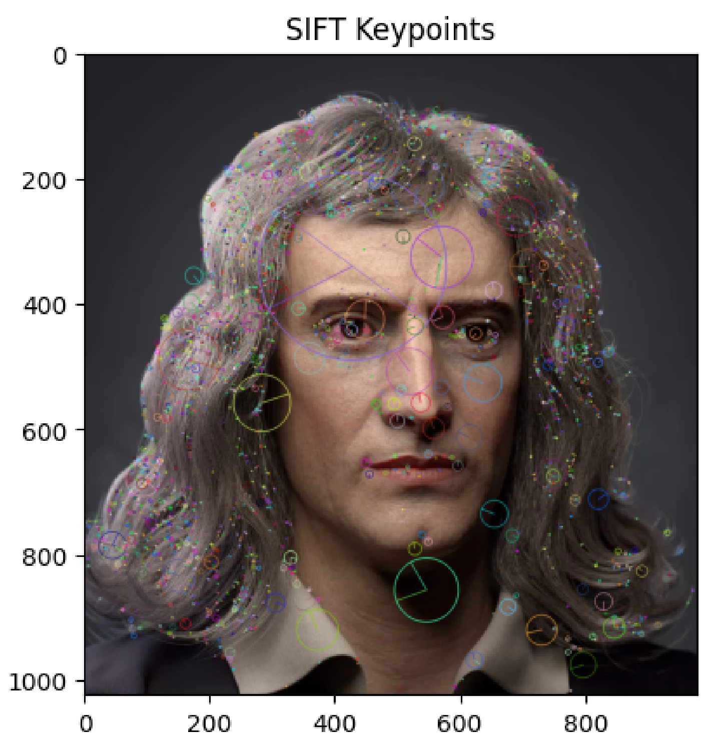
Module 2.0: Feature Extraction and Object Detection

SIFT Feature Extraction

```
import cv2
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('/content/IMAGES/NEWTON_1.webp')

# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Initialize the SIFT detector
sift = cv2.SIFT_create()

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

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

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



Approach:

- Image Loading: Load the image using OpenCV.
- Grayscale Conversion: Convert the image to grayscale for SIFT detection.
- SIFT Initialization: Initialize SIFT (Scale-Invariant Feature Transform) detector.
- Keypoint Detection: Detect keypoints and compute descriptors.
- Draw Keypoints: Draw keypoints on the image.
- Display Image: Use Matplotlib to display the image with keypoints.

Observations:

- Keypoints Representation:
  - The circles indicate the location of keypoints detected in the image.
  - The size of each circle reflects the scale at which the keypoint was detected, indicating the level of detail.
- Orientation:
  - Lines extending from the circles show the orientation of each keypoint.
  - This orientation helps in achieving rotation invariance, allowing the keypoints to be matched even if the image is rotated.
- Distribution:
  - Keypoints are distributed across various regions of the image, focusing on areas with distinct textures or patterns.
  - This distribution suggests that SIFT effectively captures features in areas with high contrast or unique structures.

Results:

- The output shows the original image with detected SIFT keypoints highlighted.
- This helps in understanding the significant points in the image useful for computer vision tasks like object recognition and image matching.

SURF Feature Extraction

```
!pip uninstall opencv-python opencv-contrib-python -y

!git clone https://github.com/opencv/opencv.git
!git clone https://github.com/opencv/opencv_contrib.git

!mkdir -p opencv/build
%cd opencv/build
!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 ..
!make -j8
!make install
```

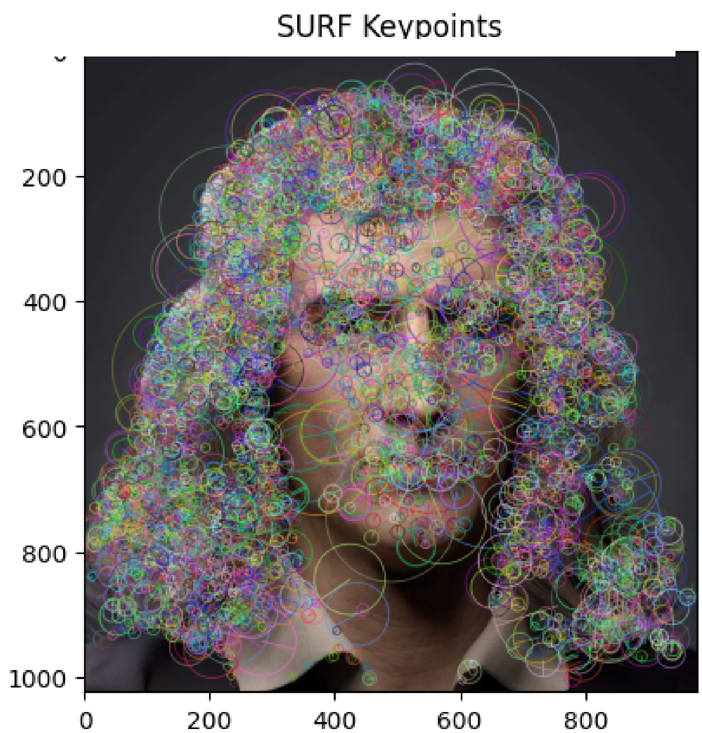
```
WARNING: Skipping opencv-python as it is not installed.
WARNING: Skipping opencv-contrib-python as it is not installed.
Cloning into 'opencv'...
remote: Enumerating objects: 336257, done.
```

```
remote: Counting objects: 100% (1025/1025), done.
remote: Compressing objects: 100% (823/823), done.
remote: Total 336257 (delta 435), reused 603 (delta 172), pack-reused 335232 (from 1)
Receiving objects: 100% (336257/336257), 527.23 MiB | 25.67 MiB/s, done.
Resolving deltas: 100% (234406/234406), done.
Updating files: 100% (7567/7567), done.
Cloning into 'opencv_contrib'...
remote: Enumerating objects: 41556, done.
remote: Counting objects: 100% (1315/1315), done.
remote: Compressing objects: 100% (950/950), done.
remote: Total 41556 (delta 495), reused 911 (delta 290), pack-reused 40241 (from 1)
Receiving objects: 100% (41556/41556), 149.98 MiB | 20.91 MiB/s, done.
Resolving deltas: 100% (25624/25624), done.
/content/opencv/build
-- The CXX compiler identification is GNU 11.4.0
-- The C compiler identification is GNU 11.4.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detected processor: x86_64
-- Found PythonInterp: /usr/bin/python3 (found suitable version "3.10.12", minimum required is "3
-- Found PythonLibs: /usr/lib/x86_64-linux-gnu/libpython3.10.so (found suitable exact version "3.
-- Looking for ccache - not found
-- Performing Test HAVE_CXX_FSIGNED_CHAR
-- Performing Test HAVE_CXX_FSIGNED_CHAR - Success
-- Performing Test HAVE_C_FSIGNED_CHAR
-- Performing Test HAVE_C_FSIGNED_CHAR - Success
-- Performing Test HAVE_CXX_W
-- Performing Test HAVE_CXX_W - Success
-- Performing Test HAVE_C_W
-- Performing Test HAVE_C_W - Success
-- Performing Test HAVE_CXX_WALL
-- Performing Test HAVE_CXX_WALL - Success
-- Performing Test HAVE_C_WALL
-- Performing Test HAVE_C_WALL - Success
-- Performing Test HAVE_CXX_WRETURN_TYPE
-- Performing Test HAVE_CXX_WRETURN_TYPE - Success
-- Performing Test HAVE_C_WRETURN_TYPE
-- Performing Test HAVE_C_WRETURN_TYPE - Success
-- Performing Test HAVE_CXX_WNON_VIRTUAL_DTOR
-- Performing Test HAVE_CXX_WNON_VIRTUAL_DTOR - Success
-- Performing Test HAVE_C_WNON_VIRTUAL_DTOR
-- Performing Test HAVE_C_WNON_VIRTUAL_DTOR - Failed
-- Performing Test HAVE_CXX_WADDRESS
-- Performing Test HAVE_CXX_WADDRESS - Success
-- Performing Test HAVE_C_WADDRESS
```

- The code block above is used to build and install OpenCV from source, enabling non-free modules and extra modules.
- The OPENCV_ENABLE_NONFREE=ON flag enables non-free modules, which include SIFT and SURF algorithms.
- The OPENCV_EXTRA_MODULES_PATH flag specifies the path to extra modules from OpenCV Contrib.

```
# Import OpenCV and check version
import cv2
```

```
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('/content/IMAGES/NEWTON_1.webp')

# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

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

# Detect SURF keypoints and descriptors
keypoints, descriptors = surf.detectAndCompute(gray, None)

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

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



Approach:

- Image Processing: Load image, convert to grayscale.
- SURF Detection: Initialize SURF (Speeded Up Robust Features) detector and compute keypoints and descriptors.
- Visualization: Draw keypoints on the image and display using Matplotlib.

Observations:

- Dense Coverage: The image has a high density of keypoints, indicating that SURF is identifying numerous features across various regions.

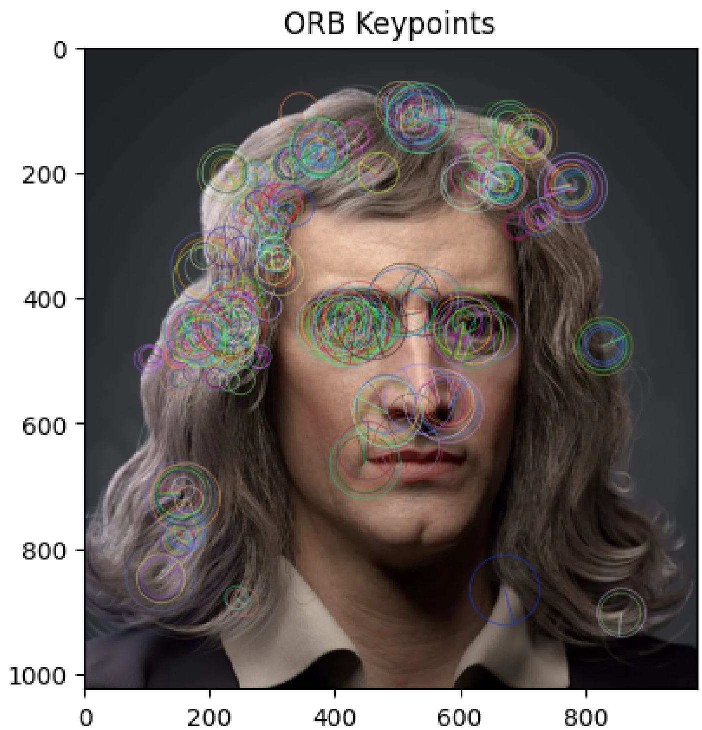
- Keypoint Distribution: Keypoints are distributed over the entire image, with a noticeable concentration around areas with high texture or contrast, such as edges of the face, hair, and clothing.
- Multiscale Detection: The circles of varying sizes suggest multiscale detection, where different scales of features are being captured.
- Color-Coding: The keypoints are shown in different colors, possibly representing orientation or scale, but primarily for visualization purposes.

Results:

- The output shows the original image with SURF keypoints highlighted.
- SURF keypoints represent distinctive features in the image, useful for tasks like object recognition and image matching.
- The visualization helps in understanding the distribution and characteristics of detected features in the image.

ORB Feature Extraction

```
# Load the image
image = cv2.imread('/content/IMAGES/NEWTON_1.webp')

# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

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

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

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

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



Approach:

- Image Processing: Load image, convert to grayscale.
- ORB Detection: Initialize ORB (Oriented FAST and rotated BRIEF) detector and compute keypoints and descriptors.
- Visualization: Draw keypoints on the image and display using Matplotlib.

Observations:

- Keypoint Distribution: Keypoints are spread across the image, with a focus on areas with distinct edges and corners, such as facial features and clothing edges.
- Uniform Size: The keypoints appear to be represented by circles of uniform size, indicating that ORB does not inherently capture scale information like SIFT or SURF.
- Orientation: ORB assigns an orientation to each keypoint, which is crucial for achieving rotation invariance. However, this orientation is not visually represented in the image.
- Efficiency: The number of keypoints is moderate, reflecting ORB's design for efficiency and speed, making it suitable for real-time applications.

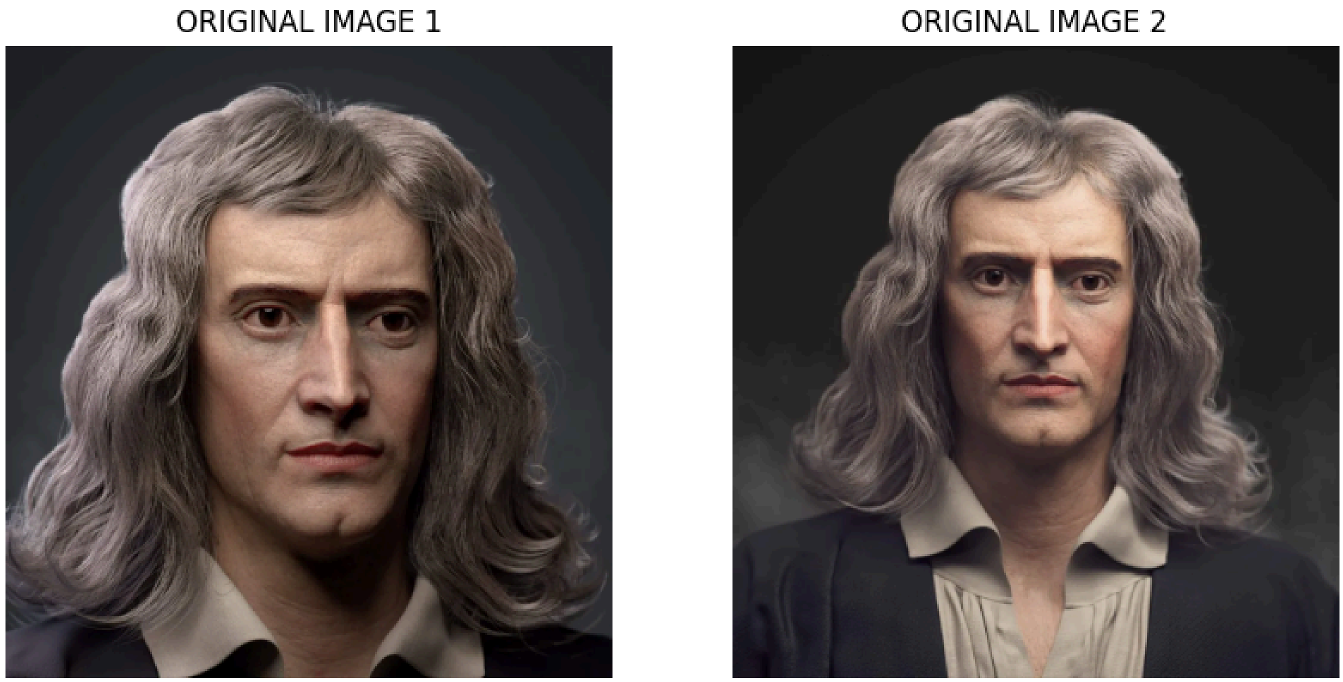
Results:

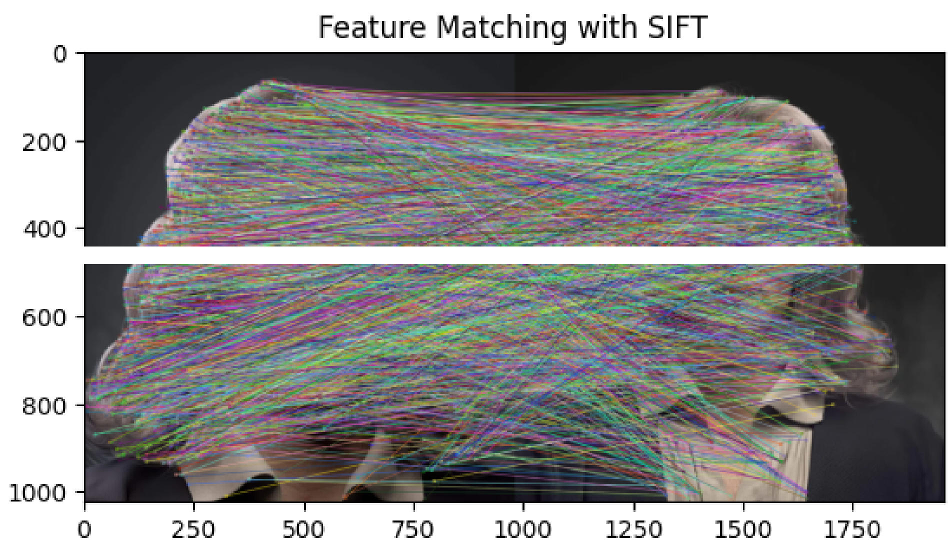
- The output shows the original image with ORB keypoints highlighted.
- ORB keypoints represent distinctive features in the image, useful for tasks like object recognition and image matching.
- The visualization helps in understanding the distribution and characteristics of detected features in the image.

Feature Matching using SIFT

```
def display_images(images, titles=None):
    """
    Display multiple images side by side.

    Args:
        images (list): List of images to display.
        titles (list, optional): List of titles for each image. Defaults to None.

    Returns:
        None
    """
    n = len(images)
    if titles is None:
        titles = [f"Image {i+1}" for i in range(n)]

    if n == 1:
        fig, ax = plt.subplots(1, 1, figsize=(5, 5))
        ax.imshow(images[0], cmap='gray')  # Use grayscale colormap
        ax.set_title(titles[0])
        ax.axis('off')
    else:
        fig, axs = plt.subplots(1, n, figsize=(n*5, 5))
        for i, (img, title, ax) in enumerate(zip(images, titles, axs)):
            ax.imshow(img, cmap='gray')  # Use grayscale colormap
            ax.set_title(title)
            ax.axis('off')

    plt.show()


# Load the image
image1 = cv2.imread('/content/IMAGES/NEWTON_1.webp')
image2 = cv2.imread('/content/IMAGES/NEWTON_2.webp')

# Initialize the SIFT detector
sift = cv2.SIFT_create()

# Detect keypoints and compute descriptors
keypoints1, descriptors1 = sift.detectAndCompute(image1, None)
keypoints2, descriptors2 = sift.detectAndCompute(image2, None)

# Initialize the matcher
bf = cv2.BFMatcher(cv2.NORM_L2)

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

# SOrt the matches by distance (best matches first)
matches = sorted(matches, key=lambda x: x.distance)

# Draw matches
image_matches = cv2.drawMatches(image1, keypoints1, image2, keypoints2, matches, None, flags=cv2.DrawMa

# Display original images
display_images([cv2.cvtColor(image1, cv2.COLOR_BGR2RGB), cv2.cvtColor(image2, cv2.COLOR_BGR2RGB)], ['OI

# Display the image with keypoints
plt.imshow(cv2.cvtColor(image_matches, cv2.COLOR_BGR2RGB)) # Assuming image_matches is in BGR format
```

```
plt.title("Feature Matching with SIFT")
plt.show()
```





Approach:

- Image Loading: Load two images for feature matching.
- SIFT Detection: Initialize SIFT detector and compute keypoints and descriptors for both images.
- Feature Matching: Use brute-force matcher (BFMatcher) to find correspondences between descriptors.
- Matching Visualization: Draw matches between the two images, highlighting corresponding keypoints.

Observations:

- Matched Keypoints:
    - Lines connect corresponding keypoints between two images, indicating successful matches.
    - These lines suggest that the SIFT algorithm has identified similar features in both images, despite potential differences in scale, rotation, or perspective.

- Accuracy of Matches:
  - The lines appear to be mostly straight and well-aligned, suggesting accurate matching of features.
  - The presence of multiple lines indicates a robust set of matches, which is essential for reliable image alignment or recognition.
- Distribution of Matches:
  - Matches are distributed across various regions of the images, covering different textures and structures.
  - This distribution shows SIFT's ability to capture and match features from diverse parts of the images.

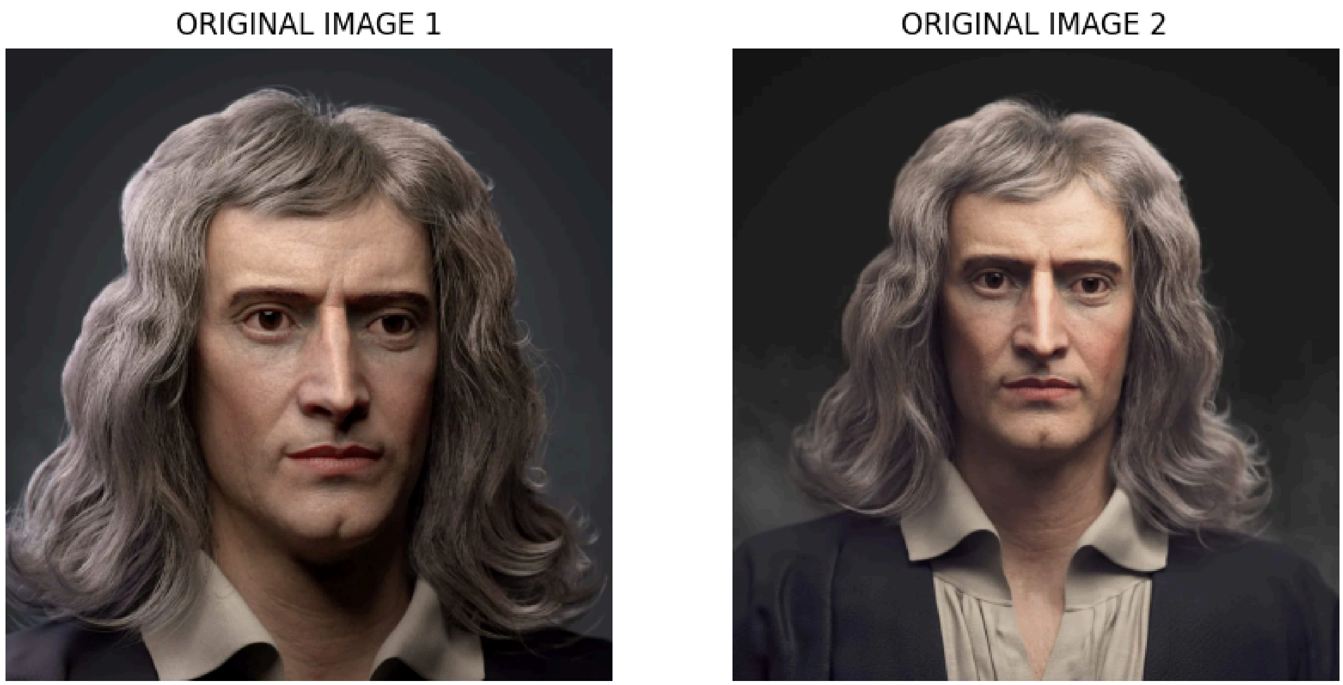
Results:

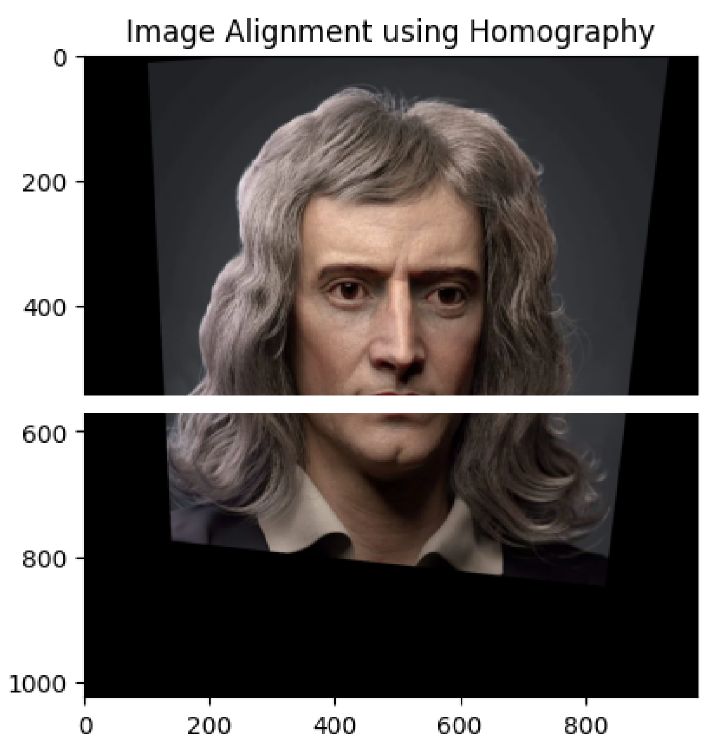
- The output displays the original images and a combined image showing the matching keypoints between them.
- Successful matching indicates similar features between the images, suggesting potential relationships between the two scenes.
- The visualization helps in understanding the alignment and common elements present in the two images.

Real-World Applications (Image Stitching using Homography)

```
import numpy as np

# Load the images
image1 = cv2.imread('/content/IMAGES/NEWTON_1.webp')
image2 = cv2.imread('/content/IMAGES/NEWTON_2.webp')

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

# Initialize the SIFT detector
sift = cv2.SIFT_create()

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

# Initialize the matcher
bf = cv2.BFMatcher(cv2.NORM_L2)

# Match the descriptors
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
```

```
    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 original images
    display_images([cv2.cvtColor(image1, cv2.COLOR_BGR2RGB), cv2.cvtColor(image2, cv2.COLOR_BGR2RGB)], ['OI

    # Display the image with keypoints
    plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
    plt.title("Image Alignment using Homography")
    plt.show()
```





Approach:

- Image Loading and Preprocessing: Load two images and convert them to grayscale.
- SIFT Feature Detection: Use SIFT to detect keypoints and compute descriptors for both images.
- Feature Matching: Use a brute-force matcher (BFMatcher) to find potential matches and apply Lowe's ratio test to filter out unreliable matches.
- Homography Estimation: Calculate the homography matrix (M) using RANSAC to find the geometric transformation between the two images based on good matches.
- Image Warping: Warp one image based on the homography matrix to align it with the other image.

Observations:

- Seamless Transition:
  - The stitched image shows a smooth transition between the two original images, indicating effective alignment and blending.
  - There are no visible seams or abrupt changes, suggesting that the homography transformation was accurately applied.
- Perspective Correction:
  - The images appear to be aligned correctly in terms of perspective, which is a key advantage of using homography for stitching.
  - This correction ensures that the combined image maintains a coherent and realistic appearance.
- Feature Alignment:
  - Key features, such as edges and patterns, are well-aligned across the stitched images, demonstrating successful feature matching and transformation.
  - This alignment is crucial for creating a visually appealing and accurate composite image.
- Overall Composition:
  - The final stitched image appears to be a single, unified scene, which is the goal of image stitching.
  - The use of homography allows for the transformation of one image plane to another, accommodating differences in viewpoint and perspective.

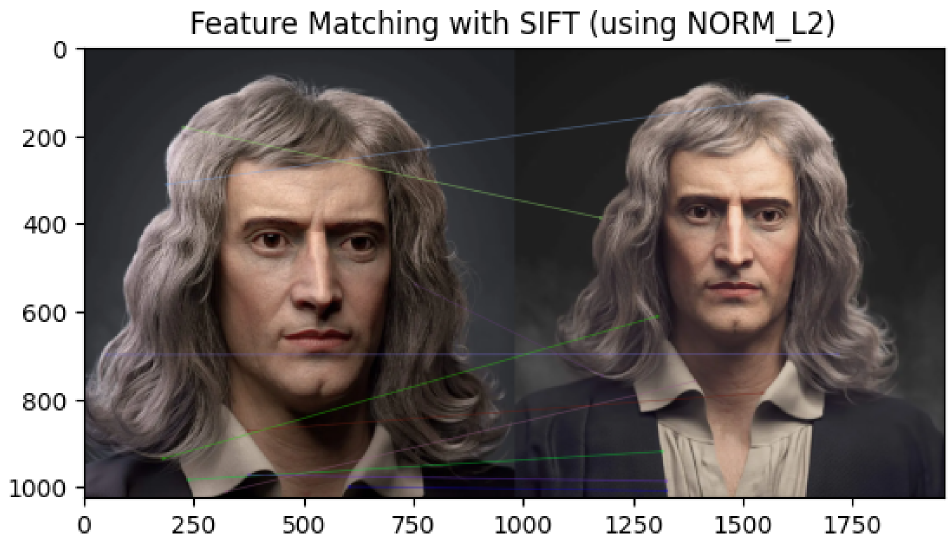
Results:

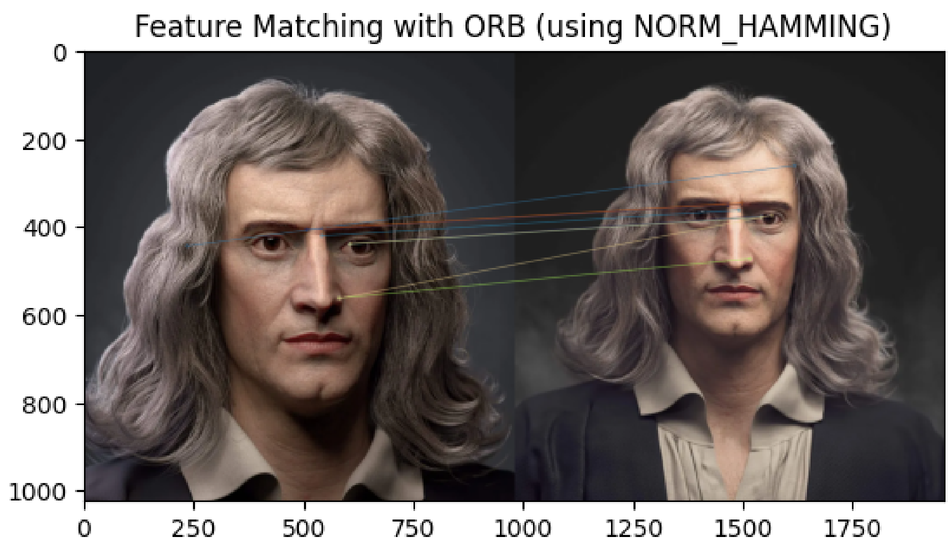
- The output displays the original images and the warped image, which is aligned with the second image.
- The aligned image demonstrates the successful transformation based on the computed homography, effectively aligning the two images.
- This approach is useful for image stitching, object tracking, and other applications where aligning images is necessary.

Combining SIFT and ORB

```
# Use SIFT and ORB to extract features from two images

# Load the image
image1 = cv2.imread('/content/IMAGES/NEWTON_1.webp')
image2 = cv2.imread('/content/IMAGES/NEWTON_2.webp')
```

```
# ----------------------------------------------------------------------------#

# Initialize the SIFT detector
sift = cv2.SIFT_create()

# Detect keypoints and compute descriptors
keypoints1_sift, descriptors1_sift = sift.detectAndCompute(image1, None)
keypoints2_sift, descriptors2_sift = sift.detectAndCompute(image2, None)

# ----------------------------------------------------------------------------#

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

# Detect keypoints and compute descriptors
keypoints1_orb, descriptors1_orb = orb.detectAndCompute(image1, None)
keypoints2_orb, descriptors2_orb = orb.detectAndCompute(image2, None)

# ----------------------------------------------------------------------------#

# Initialize the Brute-Force matcher for SIFT
bf_sift = cv2.BFMatcher(cv2.NORM_L2)  # Use NORM_L2 for SIFT

# Match the descriptors for SIFT
matches_sift = bf_sift.match(descriptors1_sift, descriptors2_sift)

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

# Draw matches for SIFT
image_matches_sift = cv2.drawMatches(image1, keypoints1_sift, image2, keypoints2_sift, matches_sift[:1(

# Display the image with SIFT keypoint matches
plt.imshow(cv2.cvtColor(image_matches_sift, cv2.COLOR_BGR2RGB))
plt.title("Feature Matching with SIFT (using NORM_L2)")
plt.show()

# ----------------------------------------------------------------------------#

# Initialize the Brute-Force matcher for ORB
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)  # Use NORM_HAMMING for ORB

# Match the descriptors for ORB
matches_orb = bf.match(descriptors1_orb, descriptors2_orb)

# Sort the matches by distance (best matches first)
matches_orb = sorted(matches_orb, key=lambda x: x.distance)

# Draw matches for ORB
image_matches_orb = cv2.drawMatches(image1, keypoints1_orb, image2, keypoints2_orb, matches_orb[:10], N

# Display the image with ORB keypoint matches
plt.imshow(cv2.cvtColor(image_matches_orb, cv2.COLOR_BGR2RGB))
plt.title("Feature Matching with ORB (using NORM_HAMMING)")
plt.show()
```





Approach:

- Image Loading: Load two images for feature extraction and matching.
- SIFT Feature Detection: Initialize the SIFT detector, compute keypoints and descriptors for both images.
- ORB Feature Detection: Initialize the ORB detector, compute keypoints and descriptors for both images.
- SIFT Feature Matching: Use Brute-Force matcher with NORM_L2 distance to match SIFT descriptors between the two images.
- ORB Feature Matching: Use Brute-Force matcher with NORM_HAMMING distance to match ORB descriptors between the two images.
- Visualization: Draw the top 10 matches for both SIFT and ORB and display the results.

Observations:

- SIFT
  - Feature Matching: The lines between the images represent matched features detected by the SIFT algorithm. These lines indicate points of similarity between the two images.
  - Line Density and Accuracy: The matching lines are dense in certain areas, indicating where the algorithm found the most features in common, such as facial features.

  - Algorithm Performance: SIFT is known for its effectiveness in matching key points between images, especially under changes in scale and rotation. The image likely demonstrates SIFT's robustness in identifying consistent features.
  - NORM_L2 Matching: The use of NORM_L2 indicates that a specific distance metric was used to match features, emphasizing Euclidean distance, which is suitable for SIFT descriptor vectors.
- ORB
  - Feature Matching: The image displays lines connecting matched features between two images. These lines indicate correspondences found by the ORB feature extraction algorithm.
  - Accuracy: The lines demonstrate the identified similarities between corresponding points in the two images, showing how well ORB can detect and match features in similar images.
  - Application: ORB is known for being efficient and fast, making it suitable for real-time applications. It combines the FAST keypoint detector and the BRIEF descriptor with added orientation component.
  - Line Quality: Some lines might be less accurate or mismatched, indicating the challenges in achieving perfect feature matching, particularly with changes in illumination, perspective, or expressions.

Results:

- The output displays two images, one showing the top 10 SIFT matches between the two input images, and the other showing the top 10 ORB matches.
- The visualizations help in understanding the different features detected by SIFT and ORB, as well as the quality