# Machine Problem No. 3: Feature Extraction and Object Detection

September 28, 2024

## 1 Objective

The objective of this machine problem is to implement and compare the three feature extraction method (**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.

## 2 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.

## 3 Task Breakdown

### 3.1 Step 1: Load Images

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

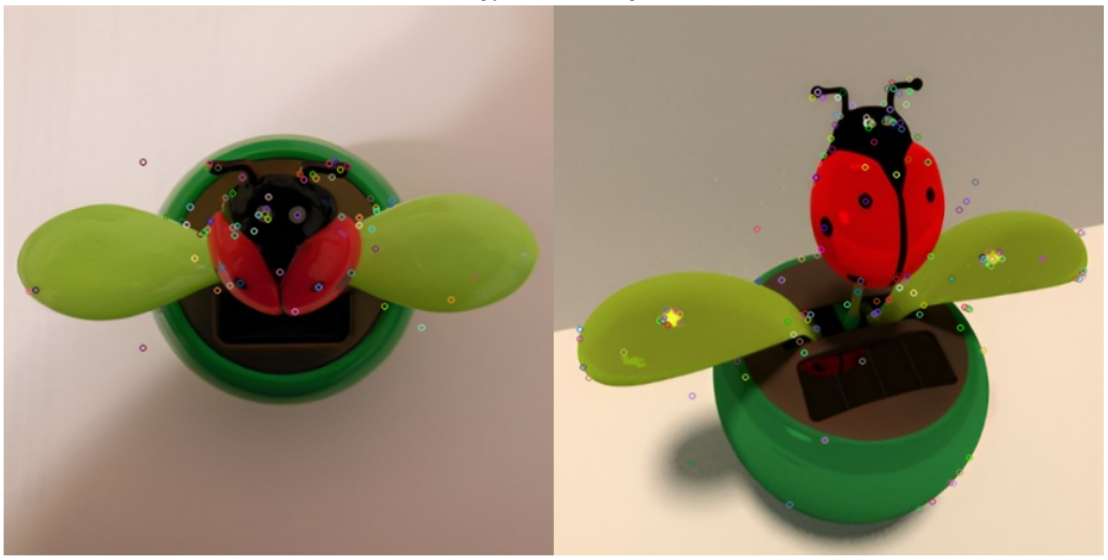
[10]:
```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the images
img1 = cv2.imread('/content/image_1.jpg')
img2 = cv2.imread('/content/image_2.jpg')
```

### 3.2 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

[23]:
```python
# Import the necessary libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the two images
image1 = cv2.imread("/content/image_1.jpg")
image2 = cv2.imread("/content/image_2.jpg")

# Convert to grayscale
gray_image1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
gray_image2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

# Initialize SIFT detector
sift = cv2.SIFT_create()

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

# Draw keypoints on both images
image1_with_keypoints = cv2.drawKeypoints(image1, keypoints1, None)
image2_with_keypoints = cv2.drawKeypoints(image2, keypoints2, None)

# Resize the images to the same size (e.g., 500x500 pixels)
image1_resized = cv2.resize(image1_with_keypoints, (500, 500))
image2_resized = cv2.resize(image2_with_keypoints, (500, 500))

# Combine both images side-by-side
combined_image = np.hstack((image1_resized, image2_resized))

# Enlarge the display by setting a larger figure size
plt.figure(figsize=(12, 6))
plt.imshow(cv2.cvtColor(combined_image, cv2.COLOR_BGR2RGB))
plt.title("SIFT Keypoints (Side-by-Side)")
plt.axis('off')
plt.show()

# Print the number of keypoints detected in each image
print(f"Number of keypoints detected in image 1: {len(keypoints1)}")
print(f"Number of keypoints detected in image 2: {len(keypoints2)}")
```

SIFT Keypoints (Side-by-Side)

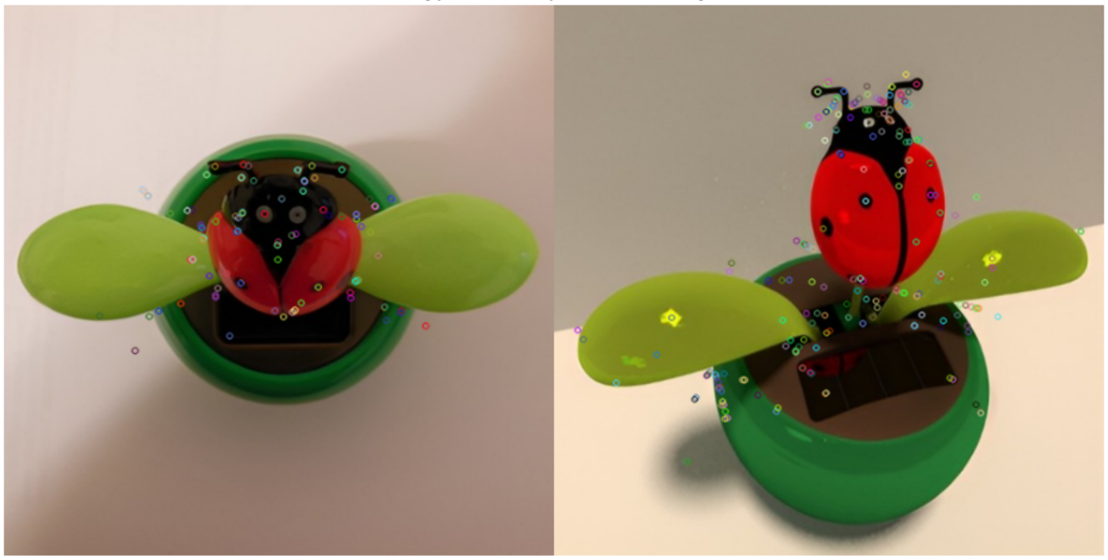


```
Number of keypoints detected in image 1: 82
Number of keypoints detected in image 2: 152
```

[31]:
```
# Import the necessary libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the two images
image1 = cv2.imread("/content/image_1.jpg")
image2 = cv2.imread("/content/image_2.jpg")

# Convert to grayscale
gray_image1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
gray_image2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

# Initialize SURF detector (adjust the Hessian threshold as needed)
surf = cv2.xfeatures2d.SURF_create(hessianThreshold=400)

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

# Draw keypoints on both images
image1_with_keypoints = cv2.drawKeypoints(image1, keypoints1, None)
image2_with_keypoints = cv2.drawKeypoints(image2, keypoints2, None)
```

```python
# Resize the images to the same size (e.g., 500x500 pixels)
image1_resized = cv2.resize(image1_with_keypoints, (500, 500))
image2_resized = cv2.resize(image2_with_keypoints, (500, 500))

# Combine both images side-by-side
combined_image = np.hstack((image1_resized, image2_resized))

# Enlarge the display by setting a larger figure size
plt.figure(figsize=(12, 6))
plt.imshow(cv2.cvtColor(combined_image, cv2.COLOR_BGR2RGB))
plt.title("SURF Keypoints Comparison (Side-by-Side)")
plt.axis('off')
plt.show()

# Print the number of keypoints detected in each image
print(f"Number of keypoints detected in image 1: {len(keypoints1)}")
print(f"Number of keypoints detected in image 2: {len(keypoints2)}")
```

SURF Keypoints Comparison (Side-by-Side)

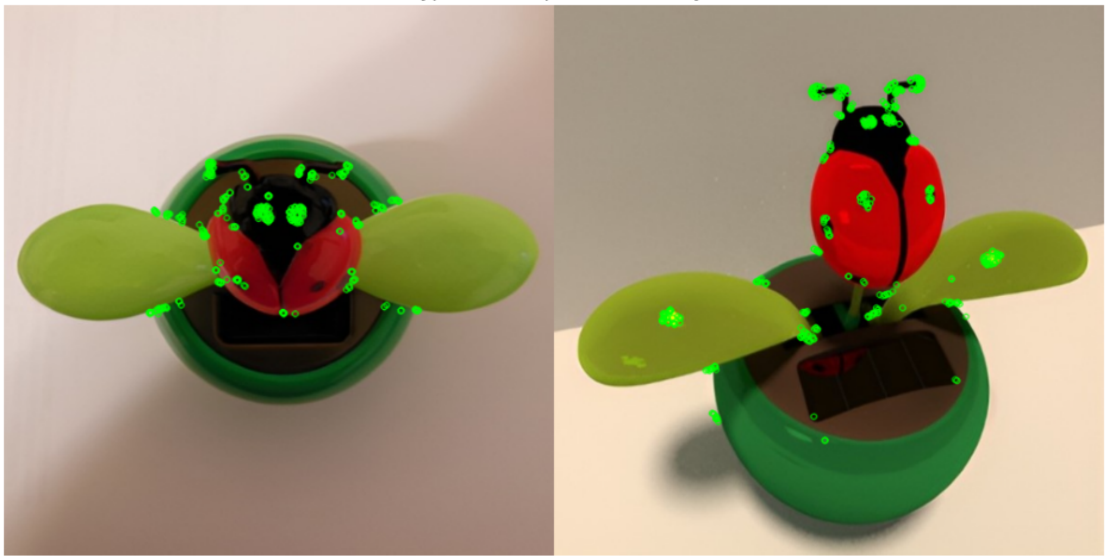


```
Number of keypoints detected in image 1: 92
Number of keypoints detected in image 2: 199
```

[32]:
```python
# Import the necessary libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the two images
```

```python
image1 = cv2.imread("/content/image_1.jpg")
image2 = cv2.imread("/content/image_2.jpg")

# Convert to grayscale
gray_image1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
gray_image2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

# Initialize ORB detector (set the number of keypoints to detect)
orb = cv2.ORB_create(nfeatures=500)

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

# Draw keypoints on both images
image1_with_keypoints = cv2.drawKeypoints(image1, keypoints1, None, color=(0,␣
↪255, 0))
image2_with_keypoints = cv2.drawKeypoints(image2, keypoints2, None, color=(0,␣
↪255, 0))

# Resize the images to the same size (e.g., 500x500 pixels)
image1_resized = cv2.resize(image1_with_keypoints, (500, 500))
image2_resized = cv2.resize(image2_with_keypoints, (500, 500))

# Combine both images side-by-side
combined_image = np.hstack((image1_resized, image2_resized))

# Enlarge the display by setting a larger figure size
plt.figure(figsize=(12, 6))
plt.imshow(cv2.cvtColor(combined_image, cv2.COLOR_BGR2RGB))
plt.title("ORB Keypoints Comparison (Side-by-Side)")
plt.axis('off')
plt.show()

# Print the number of keypoints detected in each image
print(f"Number of keypoints detected in image 1: {len(keypoints1)}")
print(f"Number of keypoints detected in image 2: {len(keypoints2)}")
```

ORB Keypoints Comparison (Side-by-Side)



```
Number of keypoints detected in image 1: 321
Number of keypoints detected in image 2: 476
```

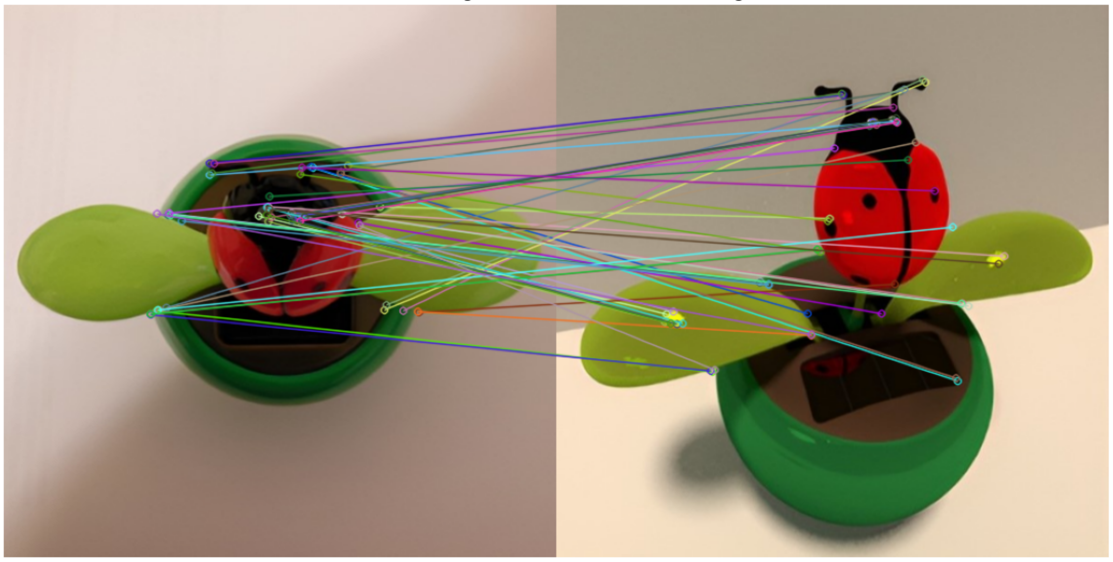
### 3.3 Step 3: Feature Matching with Brute-Force and FLANN

- 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.

[26]:
```python
# Import the necessary libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

def resize_with_aspect_ratio(image, target_width):
    """Resize an image while maintaining its aspect ratio."""
    h, w = image.shape[:2]
    aspect_ratio = w / h
    new_width = target_width
    new_height = int(new_width / aspect_ratio)
    return cv2.resize(image, (new_width, new_height))

# Load the two images
image1 = cv2.imread("/content/image_1.jpg")
image2 = cv2.imread("/content/image_2.jpg")
```

```
# Resize images to have the same width (e.g., 500 pixels) while preserving␣
 ↪aspect ratio
target_width = 500
image1_resized = resize_with_aspect_ratio(image1, target_width)
image2_resized = resize_with_aspect_ratio(image2, target_width)

# Convert to grayscale
gray_image1 = cv2.cvtColor(image1_resized, cv2.COLOR_BGR2GRAY)
gray_image2 = cv2.cvtColor(image2_resized, cv2.COLOR_BGR2GRAY)

# Initialize ORB detector (set the number of keypoints to detect)
orb = cv2.ORB_create(nfeatures=500)

# Detect keypoints and 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
matches = bf.match(descriptors1, descriptors2)

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

# Draw matches with lines connecting corresponding keypoints between images
matched_image = cv2.drawMatches(image1_resized, keypoints1, image2_resized,␣
 ↪keypoints2, matches[:50], None, flags=cv2.
 ↪DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the image with matches
plt.figure(figsize=(14, 7))
plt.imshow(cv2.cvtColor(matched_image, cv2.COLOR_BGR2RGB))
plt.title("Feature Matching with Brute Force Matcher using ORB")
plt.axis('off')
plt.show()

# Print the number of matches
print(f"Number of matches: {len(matches)}")
```

Feature Matching with Brute Force Matcher using ORB

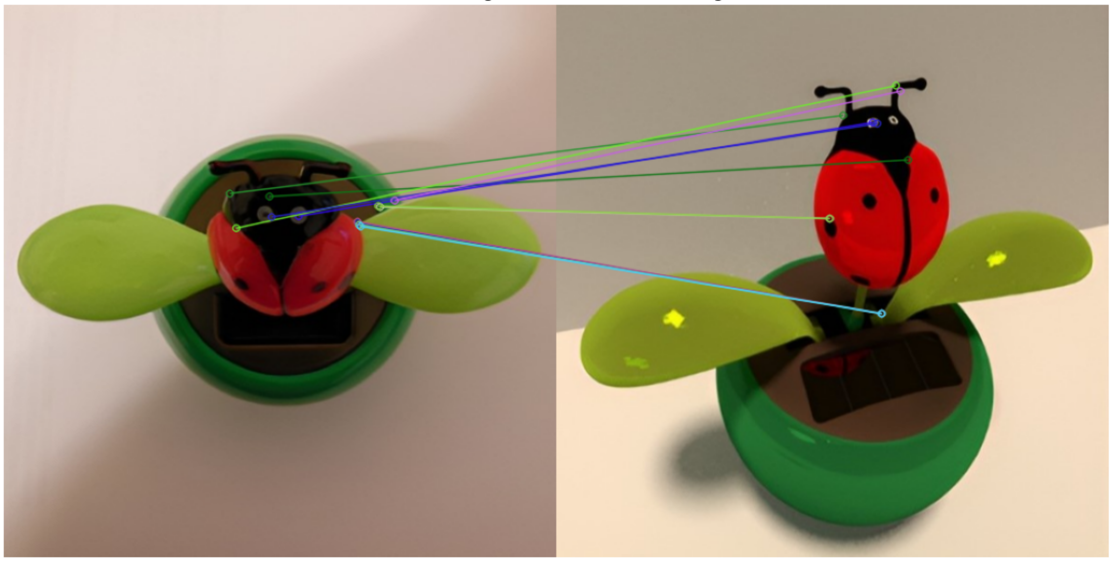


```
Number of matches: 86
```

[27]:
```python
# Import the necessary libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

def resize_with_aspect_ratio(image, target_width):
    """Resize an image while maintaining its aspect ratio."""
    h, w = image.shape[:2]
    aspect_ratio = w / h
    new_width = target_width
    new_height = int(new_width / aspect_ratio)
    return cv2.resize(image, (new_width, new_height))

# Load the two images
image1 = cv2.imread("/content/image_1.jpg")
image2 = cv2.imread("/content/image_2.jpg")

# Resize images to have the same width (e.g., 500 pixels) while preserving␣
 ↪aspect ratio
target_width = 500
image1_resized = resize_with_aspect_ratio(image1, target_width)
image2_resized = resize_with_aspect_ratio(image2, target_width)

# Convert to grayscale
gray_image1 = cv2.cvtColor(image1_resized, cv2.COLOR_BGR2GRAY)
gray_image2 = cv2.cvtColor(image2_resized, cv2.COLOR_BGR2GRAY)
```

```
# Initialize ORB detector (set the number of keypoints to detect)
orb = cv2.ORB_create(nfeatures=500)

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

# Set FLANN-based matcher parameters for binary descriptors
index_params = dict(algorithm=6,
                    table_number=12,
                    key_size=20,
                    multi_probe_level=2)

search_params = dict(checks=50)  # Higher number of checks increases precision,␣
 ↪but slower

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

# Match descriptors using K-Nearest Neighbors
matches = flann.knnMatch(descriptors1, descriptors2, k=2)

# Apply Lowe's ratio test to filter good matches
good_matches = []
for match in matches:
  # Ensure there are at least two matches for the current keypoint
  if len(match) == 2:
    m, n = match
    if m.distance < 0.75 * n.distance:
      good_matches.append(m)

# Draw matches with lines connecting corresponding keypoints between images
matched_image = cv2.drawMatches(image1_resized, keypoints1, image2_resized,␣
 ↪keypoints2, good_matches[:50], None, flags=cv2.
 ↪DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the image with matches
plt.figure(figsize=(14, 7))
plt.imshow(cv2.cvtColor(matched_image, cv2.COLOR_BGR2RGB))
plt.title("Feature Matching with FLANN Matcher using ORB")
plt.axis('off')
plt.show()

# Print the number of good matches
print(f"Number of good matches: {len(good_matches)}")
```

Feature Matching with FLANN Matcher using ORB



```
Number of good matches: 12
```

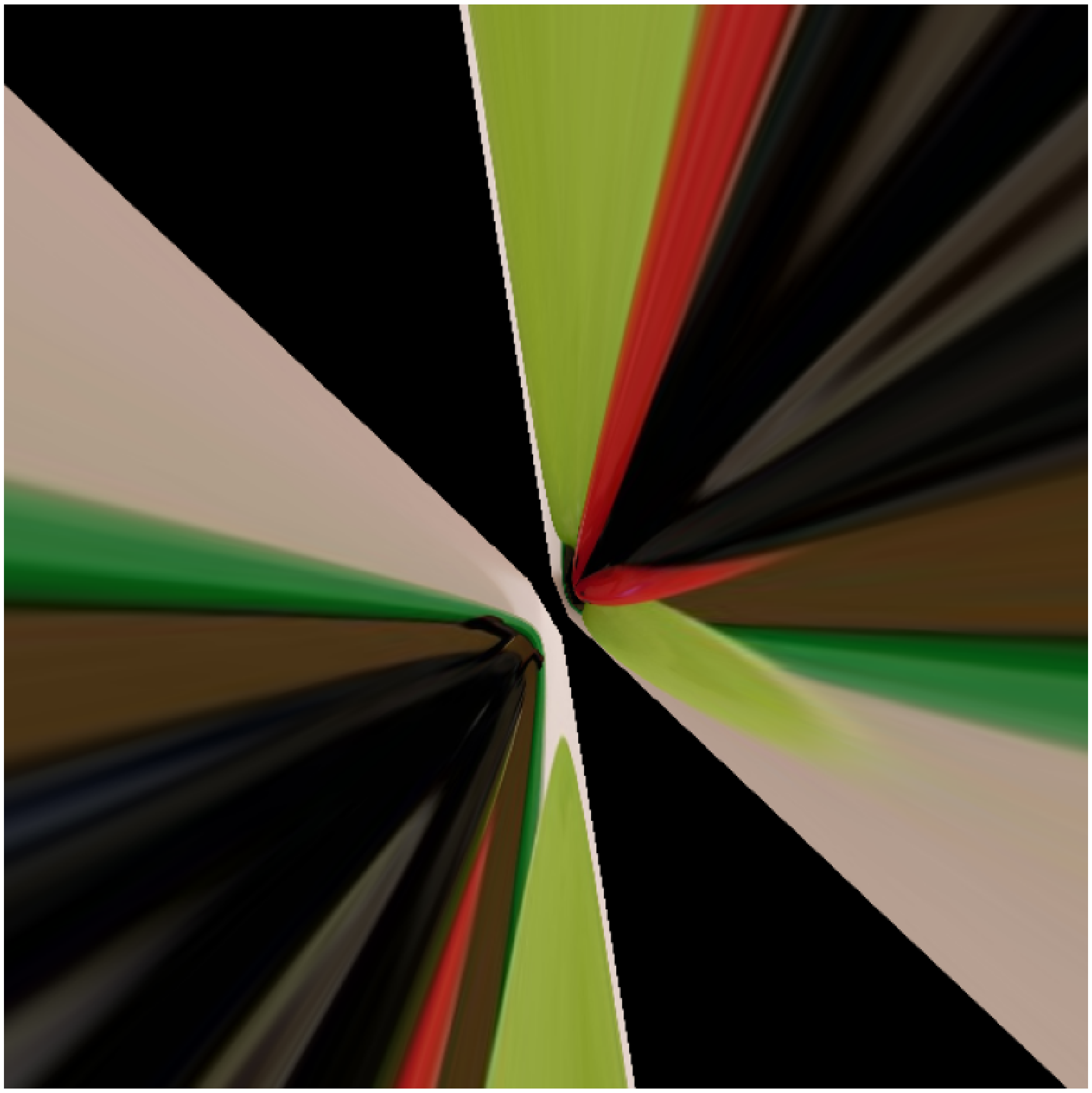
## 3.4 Step 4: Image Alignment Using Homography

- 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.

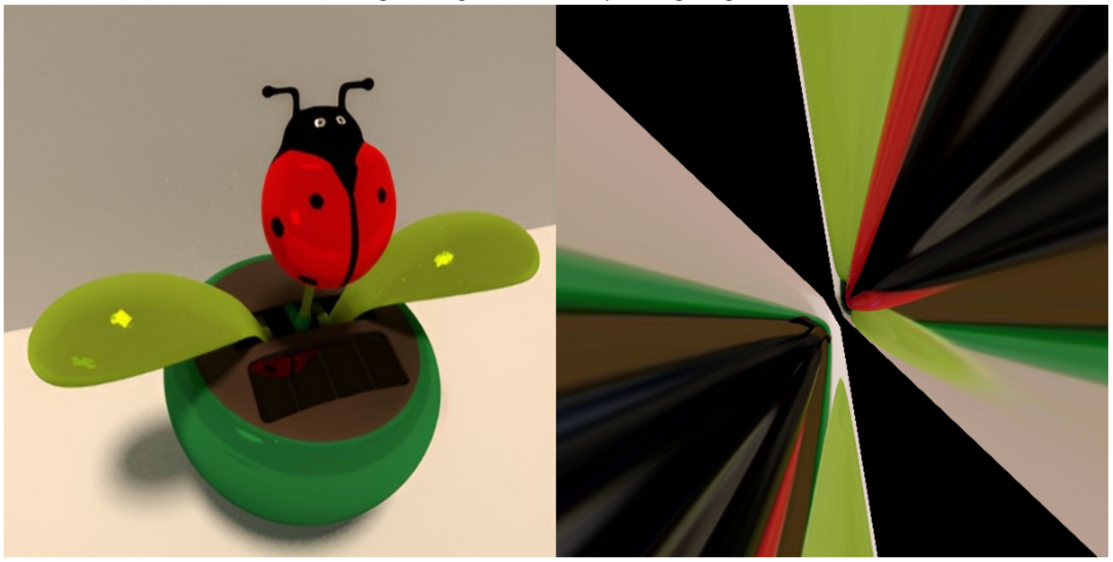
[28]:
```python
# Import the necessary libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the two images
image1 = cv2.imread("/content/image_1.jpg")
image2 = cv2.imread("/content/image_2.jpg")

# Convert to grayscale
gray_image1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
gray_image2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

# Initialize SIFT detector
sift = cv2.SIFT_create()

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

```
# Set FLANN-based matcher parameters
index_params = dict(algorithm=1, trees=5)  # Algorithm 1 is for KD-tree
search_params = dict(checks=50)  # Higher number of checks for better precision

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

# Match descriptors using K-Nearest Neighbors
matches = flann.knnMatch(descriptors1, descriptors2, k=2)

# Apply Lowe's ratio test to filter good matches
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)

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

# Warp the first image (image1) to align with the second image (image2) using␣
↪the homography matrix
height, width = image2.shape[:2]
warped_image = cv2.warpPerspective(image1, homography_matrix, (width, height))

# Display the aligned image
plt.figure(figsize=(10, 10))
plt.imshow(cv2.cvtColor(warped_image, cv2.COLOR_BGR2RGB))
plt.title("Warped Image")
plt.axis('off')
plt.show()

# Save the warped image
cv2.imwrite("warped_image.png", warped_image)

# Display both images (original and aligned) side-by-side for comparison
combined_image = np.hstack((image2, warped_image))
plt.figure(figsize=(14, 7))
plt.imshow(cv2.cvtColor(combined_image, cv2.COLOR_BGR2RGB))
plt.title("Original Image (Left) and Warped Image (Right)")
plt.axis('off')
plt.show()

# Save the side-by-side comparison
cv2.imwrite("comparison_image.png", combined_image)
```

Warped Image



Original Image (Left) and Warped Image (Right)



[28]:
```
True
```

## 3.5 Step 5: Performance Analysis

### 3.5.1 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.

**Performance of SIFT, SURF, and ORB** **Keypoint Detection Accuracy** 1. **SIFT (Scale-Invariant Feature Transform)**: - SIFT is accurate in detecting distinctive keypoints in images. It is invariant to scaling, rotation, and illumination changes. - It detected 82 keypoints in image 1 and 152 keypoints in image 2. - The quality of the keypoints is typically high, meaning they are often better localized and more distinct, even if fewer keypoints are detected compared to the other algorithms. 2. **SURF (Speeded Up Robust Features)**: - Similar to SIFT in terms of robustness to scale and rotation, SURF uses a more efficient detector. - SURF detected 92 keypoints in image 1 and 199 keypoints in image 2, generally detecting more keypoints than SIFT. - In terms of accuracy, SURF is also robust, although it might sometimes miss finer features compared to SIFT. 3. **ORB (Oriented FAST and Rotated BRIEF)**: - ORB is a fast feature detector and descriptor that uses binary features. - It detected significantly more keypoints than SIFT and SURF, with 321 keypoints in image 1 and 476 keypoints in image 2. - While it detects a large number of keypoints, its accuracy in complex or highly detailed images might not be as strong as SIFT or SURF, as it may produce noisier keypoints and lower-quality matches.

**Number of Keypoints Detected** 1. **SIFT (Scale-Invariant Feature Transform)**: - Detects fewer keypoints overall, with 82 keypoints in image 1 and 152 keypoints in image 2. It focuses on quality rather than quantity. 2. **SURF (Speeded Up Robust Features)**: - Detects more keypoints than SIFT, with 92 keypoints in image 1 and 199 keypoints in image 2. SURF aims for a balance between speed and quality. 3. **ORB (Oriented FAST and Rotated BRIEF)**: - Detects a much larger number of keypoints, with 321 keypoints in image 1 and 476 keypoints in image 2. This makes ORB ideal for tasks where a large number of features are needed, but it may come at the cost of lower keypoint distinctiveness.

**Speed** 1. **SIFT (Scale-Invariant Feature Transform)**: - Took 2 seconds to run. It is slower compared to ORB because it involves more computationally intensive operations such as scale-space construction and feature localization. 2. **SURF (Speeded Up Robust Features)**: - Also took 2 seconds. Although designed to be faster than SIFT, the performance was similar in this case. However, SURF can often be faster on larger images or higher-dimensional feature sets due to its simpler filter-based approach. 3. **ORB (Oriented FAST and Rotated BRIEF)**: - Was the fastest, completing in 1 second. This makes ORB a highly efficient algorithm, suitable for real-time applications. Its binary descriptor and the use of the FAST detector make it significantly faster than both SIFT and SURF.

**Comparison: Brute-Force Matcher vs. FLANN Matcher for Feature Matching**

1. **Brute-Force Matcher (BFMatcher)**:

- **Number of Matches**: 86
- **Explanation**: The Brute-Force Matcher works by comparing each descriptor from one image to every descriptor in the other image, calculating a distance metric (like Hamming distance for binary descriptors). It is straightforward and exhaustive.

**Effectiveness:** - Pros: - High number of matches: It found 86 matches, which is a large number for matching ORB descriptors. - Simple and thorough: BFMatcher guarantees finding the closest matches because it compares every descriptor with all others. - Cons: - Quality: Since it doesn't apply sophisticated filtering, the matches might include more false positives, where points that don't actually correspond to each other are considered matches.- Speed: It is slower, especially for large datasets, because it performs an exhaustive search through all descriptors.

2. **FLANN Matcher (Fast Library for Approximate Nearest Neighbors)**:

- **Number of Good Matches**: 12
- **Explanation**: The FLANN Matcher is designed for faster and more efficient matching, especially when working with large datasets or high-dimensional descriptors. It uses approximate nearest neighbor algorithms and is typically more suited for floating-point descriptors like those from SIFT and SURF. For ORB, which uses binary descriptors, FLANN is less commonly used but still effective with proper configuration.

**Effectiveness**: - Pros: - **Higher match quality**: By applying Lowe's ratio test, the FLANN matcher filters out poor matches, resulting in fewer but more reliable matches (12 good matches). This means it focuses on high-quality correspondences rather than quantity. - **Speed**: FLANN is optimized for faster matching, which makes it more efficient for large datasets. Though in this case, with ORB (which uses binary descriptors), the improvement may not be as dramatic as with floating-point descriptors. - Cons: - **Lower number of matches**: The number of good matches is much smaller (12), which may limit its effectiveness when a higher number of matches is needed, or when the images have a lot of detailed features to be compared. - **Suitability for ORB**: FLANN is not inherently designed for binary descriptors, so it may not be as effective as it would be for floating-point descriptors (like SIFT/SURF).

### 3.5.2 Write a Short Report

- Include your observations and conclusions on the best feature extraction and matching technique for the given images.

#### Feature Extraction

- SIFT provides the most accurate keypoint detection but detects fewer keypoints and is slower than ORB. It is suitable for applications requiring precise feature matching.
- SURF balances speed and accuracy which offers a good number of keypoints with decent accuracy. It's suitable when both speed and accuracy are needed, though not necessarily real-time.
- ORB is the fastest and detects the most keypoints. It is ideal for applications that require real-time performance and can tolerate a slight reduction in keypoint quality. It's less robust in cases where high precision is essential.

#### Matching Technique

- **Brute-Force Matcher**: It excels when you need to ensure that all possible matches are found, making it more comprehensive but potentially less precise. It works well with ORB's binary descriptors, but the number of matches might include more false positives.
- **FLANN Matcher**: More precise and efficient due to Lowe's ratio test, but it produces fewer matches. This makes it better for applications where quality is more important than quantity, though it may not fully leverage the binary nature of ORB descriptors.

```
[ ]: !jupyter nbconvert --to pdf "/content/drive/MyDrive/Colab Notebooks/Exercise 1␣
     ↪- Image Processing Techniques.ipynb"
```