CSCI 677 –Advance Computer Vision- Fall 2024 - HW 2

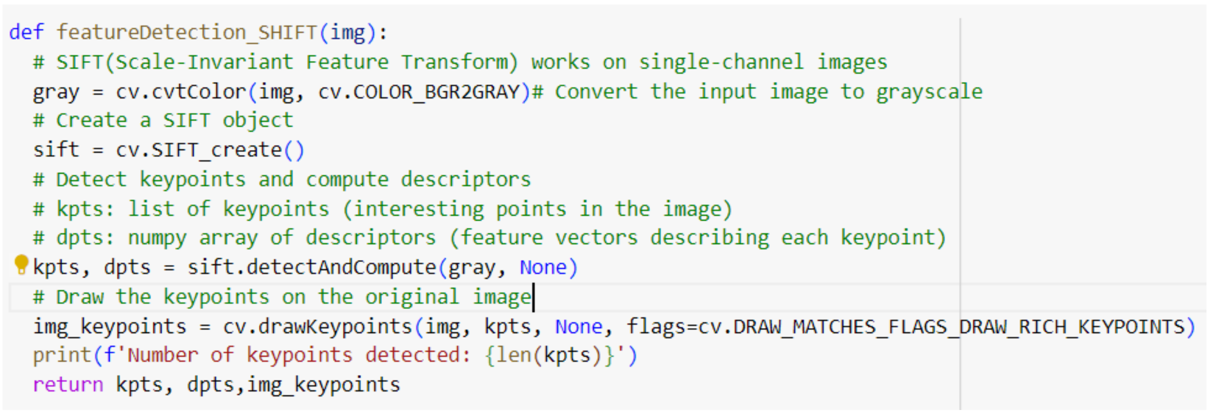
USC ID: 1154164561
NAME: Srija Madarapu
EMAIL: madarapu@usc.edu

Code:

https://colab.research.google.com/drive/1m4LLTRTDQ5SYmUlBI5WOS-33BaXOKtrw?usp=sharing

1.2 Feature Detection

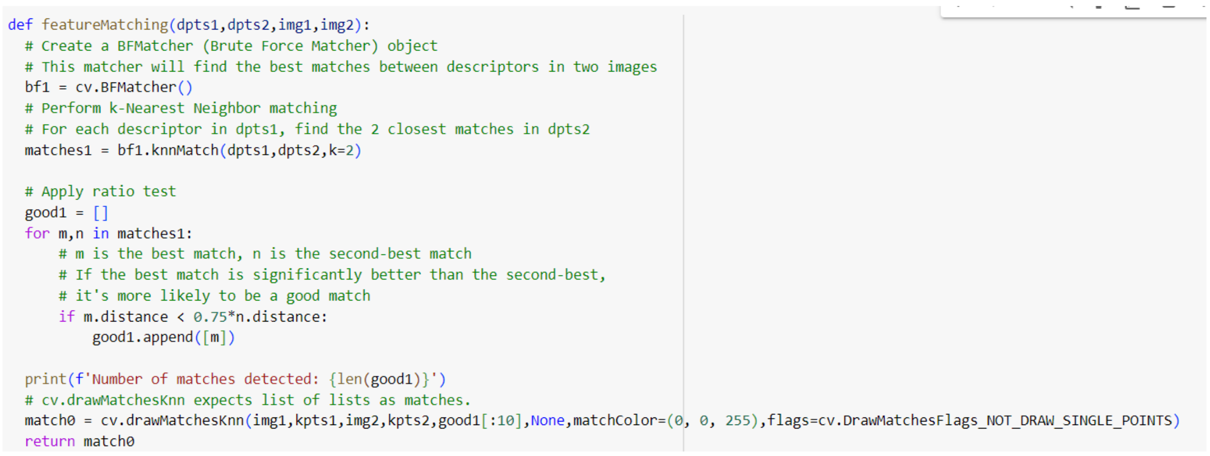
- The featureDetection_SHIFT function uses SIFT (Scale-Invariant Feature Transform) to detect keypoints and compute descriptors for each image.
- It visualizes the keypoints on each image.



```python
def featureDetection_SHIFT(img):
  # SIFT(Scale-Invariant Feature Transform) works on single-channel images
  gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)# Convert the input image to grayscale
  # Create a SIFT object
  sift = cv.SIFT_create()
  # Detect keypoints and compute descriptors
  # kpts: list of keypoints (interesting points in the image)
  # dpts: numpy array of descriptors (feature vectors describing each keypoint)
  kpts, dpts = sift.detectAndCompute(gray, None)
  # Draw the keypoints on the original image
  img_keypoints = cv.drawKeypoints(img, kpts, None, flags=cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
  print(f'Number of keypoints detected: {len(kpts)}')
  return kpts, dpts,img_keypoints
```

1.3 Feature Matching

- The featureMatching function uses a Brute Force Matcher to find matches between keypoints in pairs of images.
- It applies Lowe's ratio test to filter good matches.
- It visualizes the top 10 matches between image pairs.

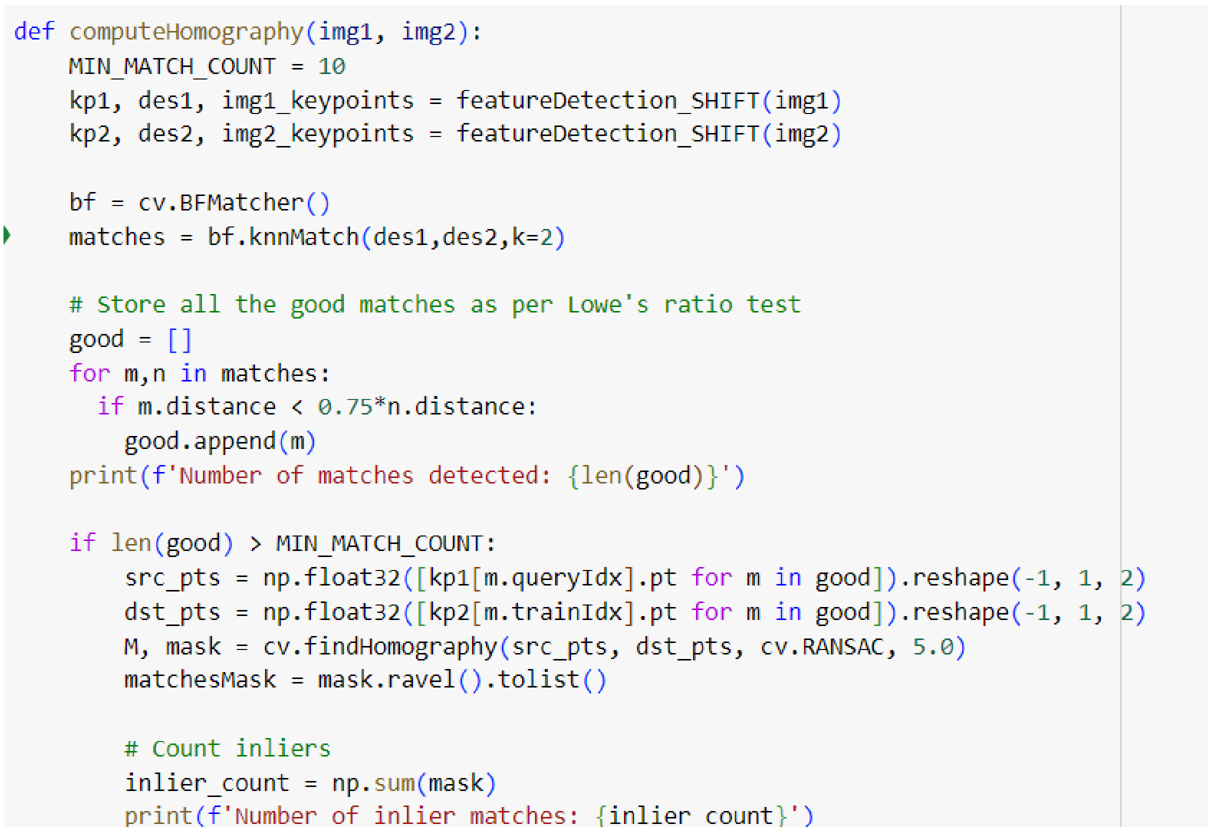


```
def featureMatching(dpts1,dpts2,img1,img2):
  # Create a BFMatcher (Brute Force Matcher) object
  # This matcher will find the best matches between descriptors in two images
  bf1 = cv.BFMatcher()
  # Perform k-Nearest Neighbor matching
  # For each descriptor in dpts1, find the 2 closest matches in dpts2
  matches1 = bf1.knnMatch(dpts1,dpts2,k=2)

  # Apply ratio test
  good1 = []
  for m,n in matches1:
      # m is the best match, n is the second-best match
      # If the best match is significantly better than the second-best,
      # it's more likely to be a good match
      if m.distance < 0.75*n.distance:
          good1.append([m])

  print(f'Number of matches detected: {len(good1)}')
  # cv.drawMatchesKnn expects list of lists as matches.
  match0 = cv.drawMatchesKnn(img1,kpts1,img2,kpts2,good1[:10],None,matchColor=(0, 0, 255),flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
  return match0
```

1.4 Compute Homography

- The computeHomography function computes the homography matrix between two images using RANSAC.
- It identifies inlier matches and calculates the error for each match.
- It visualizes all inlier matches and the top 10 matches with minimum error.

Using Knn:

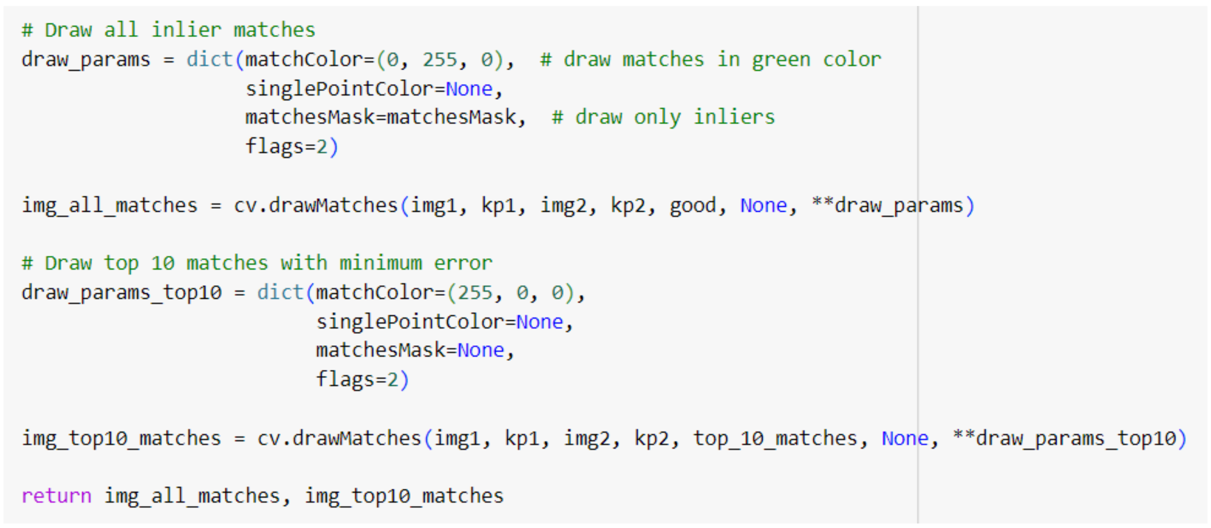


```
def computeHomography(img1, img2):
    MIN_MATCH_COUNT = 10
    kp1, des1, img1_keypoints = featureDetection_SHIFT(img1)
    kp2, des2, img2_keypoints = featureDetection_SHIFT(img2)

    bf = cv.BFMatcher()
    matches = bf.knnMatch(des1,des2,k=2)

    # Store all the good matches as per Lowe's ratio test
    good = []
    for m,n in matches:
      if m.distance < 0.75*n.distance:
        good.append(m)
    print(f'Number of matches detected: {len(good)}')

    if len(good) > MIN_MATCH_COUNT:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)
        M, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC, 5.0)
        matchesMask = mask.ravel().tolist()

        # Count inliers
        inlier_count = np.sum(mask)
        print(f'Number of inlier matches: {inlier_count}')
```

Using flann:

```python
def computeHomography(img1, img2):
    MIN_MATCH_COUNT = 10
    kp1, des1, img1_keypoints = featureDetection_SHIFT(img1)
    kp2, des2, img2_keypoints = featureDetection_SHIFT(img2)

    # FLANN parameters for fast nearest neighbor matching
    FLANN_INDEX_KDTREE = 1
    index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
    search_params = dict(checks=50)
    flann = cv.FlannBasedMatcher(index_params, search_params)

    # Perform knn matching
    matches = flann.knnMatch(des1, des2, k=2)

    # Store all the good matches as per Lowe's ratio test
    good = []
    for m in matches:
        # Check if we have two matches
        if len(m) == 2:
            n = m[1]  # Get the second match
            if m[0].distance < 0.7 * n.distance:
                good.append(m[0])

    if len(good) > MIN_MATCH_COUNT:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)
        M, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC, 5.0)
        matchesMask = mask.ravel().tolist()
```

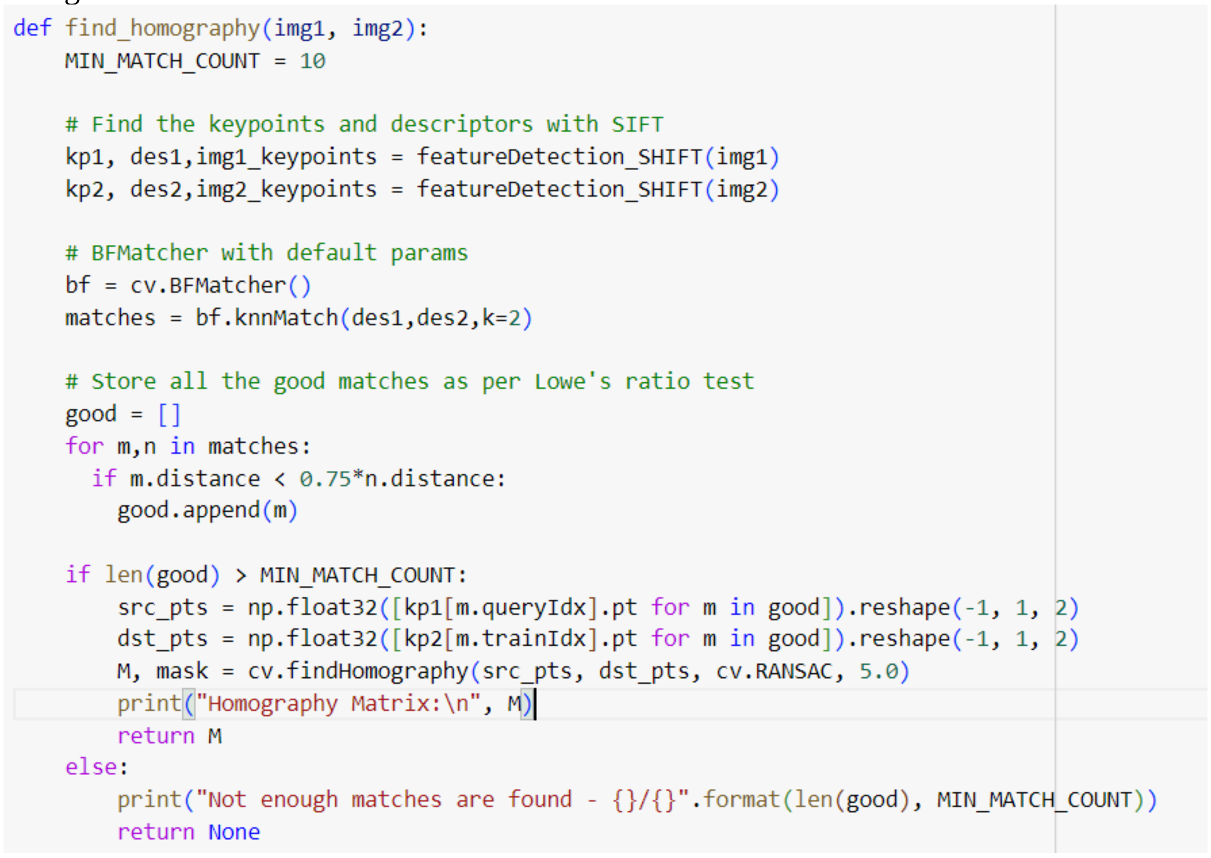
```python
        # Count inliers
        inlier_count = np.sum(mask)
        print(f'Number of inlier matches: {inlier_count}')

        # Calculate error for each match
        errors = []
        for i, (src_pt, dst_pt) in enumerate(zip(src_pts, dst_pts)):
            if matchesMask[i]:
                projected_pt = cv.perspectiveTransform(src_pt.reshape(-1, 1, 2), M)
                error = np.linalg.norm(projected_pt - dst_pt)
                errors.append((error, good[i]))

        # Sort matches by error
        errors.sort(key=lambda x: x[0])

        # Get top 10 matches with minimum error
        top_10_matches = [match for _, match in errors[:10]]

        h, w = img1.shape[:2]
        pts = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1, 1, 2)
        dst = cv.perspectiveTransform(pts, M)
        img2 = cv.polylines(img2, [np.int32(dst)], True, 255, 3, cv.LINE_AA)
    else:
        print("Not enough matches are found - {}/{}".format(len(good), MIN_MATCH_COUNT))
        matchesMask = None
        top_10_matches = []
```


```
# Draw all inlier matches
draw_params = dict(matchColor=(0, 255, 0),  # draw matches in green color
                   singlePointColor=None,
                   matchesMask=matchesMask,  # draw only inliers
                   flags=2)

img_all_matches = cv.drawMatches(img1, kp1, img2, kp2, good, None, **draw_params)

# Draw top 10 matches with minimum error
draw_params_top10 = dict(matchColor=(255, 0, 0),
                         singlePointColor=None,
                         matchesMask=None,
                         flags=2)

img_top10_matches = cv.drawMatches(img1, kp1, img2, kp2, top_10_matches, None, **draw_params_top10)

return img_all_matches, img_top10_matches
```

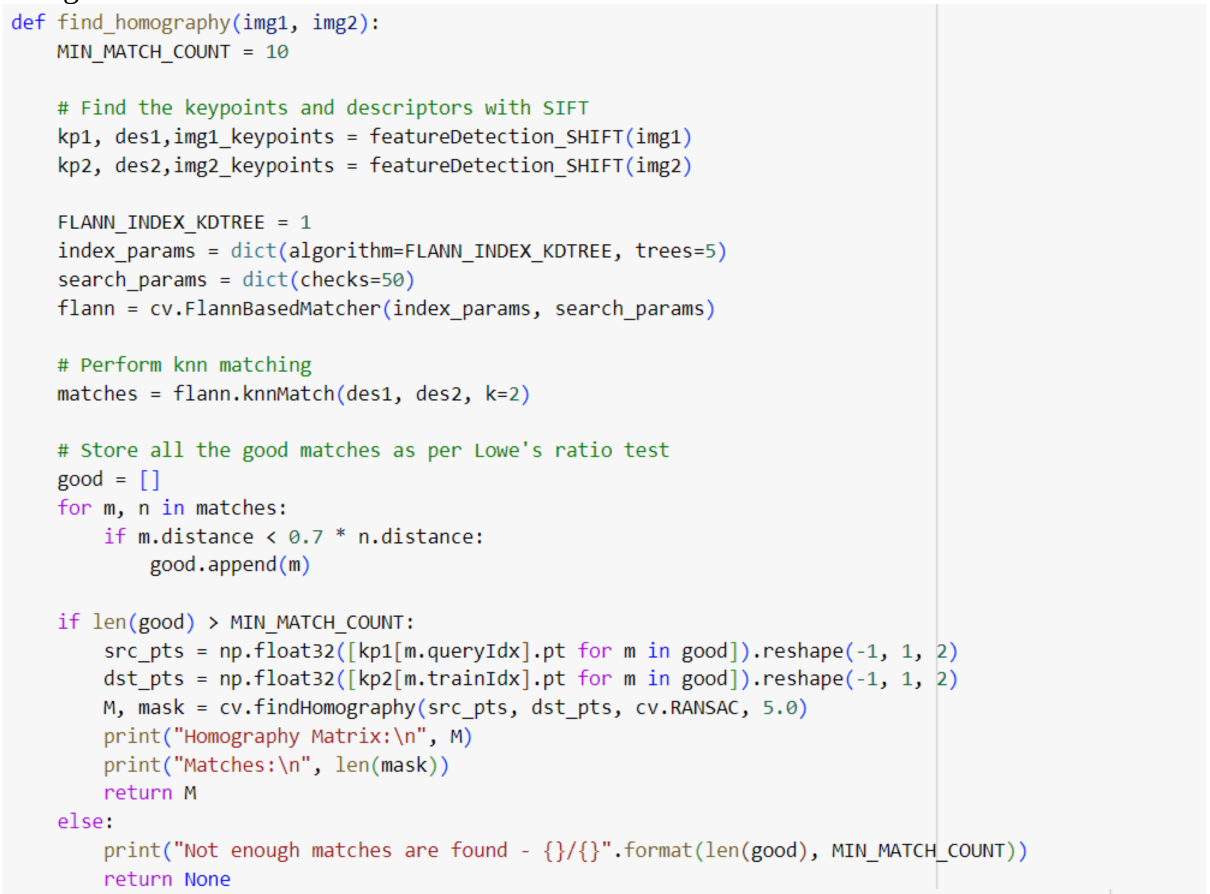
1.5 Stich into a Panorama

- The stitch_images function uses the computed homographies to warp and blend the images into a panorama.
- It calculates the size of the final panorama and applies appropriate transformations to each image.

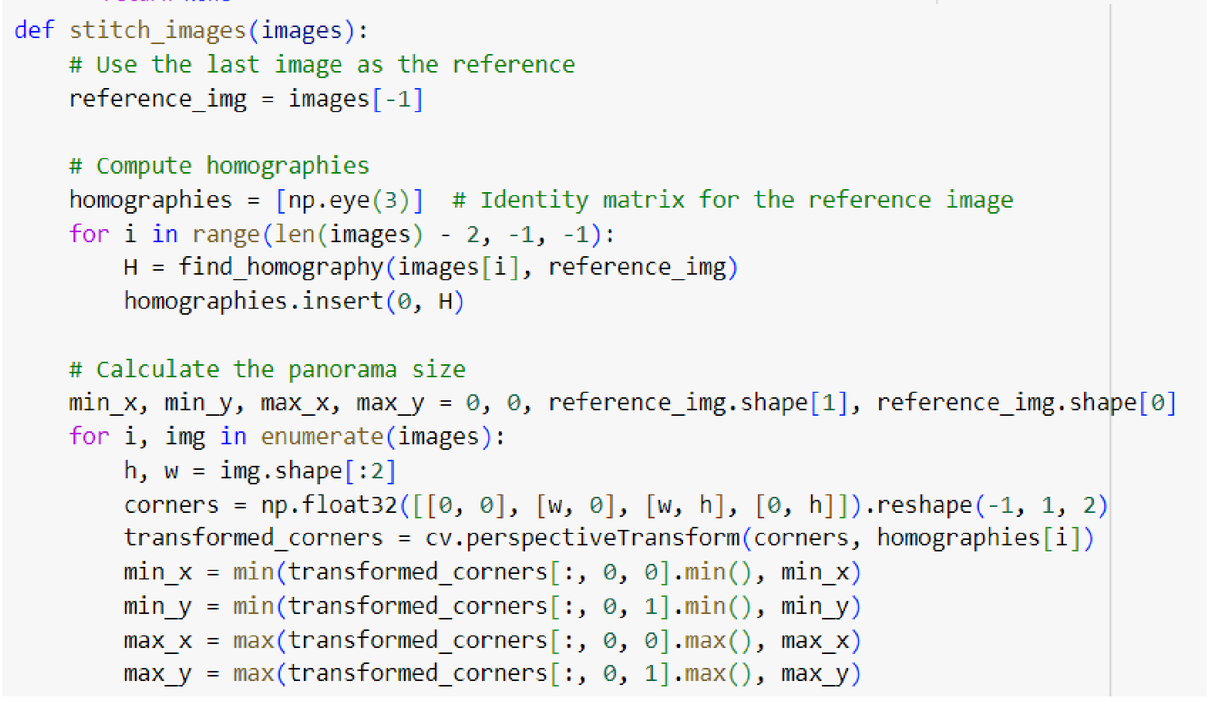
Using Knn:


```
def find_homography(img1, img2):
    MIN_MATCH_COUNT = 10

    # Find the keypoints and descriptors with SIFT
    kp1, des1,img1_keypoints = featureDetection_SHIFT(img1)
    kp2, des2,img2_keypoints = featureDetection_SHIFT(img2)

    # BFMatcher with default params
    bf = cv.BFMatcher()
    matches = bf.knnMatch(des1,des2,k=2)

    # Store all the good matches as per Lowe's ratio test
    good = []
    for m,n in matches:
      if m.distance < 0.75*n.distance:
        good.append(m)

    if len(good) > MIN_MATCH_COUNT:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)
        M, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC, 5.0)
        print("Homography Matrix:\n", M)
        return M
    else:
        print("Not enough matches are found - {}/{}".format(len(good), MIN_MATCH_COUNT))
        return None
```

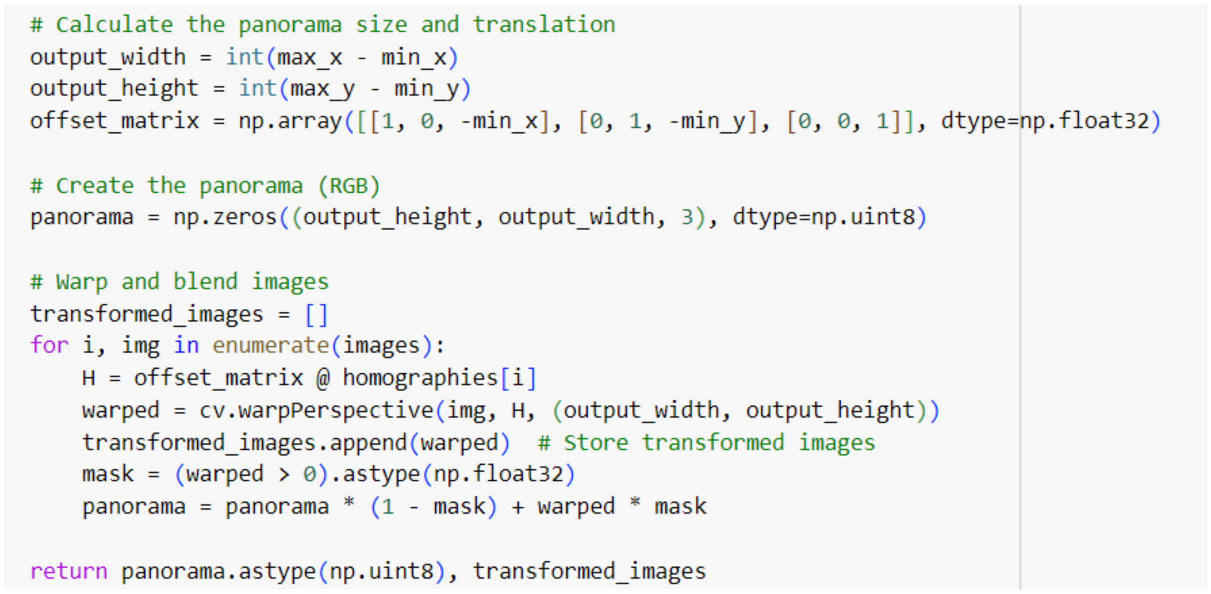
Using FLANN:



```python
def find_homography(img1, img2):
    MIN_MATCH_COUNT = 10

    # Find the keypoints and descriptors with SIFT
    kp1, des1,img1_keypoints = featureDetection_SHIFT(img1)
    kp2, des2,img2_keypoints = featureDetection_SHIFT(img2)

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

    # Perform knn matching
    matches = flann.knnMatch(des1, des2, k=2)

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

    if len(good) > MIN_MATCH_COUNT:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)
        M, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC, 5.0)
        print("Homography Matrix:\n", M)
        print("Matches:\n", len(mask))
        return M
    else:
        print("Not enough matches are found - {}/{}".format(len(good), MIN_MATCH_COUNT))
        return None
```



```python
def stitch_images(images):
    # Use the last image as the reference
    reference_img = images[-1]

    # Compute homographies
    homographies = [np.eye(3)]  # Identity matrix for the reference image
    for i in range(len(images) - 2, -1, -1):
        H = find_homography(images[i], reference_img)
        homographies.insert(0, H)

    # Calculate the panorama size
    min_x, min_y, max_x, max_y = 0, 0, reference_img.shape[1], reference_img.shape[0]
    for i, img in enumerate(images):
        h, w = img.shape[:2]
        corners = np.float32([[0, 0], [w, 0], [w, h], [0, h]]).reshape(-1, 1, 2)
        transformed_corners = cv.perspectiveTransform(corners, homographies[i])
        min_x = min(transformed_corners[:, 0, 0].min(), min_x)
        min_y = min(transformed_corners[:, 0, 1].min(), min_y)
        max_x = max(transformed_corners[:, 0, 0].max(), max_x)
        max_y = max(transformed_corners[:, 0, 1].max(), max_y)
```



```
# Calculate the panorama size and translation
output_width = int(max_x - min_x)
output_height = int(max_y - min_y)
offset_matrix = np.array([[1, 0, -min_x], [0, 1, -min_y], [0, 0, 1]], dtype=np.float32)

# Create the panorama (RGB)
panorama = np.zeros((output_height, output_width, 3), dtype=np.uint8)

# Warp and blend images
transformed_images = []
for i, img in enumerate(images):
    H = offset_matrix @ homographies[i]
    warped = cv.warpPerspective(img, H, (output_width, output_height))
    transformed_images.append(warped)  # Store transformed images
    mask = (warped > 0).astype(np.float32)
    panorama = panorama * (1 - mask) + warped * mask

return panorama.astype(np.uint8), transformed_images
```

1. SIFT features: show the detected features overlaid on the images. Also give out the number of features detected in each image.

```
Number of keypoints detected: 75295
Number of keypoints detected: 67155
Number of keypoints detected: 52663
```

```python
img1 = cv.imread('001.jpg')
img2 = cv.imread('002.jpg')
img3 = cv.imread('003.jpg')


kpts1, dpts1,img1_keypoints = featureDetection_SHIFT(img1)
kpts2, dpts2,img2_keypoints = featureDetection_SHIFT(img2)
kpts3, dpts3,img3_keypoints = featureDetection_SHIFT(img3)

plt.figure(figsize=(20, 10))

plt.subplot(2, 2, 1)
plt.imshow(cv.cvtColor(img1_keypoints, cv.COLOR_BGR2RGB))
plt.title('Keypoints in Image 001')
plt.axis('off')

plt.subplot(2, 2, 2)
plt.imshow(cv.cvtColor(img2_keypoints, cv.COLOR_BGR2RGB))
plt.title('Keypoints in Image 002')
plt.axis('off')

plt.subplot(2, 2, 3)
plt.imshow(cv.cvtColor(img3_keypoints, cv.COLOR_BGR2RGB))
plt.title('Keypoints in Image 003')
plt.axis('off')

plt.show()
```

Keypoints in Image 001

Keypoints in Image 002

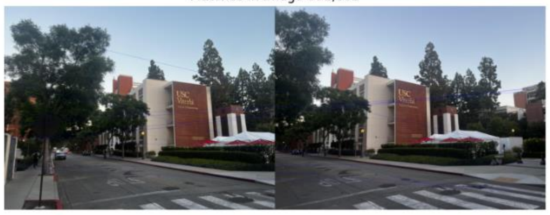
Keypoints in Image 003

2. Graphically show the top-10 scoring matches found by the matcher before the RANSAC operation. Provide statistics of how many matches are found for each image pair.

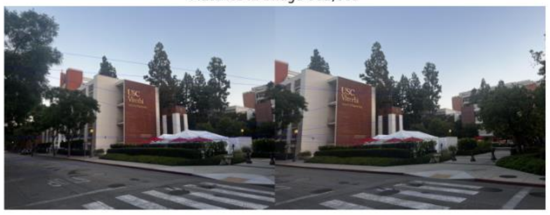
Number of matches detected: 9270
Number of matches detected: 8674

```
match1 = featureMatching(dpts1,dpts2,img1,img2)
match2 = featureMatching(dpts2,dpts3,img2,img3)

plt.figure(figsize=(20, 5))

plt.subplot(1, 2, 1)
plt.imshow(match1)
plt.title('Matches in Image 001,002')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(match2)
plt.title('Matches in Image 002,003')
plt.axis('off')

plt.show()
```

Matches in Image 001,002

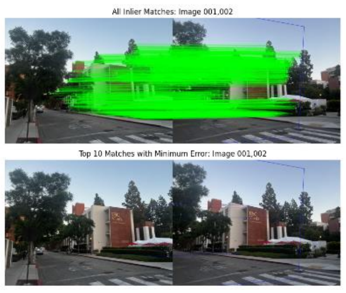
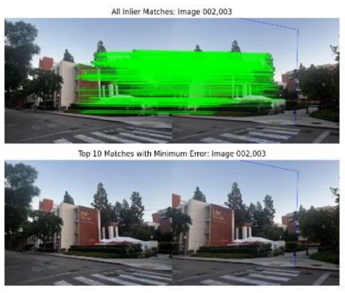


Matches in Image 002,003



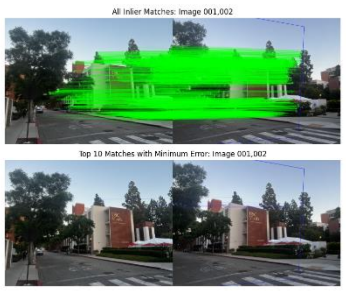
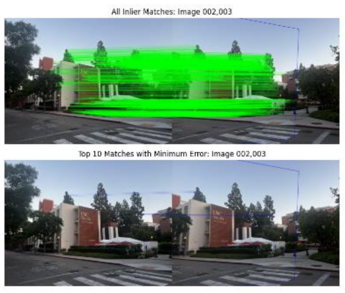
3. Show total number of inlier matches after homography estimations. Also show top-10 matches that have the minimum error between the projected source keypoint and the destination keypoint. (Hint: check the mask value returned by the function estimating the homography).

Using Knn:
Number of inlier matches: 4712
Number of inlier matches: 3532
Number of inlier matches: 2119


All Inlier Matches: Image 001,002
Top 10 Matches with Minimum Error: Image 001,002

All Inlier Matches: Image 002,003
Top 10 Matches with Minimum Error: Image 002,003

All Inlier Matches: Image 001,003
Top 10 Matches with Minimum Error: Image 001,003

Using flann:
Number of inlier matches: 3748
Number of inlier matches: 3091
Number of inlier matches: 1758


All Inlier Matches: Image 001,002
Top 10 Matches with Minimum Error: Image 001,002

All Inlier Matches: Image 002,003
Top 10 Matches with Minimum Error: Image 002,003

All Inlier Matches: Image 002,003
Top 10 Matches with Minimum Error: Image 001,003

4. Output the computed homography matrix. (The built-in homography finder also applies a non-linear optimization step at the end; you can ignore or disable this step if you wish.)

Using Knn:

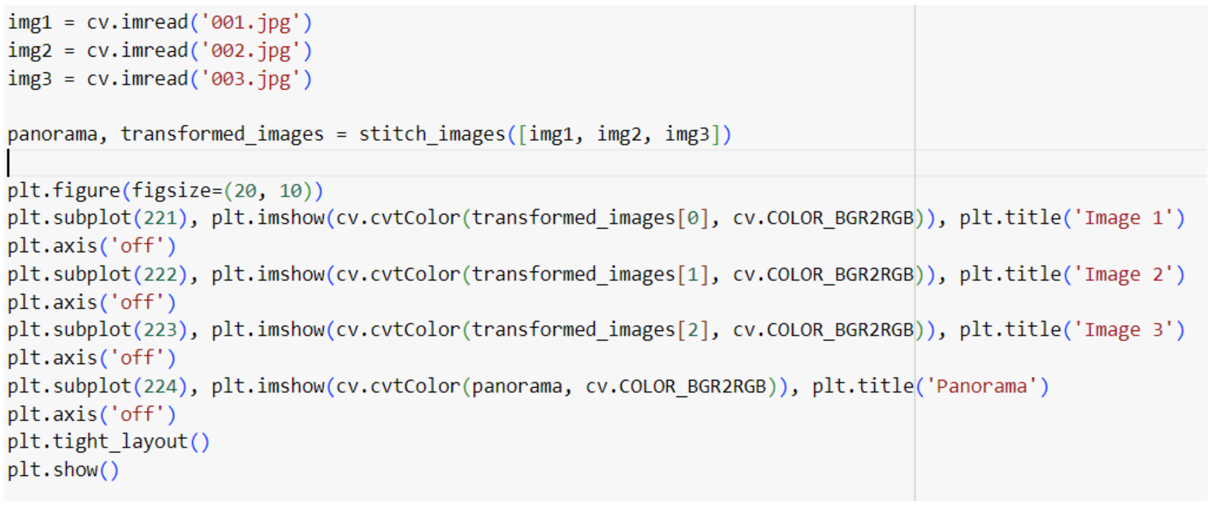
```
Homography Matrix for 1&2:
 [[ 1.47528093e+00 -4.62819220e-02 -1.33342632e+03]
 [ 2.25302869e-01  1.29650792e+00 -5.99816381e+02]
 [ 1.14821958e-04  6.77674899e-06  1.00000000e+00]]
Homography Matrix for 2&3:
 [[ 1.58950028e+00 -6.02160160e-02 -1.60347389e+03]
 [ 2.65764410e-01  1.33575101e+00 -6.25880950e+02]
 [ 1.45378693e-04 -6.71883688e-06  1.00000000e+00]]
Homography Matrix for 1&3:
 [[ 2.38983015e+00 -1.82557054e-01 -4.05953529e+03]
 [ 6.68375413e-01  1.90853545e+00 -1.87729252e+03]
 [ 3.47002693e-04 -2.30011544e-05  1.00000000e+00]]
```

Using Flann:

```
Homography Matrix for 1&2:
 [[ 1.44632304e+00 -4.26601744e-02 -1.29996673e+03]
 [ 2.10437686e-01  1.27540361e+00 -5.52578593e+02]
 [ 1.07920556e-04  7.02940775e-06  1.00000000e+00]]
Homography Matrix for 2&3:
 [[ 1.53945069e+00 -5.94334795e-02 -1.52722178e+03]
 [ 2.46324540e-01  1.32152494e+00 -5.89355959e+02]
 [ 1.32693130e-04 -1.17636599e-06  1.00000000e+00]]
```

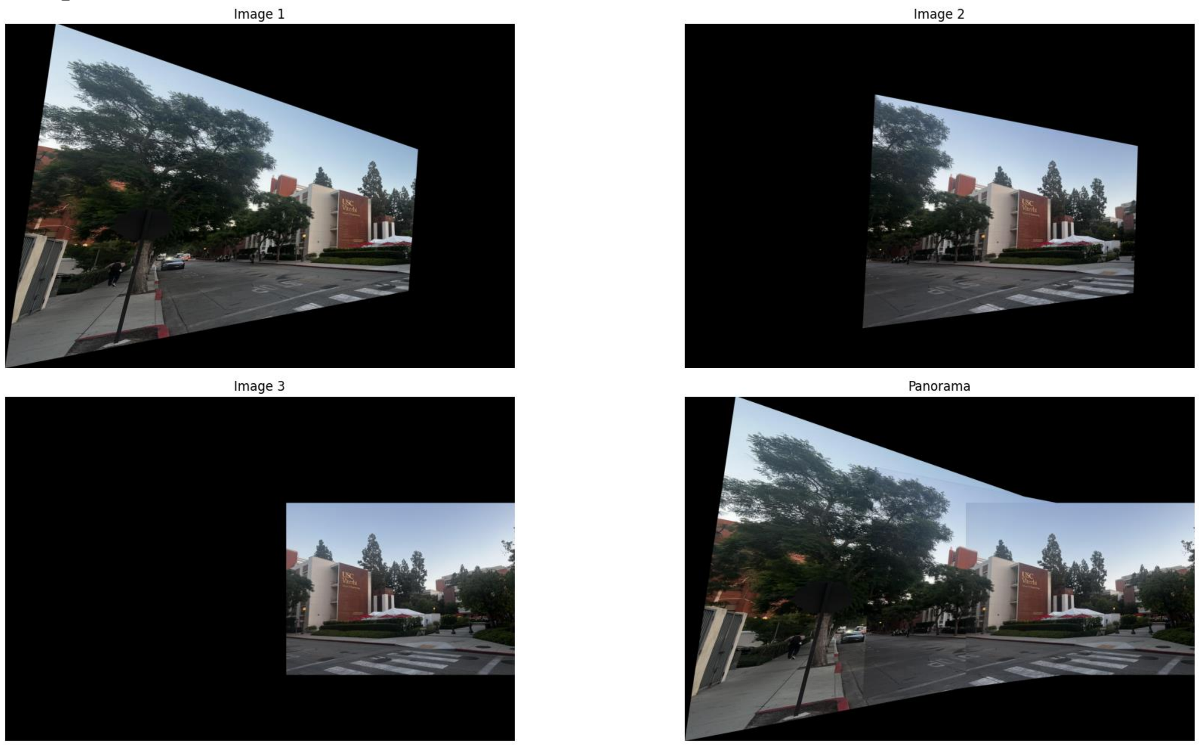
```
Homography Matrix for 1&3:
 [[ 2.23687131e+00 -1.82217764e-01 -3.75718671e+03]
 [ 6.20172566e-01  1.80263699e+00 -1.73672361e+03]
 [ 3.15715227e-04 -3.49153061e-05  1.00000000e+00]]
```

5. Show the final panorama along with each transformed image.

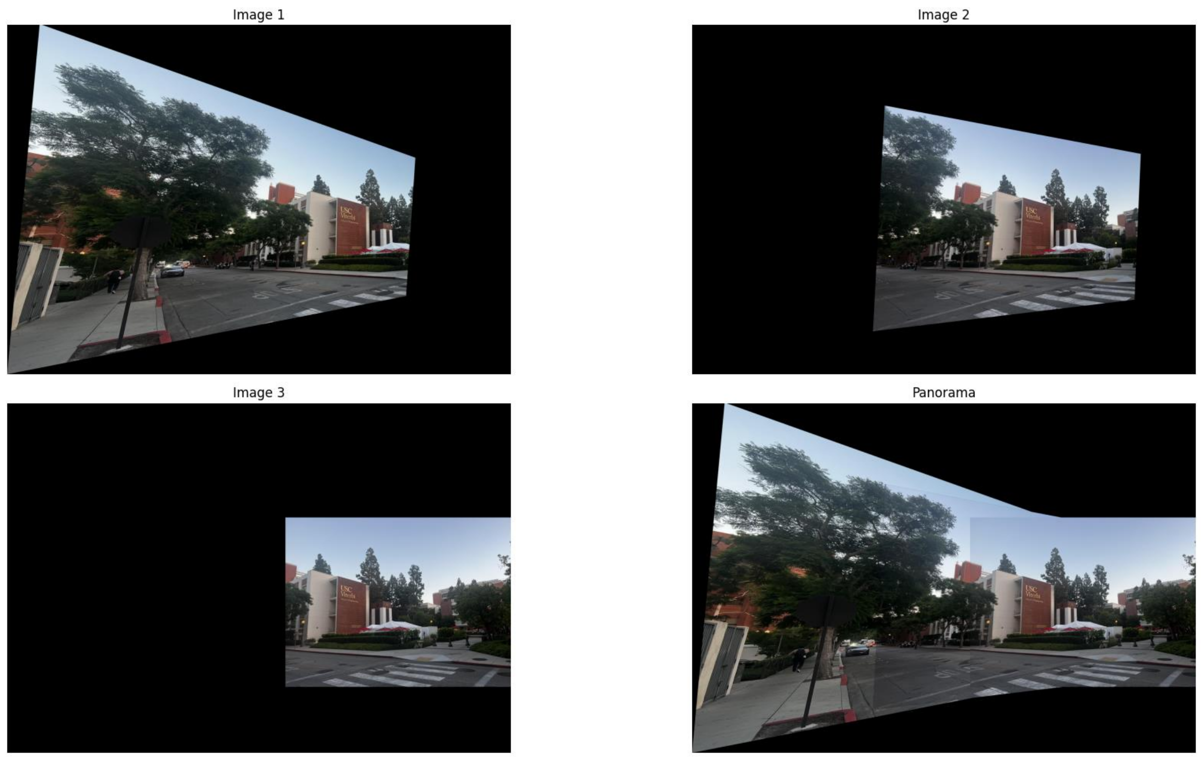


```python
img1 = cv.imread('001.jpg')
img2 = cv.imread('002.jpg')
img3 = cv.imread('003.jpg')

panorama, transformed_images = stitch_images([img1, img2, img3])

plt.figure(figsize=(20, 10))
plt.subplot(221), plt.imshow(cv.cvtColor(transformed_images[0], cv.COLOR_BGR2RGB)), plt.title('Image 1')
plt.axis('off')
plt.subplot(222), plt.imshow(cv.cvtColor(transformed_images[1], cv.COLOR_BGR2RGB)), plt.title('Image 2')
plt.axis('off')
plt.subplot(223), plt.imshow(cv.cvtColor(transformed_images[2], cv.COLOR_BGR2RGB)), plt.title('Image 3')
plt.axis('off')
plt.subplot(224), plt.imshow(cv.cvtColor(panorama, cv.COLOR_BGR2RGB)), plt.title('Panorama')
plt.axis('off')
plt.tight_layout()
plt.show()
```

Using Knn:



Using Flann:



Observations:

Feature Detection: SIFT finds a lot of keypoints in each image, which helps in matching them accurately. These keypoints are spread out nicely, capturing different parts of the scene.

Feature Matching: The matching process successfully links points between pairs of images. Lowe's ratio test effectively removes many incorrect matches, improving the overall match quality.

Homography Computation: The RANSAC algorithm works well to calculate the homography matrices between image pairs, even when some matches are incorrect. The visualization of inlier matches shows good alignment, and the top 10 best matches highlight the most reliable pairs.

Image Stitching: The stitching process effectively aligns and blends the images into a smooth panorama. The transformed images show how each original image fits into the final view. The final panorama offers a wider look at the scene, combining information from all images. This stitched panorama often needs further refinement to become the final product(one similar to camera). Additional processing is necessary to ensure a clean, rectangular image free of visible seams or empty spaces.