_matches_flann = match_descriptors_flann(sift_features[1], sift_features[3])

#Perform matching for SURF.
surf_matches_bf = match_descriptors_bf(surf_features[1], surf_features[3])
surf_matches_flann = match_descriptors_flann(surf_features[1], surf_features[3])

#Perform matching for ORB.
orb_matches_bf = match_descriptors_bf(orb_features[1], orb_features[3])
orb_matches_flann = match_descriptors_flann(orb_features[1].astype(np.float32), orb_features[3].astype(np.float32))
```

```python
#Draw matches for each method.
sift_bf_matches_img = draw_matches(image1_np, sift_features[0], image2_np, sift_features[2], sift_matches_bf)
sift_flann_matches_img = draw_matches(image1_np, sift_features[0], image2_np, sift_features[2], sift_matches_flann)

surf_bf_matches_img = draw_matches(image1_np, surf_features[0], image2_np, surf_features[2], surf_matches_bf)
surf_flann_matches_img = draw_matches(image1_np, surf_features[0], image2_np, surf_features[2], surf_matches_flann)

orb_bf_matches_img = draw_matches(image1_np, orb_features[0], image2_np, orb_features[2], orb_matches_bf)
orb_flann_matches_img = draw_matches(image1_np, orb_features[0], image2_np, orb_features[2], orb_matches_flann)

#Display the matching results.
plt.figure(figsize=(19, 15))
#SIFT using Brute-Force Matcher.
plt.subplot(3, 2, 1)
plt.title(r'$\bf{SIFT\ Matches\ (Brute-Force)}$')
plt.imshow(sift_bf_matches_img)
#SIFT using FLANN Matcher.
plt.subplot(3, 2, 2)
plt.title(r'$\bf{SIFT\ Matches\ (FLANN)}$')
plt.imshow(sift_flann_matches_img)
#SURF using Brute-Force Matcher.
plt.subplot(3, 2, 3)
plt.title(r'$\bf{SURF\ Matches\ (Brute-Force)}$')
plt.imshow(surf_bf_matches_img)
#SURF using FLANN Matcher.
plt.subplot(3, 2, 4)
plt.title(r'$\bf{SURF\ Matches\ (FLANN)}$')
plt.imshow(surf_flann_matches_img)
#ORB using Brute-Force Matcher.
plt.subplot(3, 2, 5)
plt.title(r'$\bf{ORB\ Matches\ (Brute-Force)}$')
plt.imshow(orb_bf_matches_img)
#ORB using FLANN Matcher.
plt.subplot(3, 2, 6)
plt.title(r'$\bf{ORB\ Matches\ (FLANN)}$')
plt.imshow(orb_flann_matches_img)
plt.tight_layout()
plt.show()
```

RESULT:



Step 4: Image Alignment Using Homography

- Use the matched keypoints from **SIFT** (or any other method) to compute a **homography matrix**.
- Use this matrix to warp one image onto the other.
- Display and save the aligned and warped images.

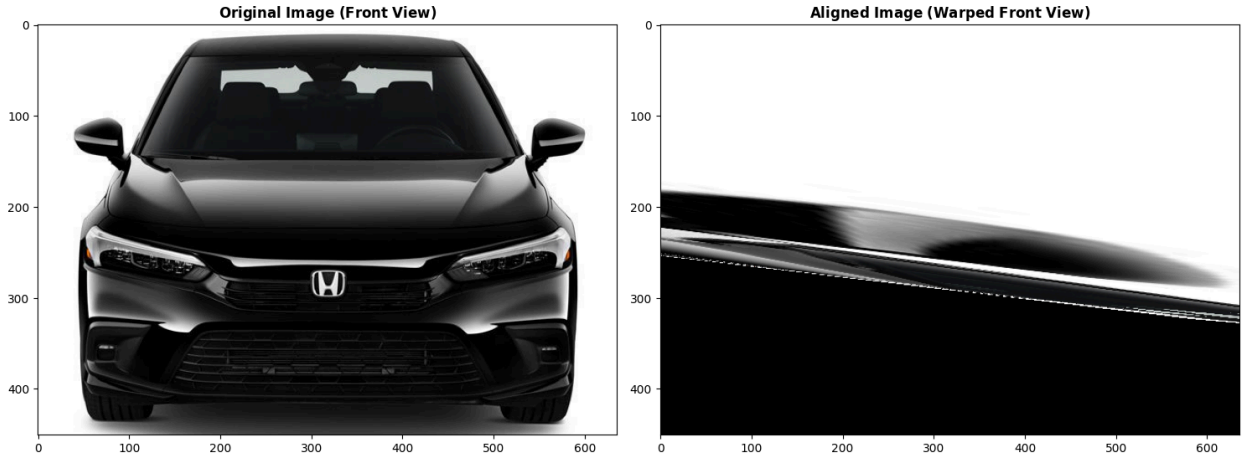
CODE:

This code performs image alignment by using the Scale-Invariant Feature Transform (SIFT) to detect key features and match them across two images, followed by calculating a homography matrix to align one image with the other. The process starts by converting each image to grayscale, as grayscale simplifies the feature detection process without needing color information. A SIFT object is created to detect and compute keypoints and descriptors in both images. Keypoints are distinctive points in each image, and descriptors describe the local visual information around each keypoint. After extracting the descriptors, the code employs the Brute-Force Matcher (BFMatcher) with the Euclidean distance (L2 norm) metric and cross-checking to find the best matching features between the two images. The matches are sorted based on distance to prioritize closer matches, which usually indicate better alignment of similar features.

For the matching points, the coordinates of the best feature matches from both images are stored in two arrays. These matching points are then used to compute the homography matrix using the RANSAC (Random Sample Consensus) method, which calculates the transformation needed to warp the first image to align with the second. This transformation corrects for differences in orientation or perspective between the two images. Once the homography matrix is determined, the first image is warped to match the perspective of the second image. The warped (or aligned) image is then displayed alongside the original image for visual comparison, showing how well the images are aligned. Finally, the aligned image is saved to a specified file path for future reference or use.

```python
#Function to extract keypoints and descriptors using SIFT.
def extract_features(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    sift = cv2.SIFT_create()
    kp, des = sift.detectAndCompute(gray, None)
    return kp, des

#Extract features from both images using SIFT.
kp1, des1 = extract_features(image1_np)
kp2, des2 = extract_features(image2_np)

#Function for matching descriptors using Brute-Force Matcher.
def match_descriptors_bf(des1, des2):
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(des1, des2)
    matches = sorted(matches, key=lambda x: x.distance)
    return matches

#Match descriptors between the two images.
matches = match_descriptors_bf(des1, des2)

#Extract location of good matches.
points1 = np.zeros((len(matches), 2), dtype=np.float32)
points2 = np.zeros((len(matches), 2), dtype=np.float32)

for i, match in enumerate(matches):
    points1[i, :] = kp1[match.queryIdx].pt
    points2[i, :] = kp2[match.trainIdx].pt

#Compute the homography matrix.
H, mask = cv2.findHomography(points1, points2, cv2.RANSAC)

#Warp the first image to align with the second image.
height, width = image2_np.shape[:2]
aligned_image = cv2.warpPerspective(image1_np, H, (width, height))

#Display the original and aligned images.
plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
plt.title(r'$\bf{Original\ Image\ (Front\ View)}$')
plt.imshow(image1_np)
plt.subplot(1, 2, 2)
plt.title(r'$\bf{Aligned\ Image\ (Warped\ Front\ View)}$')
plt.imshow(aligned_image)
plt.tight_layout()
plt.show()

#Save the aligned image.
aligned_image_path = '/content/drive/MyDrive/Aligned_Honda_Civic_Frontview.jpg'
cv2.imwrite(aligned_image_path, aligned_image)
```

RESULT:



Step 5: Performance Analysis

1. **Compare the Results**:
   - Analyze the performance of **SIFT**, **SURF**, and **ORB** in terms of keypoint detection accuracy, number of keypoints detected, and speed.
   - Comment on the effectiveness of **Brute-Force Matcher** versus **FLANN Matcher** for feature matching.
2. **Write a Short Report**:
   - Include your observations and conclusions on the best feature extraction and matching technique for the given images.

CODE:

```
#Install pandas library.
!pip install pandas
```

```python
#Import Libraries.
import cv2
import numpy as np
import time
import pandas as pd

#Function to analyze keypoints and descriptors for a given method.
def analyze_feature_extraction(method, image1, image2):
    start_time = time.time()

    if method == 'SIFT':
        sift = cv2.SIFT_create()
        kp1, des1 = sift.detectAndCompute(image1, None)
        kp2, des2 = sift.detectAndCompute(image2, None)

    elif method == 'SURF':
        surf = cv2.xfeatures2d.SURF_create()
        kp1, des1 = surf.detectAndCompute(image1, None)
        kp2, des2 = surf.detectAndCompute(image2, None)

    elif method == 'ORB':
        orb = cv2.ORB_create()
        kp1, des1 = orb.detectAndCompute(image1, None)
        kp2, des2 = orb.detectAndCompute(image2, None)

    end_time = time.time()

    #Calculate the number of keypoints detected and time taken.
    num_keypoints1 = len(kp1)
    num_keypoints2 = len(kp2)
    time_taken = end_time - start_time

    return num_keypoints1, num_keypoints2, time_taken, des1, des2

#Analyze each feature extraction method.
image1_gray = cv2.cvtColor(image1_np, cv2.COLOR_BGR2GRAY)
image2_gray = cv2.cvtColor(image2_np, cv2.COLOR_BGR2GRAY)

methods = ['SIFT', 'SURF', 'ORB']
results = {method: analyze_feature_extraction(method, image1_gray, image2_gray) for method in methods}

#Create a list for feature detection results.
feature_detection_data = []

for method, data in results.items():
    num_kp1, num_kp2, time_taken, _, _ = data
    feature_detection_data.append({
        "Method": method,
        "Keypoints Detected (Image 1)": num_kp1,
        "Keypoints Detected (Image 2)": num_kp2,
        "Time Taken (s)": time_taken
    })

#Create DataFrame for feature detection results.
feature_detection_df = pd.DataFrame(feature_detection_data)

#Display results for feature detection.
print("Feature Detection Performance Analysis:")
print(feature_detection_df.to_string(index=False))
```

```python
#Function for matching descriptors using FLANN Matcher.
def match_descriptors_flann(des1, des2):
    des1 = des1.astype(np.float32)
    des2 = des2.astype(np.float32)

    index_params = dict(algorithm=1, trees=5)
    search_params = dict(checks=50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)

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

    good_matches = []
    for m, n in matches:
        if m.distance < 0.7 * n.distance:
            good_matches.append(m)

    return good_matches

#Function to match descriptors and analyze matching performance.
def analyze_matching(des1, des2):
    start_time_bf = time.time()
    matches_bf = match_descriptors_bf(des1, des2)
    end_time_bf = time.time()

    start_time_flann = time.time()
    matches_flann = match_descriptors_flann(des1, des2)
    end_time_flann = time.time()

    bf_time = end_time_bf - start_time_bf
    flann_time = end_time_flann - start_time_flann

    return len(matches_bf), bf_time, len(matches_flann), flann_time

#Analyze matching performance for each method.
matching_results = {}
for method, data in results.items():
    _, _, _, des1, des2 = data
    des1 = des1.astype(np.float32)
    des2 = des2.astype(np.float32)

    matching_results[method] = analyze_matching(des1, des2)

#Create a list for matching performance results.
matching_performance_data = []

for method, data in matching_results.items():
    num_matches_bf, time_bf, num_matches_flann, time_flann = data
    matching_performance_data.append({
        "Method": method,
        "Matches (Brute-Force)": num_matches_bf,
        "Time Taken (BF)": time_bf,
        "Matches (FLANN)": num_matches_flann,
        "Time Taken (FLANN)": time_flann
    })

#Create DataFrame for matching performance results.
matching_performance_df = pd.DataFrame(matching_performance_data)

#Display results for matching performance.
print("\nFeature Matching Performance Analysis:")
print(matching_performance_df.to_string(index=False))
```

This code performs feature extraction and matching analysis using various computer vision techniques to compare two images. First, it imports necessary libraries such as OpenCV, NumPy, and Pandas for image processing and data handling. The code then defines a function, `analyze_feature_extraction`, which detects keypoints and computes descriptors for two input images using one of three methods: SIFT, SURF, or ORB. These methods vary in approach and efficiency for extracting features from images. Each method's time for detection and the number of keypoints found in each image are measured and stored. The input images are converted to grayscale to simplify processing, as many feature extraction algorithms work best with single-channel images. The results are compiled into a DataFrame, which includes the number of keypoints detected in each image and the time taken for each method, allowing for comparison.

In addition to feature extraction, the code defines a `match_descriptors_flann` function to match feature descriptors using the FLANN (Fast Library for Approximate Nearest Neighbors) Matcher, optimized for high-speed descriptor matching. The matching criteria are filtered using Lowe's ratio test to retain only strong matches. Another function, `analyze_matching`, compares performance between two matching approaches: Brute-Force and FLANN, recording the number of matches and time taken for each approach. Lastly, the results of feature matching performance, including the number of matches and time taken for both Brute-Force and FLANN methods, are stored in a DataFrame and displayed. This analysis provides insights into the efficiency and effectiveness of each feature extraction and matching method, which is crucial in applications such as image recognition, object detection, and computer vision tasks where precise feature matching is essential.

RESULT:

```
Feature Detection Performance Analysis:
Method  Keypoints Detected (Image 1)  Keypoints Detected (Image 2)  Time Taken (s)
  SIFT                           369                           310        0.175084
  SURF                          1346                          1235        0.905215
   ORB                           500                           500        0.022116

Feature Matching Performance Analysis:
Method  Matches (Brute-Force)  Time Taken (BF)  Matches (FLANN)  Time Taken (FLANN)
  SIFT                     84         0.008679                9            0.013732
  SURF                    281         0.074744               34            0.046170
   ORB                    151         0.010586                5            0.014428
```

Submission:

- A PDF or markdown document (performance_analysis.pdf or performance_analysis.md).

Submission Guidelines:

- **GitHub Repository**:
  - Create a folder in your CSST106-Perception and Computer Vision repository named Feature-Extraction-Machine-Problem.
  - Upload all code, images, and reports to this folder.

- **File Naming Format**: **[SECTION-LASTNAME-MP3]** 4D-LASTNAME-MP3
  - 4D-BERNARDINO-SIFT.py
  - 4D-BERNARDINO-Matching.jpg

Additional Penalties:

- **Incorrect Filename Format**: -5 points
- **Late Submission**: -5 points per day
- **Cheating/Plagiarism**: Zero points for the entire task

Republic of the Philippines
Laguna State Polytechnic University
Province of Laguna

Rubric for Feature Extraction and Object Detection Machine Problem

Criteria	Excellent (90-100%)	Good (75-89%)	Satisfactory (60-74%)	Needs Improvement (0-59%)
Step 2: Feature Extraction (SIFT, SURF, ORB)	All feature extraction methods (SIFT, SURF, ORB) are implemented correctly. The extracted keypoints are clearly visualized and well explained. The code is well-commented, and outputs are saved properly.	Feature extraction is implemented correctly, but there may be minor visualization issues or explanations lacking depth.	At least two methods are implemented correctly, with basic explanations and some issues with visualization or code.	Feature extraction methods are incomplete, implemented incorrectly, or not explained well. Poor or no visualization provided.
Step 3: Feature Matching (Brute-Force and FLANN)	Both Brute-Force and FLANN matchers are implemented correctly, and keypoint matches are clearly visualized with detailed explanations. The matching performance for each method is analyzed.	Both matchers are implemented correctly, but there may be minor issues with the visualization, or the explanation lacks depth.	At least one matcher is implemented correctly, with basic explanations and minimal analysis of matching performance.	Feature matching methods are incomplete, implemented incorrectly, or poorly explained. Matches are not visualized, or results are unclear.
Step 4: Image Alignment Using Homography	The Homography matrix is computed correctly using matched keypoints, and the image is aligned and warped successfully. The output is visually accurate, and the process is well explained.	The Homography matrix is computed correctly, but the alignment has minor issues, or the explanation lacks depth.	The Homography matrix is computed, but there are significant alignment issues, or the explanation is basic.	Homography computation is incorrect or incomplete. Image alignment does not work as expected, or no explanation is provided.
Step 5: Performance Analysis	The performance analysis is thorough, comparing the accuracy and speed of SIFT, SURF, and ORB, and evaluating the effectiveness of Brute-Force and FLANN. The conclusion is insightful and well-supported.	The performance analysis is good but lacks some depth in comparing the methods or has minor gaps in the evaluation of the matchers.	The performance analysis is basic, with minimal comparison or weak conclusions. Some methods or matchers are not evaluated.	The performance analysis is incomplete or missing. Little to no comparison or evaluation of methods and matchers is provided.