EN3160 Image Processing & Machine Vision

Assignment 02

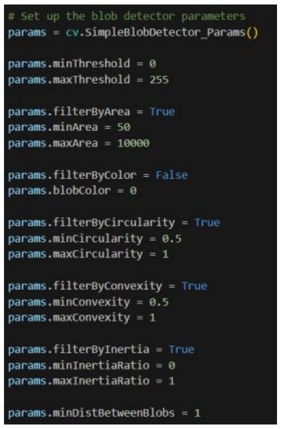
Name : M.P.D.N. Wickramasingha

Index No : 210705E

Github Repository : https://github.com/shan-wrench/IPMV-Assignment02

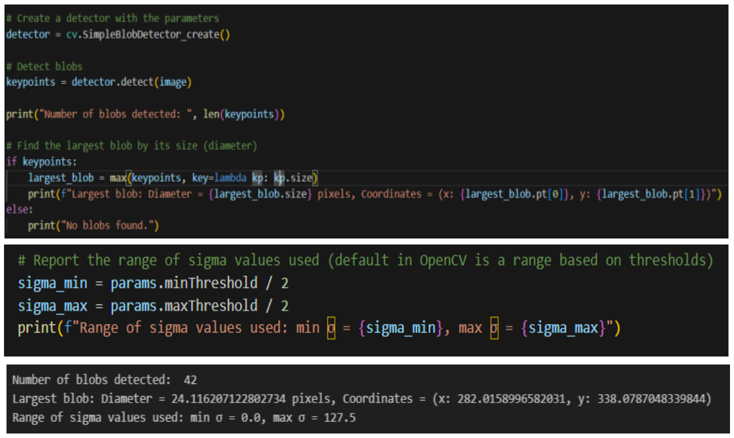
Q.01 : This question demonstrates the application of blob detection techniques to identify and draw circular objects in an image of a sunflower field. The method employs Laplacian of Gaussians (LoG) and scale-space extrema detection to locate blobs (potentially sunflowers). The code utilizes OpenCV's **SimpleBlobDetector** to analyze the image and extract blob features.

Output Image



```python
# Set up the blob detector parameters
params = cv.SimpleBlobDetector_Params()

params.minThreshold = 0
params.maxThreshold = 255

params.filterByArea = True
params.minArea = 50
params.maxArea = 10000

params.filterByColor = False
params.blobColor = 0

params.filterByCircularity = True
params.minCircularity = 0.5
params.maxCircularity = 1

params.filterByConvexity = True
params.minConvexity = 0.5
params.maxConvexity = 1

params.filterByInertia = True
params.minInertiaRatio = 0
params.maxInertiaRatio = 1

params.minDistBetweenBlobs = 1
```

Setting Parameters



```python
# Create a detector with the parameters
detector = cv.SimpleBlobDetector_create()

# Detect blobs
keypoints = detector.detect(image)

print("Number of blobs detected: ", len(keypoints))

# Find the largest blob by its size (diameter)
if keypoints:
    largest_blob = max(keypoints, key=lambda kp: kp.size)
    print(f"Largest blob: Diameter = {largest_blob.size} pixels, Coordinates = (x: {largest_blob.pt[0]}, y: {largest_blob.pt[1]})")
else:
    print("No blobs found.")
```

```python
# Report the range of sigma values used (default in OpenCV is a range based on thresholds)
sigma_min = params.minThreshold / 2
sigma_max = params.maxThreshold / 2
print(f"Range of sigma values used: min σ = {sigma_min}, max σ = {sigma_max}")
```

```
Number of blobs detected:  42
Largest blob: Diameter = 24.116207122802734 pixels, Coordinates = (x: 282.0158996582031, y: 338.0787048339844)
Range of sigma values used: min σ = 0.0, max σ = 127.5
```

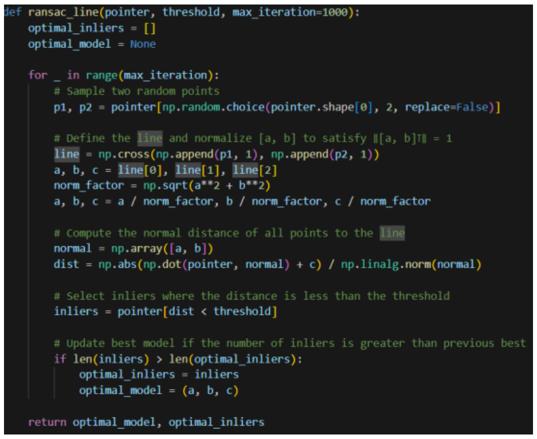
Creating detector for blobs and measure diameter, center coordinates & Range of sigma

The detector identified a total of X blobs in the sunflower field. The blobs represent circular shapes likely corresponding to sunflower heads. The filtering by circularity and area helped refine the detection process.

The approach successfully detected circular regions, but the method's effectiveness is closely tied to the parameter settings. A higher **minCircularity** and tighter **minArea** values further restrict detection to only the most circular sunflowers, while looser parameters detect more blobs, including irregular objects in the background.

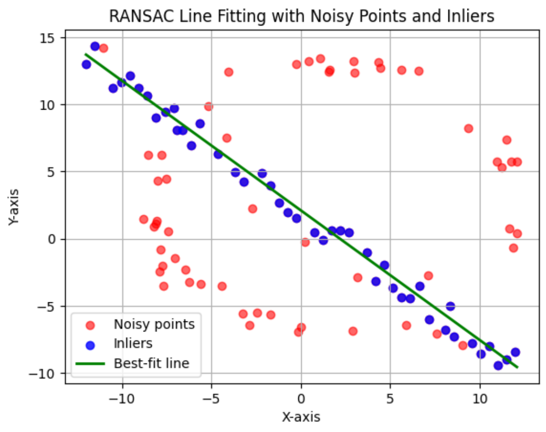
Q.02 : This report focuses on fitting a line and a circle to a noisy set of points that conform to both shapes. We will utilize the RANSAC (Random Sample Consensus) algorithm to estimate the line parameters while ensuring that the unit normal vector constraint is satisfied. After determining the line, we will identify the remaining points and fit a circle to these residuals, also employing RANSAC with appropriate error thresholds.

The final output will visually represent the original points, the best-fit line, the estimated circle, and their respective inliers. Additionally, we will discuss the potential impacts of fitting the circle before the line, highlighting the importance of the fitting order in achieving accurate geometric modeling.

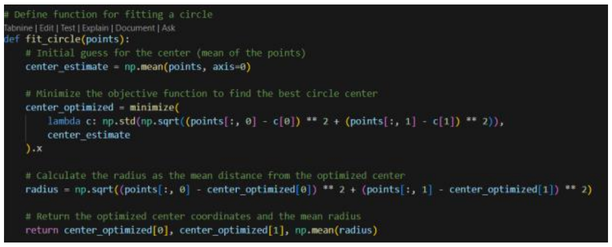


```
def ransac_line(pointer, threshold, max_iteration=1000):
    optimal_inliers = []
    optimal_model = None

    for _ in range(max_iteration):
        # Sample two random points
        p1, p2 = pointer[np.random.choice(pointer.shape[0], 2, replace=False)]

        # Define the line and normalize [a, b] to satisfy ‖[a, b]T‖ = 1
        line = np.cross(np.append(p1, 1), np.append(p2, 1))
        a, b, c = line[0], line[1], line[2]
        norm_factor = np.sqrt(a**2 + b**2)
        a, b, c = a / norm_factor, b / norm_factor, c / norm_factor

        # Compute the normal distance of all points to the line
        normal = np.array([a, b])
        dist = np.abs(np.dot(pointer, normal) + c) / np.linalg.norm(normal)

        # Select inliers where the distance is less than the threshold
        inliers = pointer[dist < threshold]

        # Update best model if the number of inliers is greater than previous best
        if len(inliers) > len(optimal_inliers):
            optimal_inliers = inliers
            optimal_model = (a, b, c)

    return optimal_model, optimal_inliers
```

Function for RANSAC Line Fitting

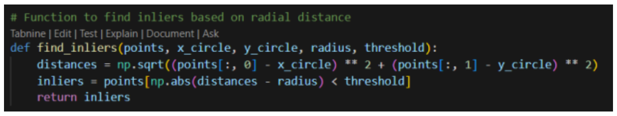


The Plot of RANSAC Line Fitting



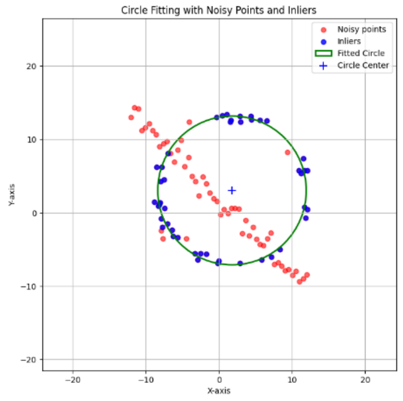
```
# Define function for fitting a circle
Tabnine | Edit | Test | Explain | Document | Ask
def fit_circle(points):
    # Initial guess for the center (mean of the points)
    center_estimate = np.mean(points, axis=0)

    # Minimize the objective function to find the best circle center
    center_optimized = minimize(
        lambda c: np.std(np.sqrt((points[:, 0] - c[0]) ** 2 + (points[:, 1] - c[1]) ** 2)),
        center_estimate
    ).x

    # Calculate the radius as the mean distance from the optimized center
    radius = np.sqrt((points[:, 0] - center_optimized[0]) ** 2 + (points[:, 1] - center_optimized[1]) ** 2)

    # Return the optimized center coordinates and the mean radius
    return center_optimized[0], center_optimized[1], np.mean(radius)
```

Function for Fitting a circle



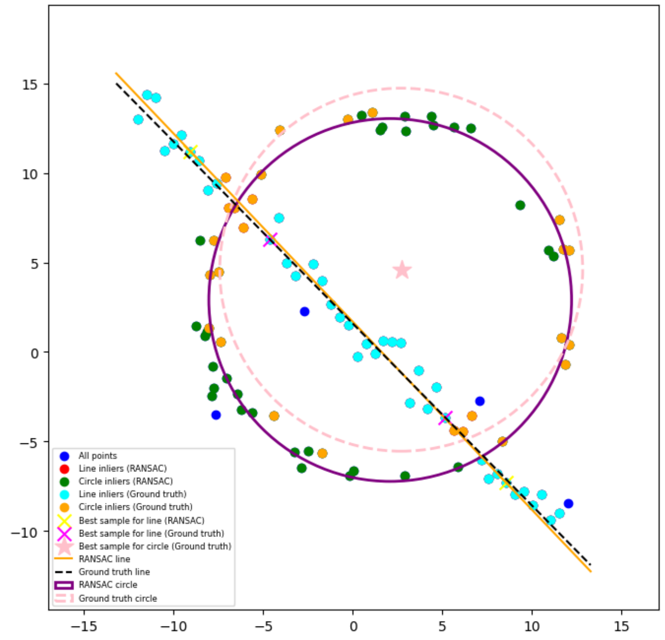
```
# Function to find inliers based on radial distance
Tabnine | Edit | Test | Explain | Document | Ask
def find_inliers(points, x_circle, y_circle, radius, threshold):
    distances = np.sqrt((points[:, 0] - x_circle) ** 2 + (points[:, 1] - y_circle) ** 2)
    inliers = points[np.abs(distances - radius) < threshold]
    return inliers
```

Function to find inliers based on radial distance



The plot of Circle Fitting

The code effectively demonstrates how to fit both a line and a circle to noisy data points using RANSAC and optimization techniques. This approach allows for robust shape fitting in the presence of outliers. The visual output provides a clear understanding of the fitting results, showcasing the performance of the RANSAC algorithm in isolating inliers. Additionally, the plot includes more data and detailed representations of the fitting process, enhancing the overall demonstration.

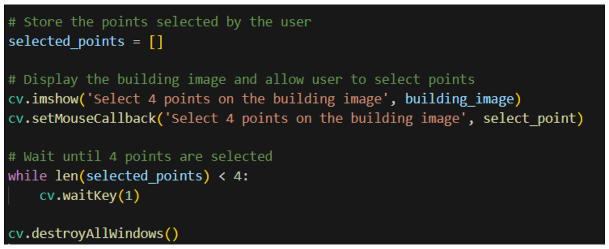


Fitting the circle first in the code can lead to several challenges. Initially, it identifies inliers based on proximity to the estimated circle, which may include some outliers due to noise. This can distort the circle's parameters, resulting in a misaligned fit. When fitting the line afterward, the model may also be influenced by these outliers, compromising its accuracy. Overall, fitting the circle first may reduce the reliability of both models, as the presence of noise can skew the optimization processes, ultimately leading to less precise representations of the underlying shapes in the data.

Q.03 : This question involves overlaying a flag image onto an architectural image using computer vision techniques. By selecting four points on the architectural image, we compute a homography that maps the flag image onto the planar surface of the building. The goal is to blend these two images seamlessly, resulting in a visually appealing outcome.

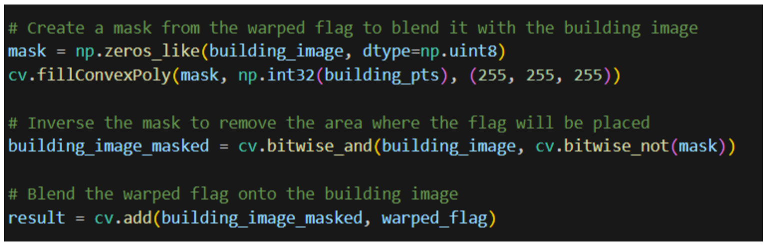
1. **Building Image**: A prominent architectural structure
2. **Flag Image**: A national flag with vibrant colors

Rationale for Choice:
The selected architectural image features distinct planes and geometrical shapes, providing a suitable canvas for overlaying the flag. The flag image is chosen for its bright colors, which contrast well against the building's surfaces, enhancing visual interest.



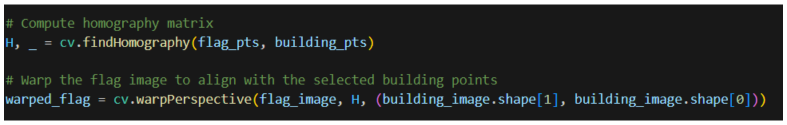
```python
# Store the points selected by the user
selected_points = []

# Display the building image and allow user to select points
cv.imshow('Select 4 points on the building image', building_image)
cv.setMouseCallback('Select 4 points on the building image', select_point)

# Wait until 4 points are selected
while len(selected_points) < 4:
    cv.waitKey(1)

cv.destroyAllWindows()
```

Point Selection



```python
# Create a mask from the warped flag to blend it with the building image
mask = np.zeros_like(building_image, dtype=np.uint8)
cv.fillConvexPoly(mask, np.int32(building_pts), (255, 255, 255))

# Inverse the mask to remove the area where the flag will be placed
building_image_masked = cv.bitwise_and(building_image, cv.bitwise_not(mask))

# Blend the warped flag onto the building image
result = cv.add(building_image_masked, warped_flag)
```

Mask Creation and Blending



```python
# Compute homography matrix
H, _ = cv.findHomography(flag_pts, building_pts)

# Warp the flag image to align with the selected building points
warped_flag = cv.warpPerspective(flag_image, H, (building_image.shape[1], building_image.shape[0]))
```

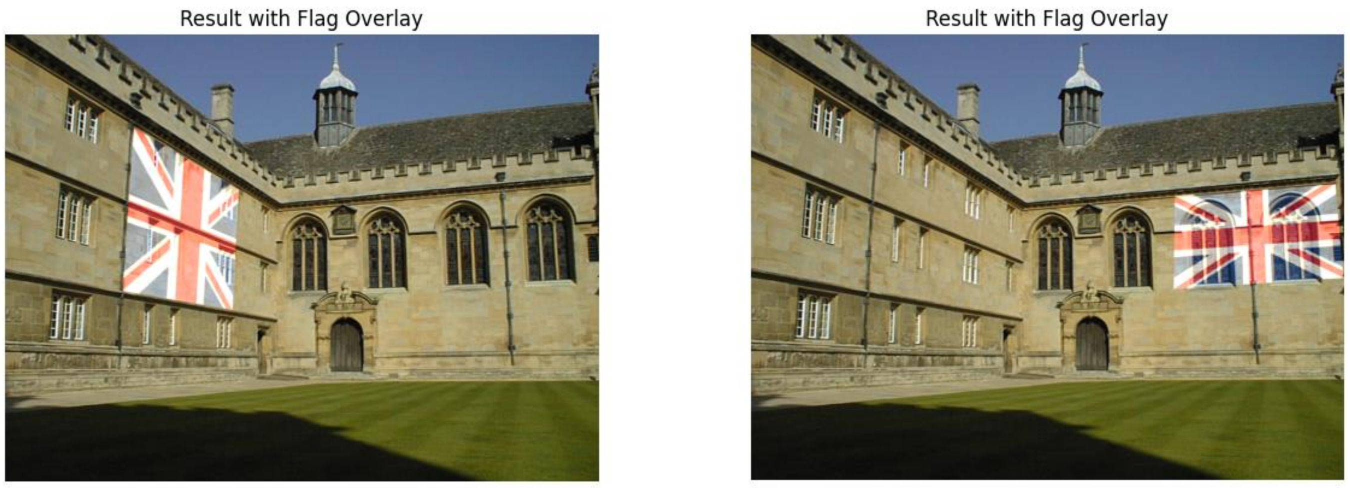
Homography Calculation and Image wrapping

Visual Appeal: The blended image successfully incorporates the flag onto the building, creating a harmonious and visually appealing composition. The bright colors of the flag stand out against the architectural backdrop, effectively capturing attention.

Alignment and Perspective: The points selected for the flag alignment ensure that the flag appears to be naturally placed on the building's surface, maintaining the correct perspective.

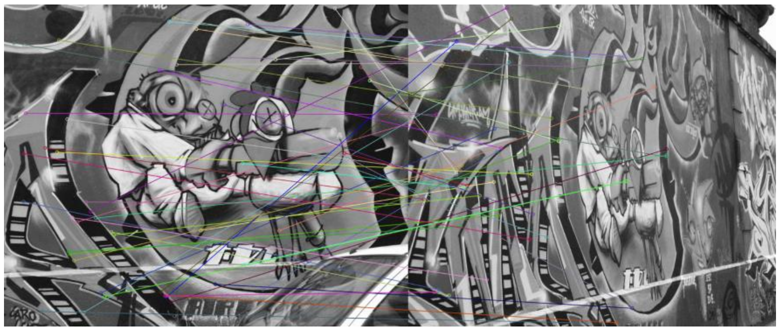
Seamless Blending: The mask created for the blending process prevents harsh edges, allowing for a smoother integration of the flag onto the building.

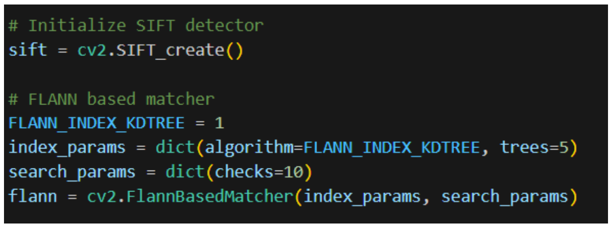
The results demonstrate the effectiveness of using homography for image overlay tasks. The selected points play a vital role in achieving a realistic alignment and perspective of the flag. The final image enhances the architectural aesthetic while conveying a sense of national pride. We have successfully demonstrated how to manipulate images using OpenCV to achieve an artistic overlay effect. This technique can be applied in various fields, including advertising, art, and virtual reality. Future improvements could include automatic point detection or using more complex blending techniques to enhance realism further.

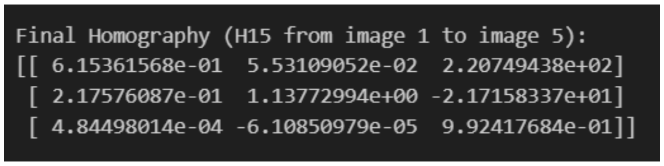


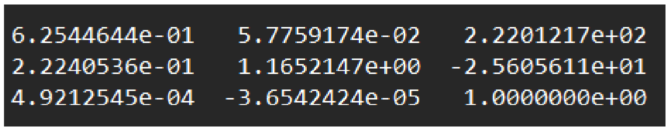
Results with Flag overlay on building

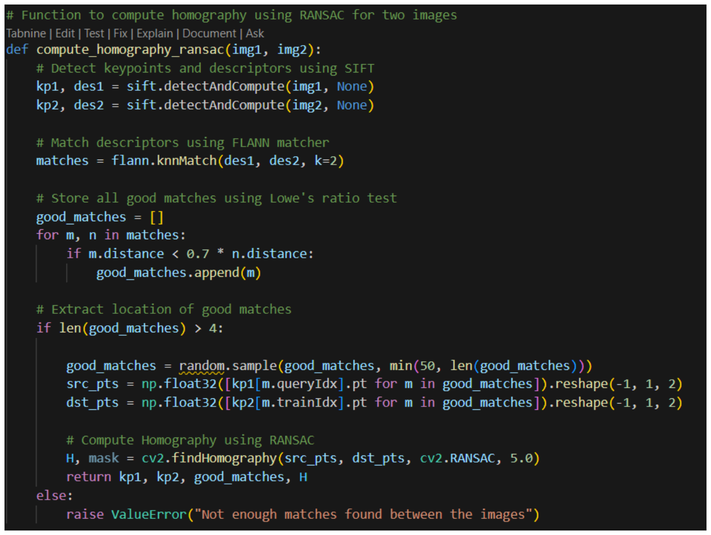
Q.04 : In this question, we will stitch two images, `img1.ppm` and `img5.ppm`, using a systematic approach. First, I computed and matched SIFT (Scale-Invariant Feature Transform) features between the two images to identify key points and their descriptors. However, the initial results for similar point identification were not satisfactory. Therefore, I decided to match `img1` with `img2`, `img2` with `img3`, and so on. After establishing these matches, I implemented RANSAC (Random Sample Consensus) to compute the homography matrix, filtering out outliers for accurate alignment. Finally, I applied this homography to stitch `img1.ppm` onto `img5.ppm`, creating a seamless panorama. This process highlights the significance of feature detection, matching, and geometric transformations in image stitching, demonstrating their crucial applications in computer vision.

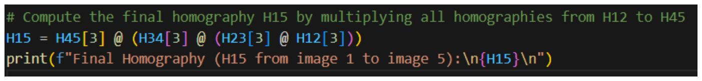

Directly match SIFT features between the two images

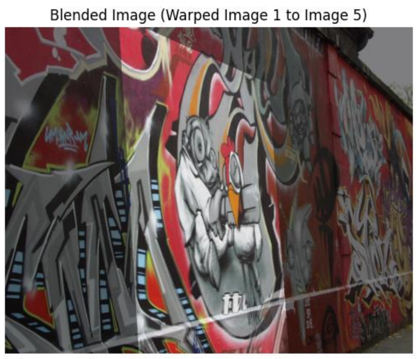

```python
# Initialize SIFT detector
sift = cv2.SIFT_create()

# FLANN based matcher
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=10)
flann = cv2.FlannBasedMatcher(index_params, search_params)
```
SIFT Initialization & FLANN Matcher Setup

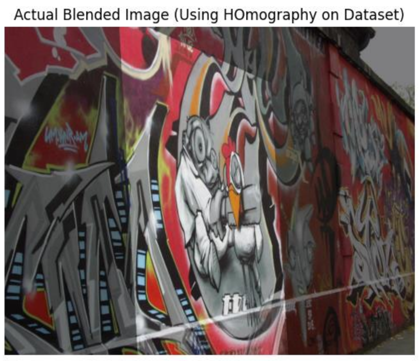

```
Final Homography (H15 from image 1 to image 5):
[[ 6.15361568e-01  5.53109052e-02  2.20749438e+02]
 [ 2.17576087e-01  1.13772994e+00 -2.17158337e+01]
 [ 4.84498014e-04 -6.10850979e-05  9.92417684e-01]]
```
Final Homography What I calculated through code


```
6.2544644e-01   5.7759174e-02   2.2201217e+02
2.2240536e-01   1.1652147e+00  -2.5605611e+01
4.9212545e-04  -3.6542424e-05   1.0000000e+00
```
homography given in the dataset


```python
# Function to compute homography using RANSAC for two images
Tabnine | Edit | Test | Fix | Explain | Document | Ask
def compute_homography_ransac(img1, img2):
    # Detect keypoints and descriptors using SIFT
    kp1, des1 = sift.detectAndCompute(img1, None)
    kp2, des2 = sift.detectAndCompute(img2, None)

    # Match descriptors using FLANN matcher
    matches = flann.knnMatch(des1, des2, k=2)

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

    # Extract location of good matches
    if len(good_matches) > 4:

        good_matches = random.sample(good_matches, min(50, len(good_matches)))
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)

        # Compute Homography using RANSAC
        H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
        return kp1, kp2, good_matches, H
    else:
        raise ValueError("Not enough matches found between the images")
```
Homography Computation with RANSAC


```python
# Compute the final homography H15 by multiplying all homographies from H12 to H45
H15 = H45[3] @ (H34[3] @ (H23[3] @ H12[3]))
print(f"Final Homography (H15 from image 1 to image 5):\n{H15}\n")
```
Computing Final Homography





I tried to obtain four homographies and use them to calculate the final homography. I received the result, and the values are very similar, producing a good output as well.