Dhruv Maheshwari 9001607595

CSCI 677 – Advanced Computer Vision
Assignment 2

This program is used for stitching images together by exploring multiple images for the detection and correlation of features to form a large single picture. The significant scientific methods are associated with different algorithms such as SIFT for detecting features and random sample consensus estimate (RANSAC) for homography matrices.

Feature Detection with SIFT:
- It is used to detect keypoints and compute descriptors for each image.
- These keypoints are invariant to scale, rotation, and lighting, making them ideal for matching features between overlapping images.

Feature Matching:
- The descriptors are matched between consecutive images using the BFMatcher (Brute Force Matcher) with k-nearest neighbors where k=2.
- A ratio test is applied to filter out poor matches, retaining only good matches.

Homography Estimation with RANSAC:
- RANSAC is used to estimate the homography matrix, which transforms points from one image to the other, using the good matches.
- RANSAC helps reject outliers by iteratively finding the best homography with the most inlier matches, ensuring a robust transformation even in the presence of noise or incorrect matches.

Panorama Creation:
- Using the estimated homographies, the images are warped and blended together to create a seamless panorama.
- The result is a stitched image that combines all input images into a single wide view.

Dhruv Maheshwari 9001607595

SIFT FEATURE DETECTION

Number of features detected in image 0: 75295
Number of features detected in image 1: 67155
Number of features detected in image 2: 52663

Image 1 with keypoints detected

Image 2 with keypoints detected

Image 3 with keypoints detected

SIFT works on finding regions with high contrast features and distinctive patterns. Hence there are lot of features on the building and lesser SIFT features on the sky which appears like a smooth plane.

<u>MATCHES BEFORE RANSAC</u>

Number of matches between images 0 and 1: 75295
Number of matches between images 1 and 2: 67155

Feature matching involves finding similar points between images by comparing feature descriptors. From SIFT, keypoints (distinct image locations) and descriptors (vectors representing the local image structure) for each keypoint are obtained. Brute Force Matcher computes the Euclidean distance between descriptors in two pair of images.

We use K = 2 nearest neighbor for each keypoint in the other image. To ensure reliable correspondences, we use Lowe's Ratio Test (0.5 threshold) to accept a match only if the closest match is significantly better than the second-best match.

The good matches that pass this test are used to compute homography.

Matches between images 0 and 1

Matches between images 1 and 2

HOMOGRAPHY MATRIX

Homography is a matrix that transforms points from one image plane to another, preserving straight lines. It aligns images in such a fashion that it helps in overlapping. It transforms one image's keypoints into the coordinate space of another.

However, real-world images are noist and may have wrong feature matches, which can distort the homography estimation. Therefore, RANSAC is used which estimates homography by continuously selecting random subsets of good matches and calculates the transformation. It identifies the subset with the highest number of inliers while rejecting outliers thus making sure that points align geometrically too. This eliminates noisy or imperfect data, to get right image alignment and hence stitching.

[[1.45415673e+00 -4.32776151e-02 -1.31312565e+03]

 [2.14515069e-01 1.27444890e+00 -5.60453583e+02]

 [1.09951897e-04 5.00654694e-06 1.00000000e+00]]

[[1.51883909e+00 -6.19845574e-02 -1.49145846e+03]

 [2.37108494e-01 1.30734028e+00 -5.65620314e+02]

 [1.27651665e-04 -1.57846525e-06 1.00000000e+00]]

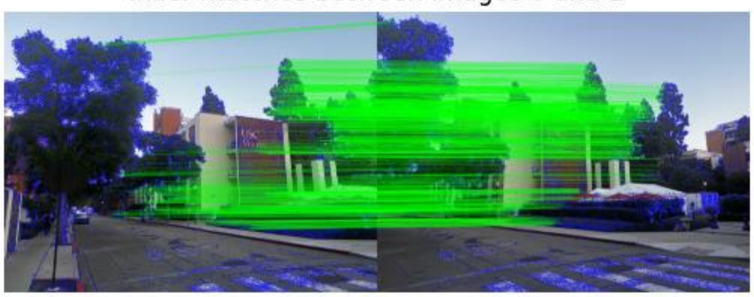
Number of inlier matches between images 0 and 1: 1928
Number of inlier matches between images 1 and 2: 1639

Top 10 inlier matches with minimum error between images 0 and 1

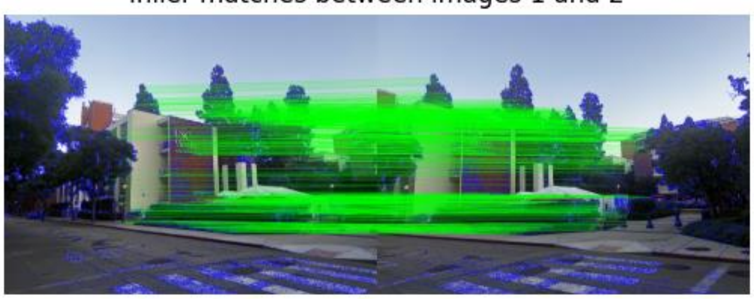
Top 10 inlier matches with minimum error between images 1 and 2

Total inlier matches

Inlier matches between images 0 and 1



Inlier matches between images 1 and 2



The inliers are present more around the buildings as opposed to some of the good matches present previously on the road too.

WARPED IMAGES

Transformed Image 1

Transformed Image 2

Transformed Image 3

Cummulative homography is calculated so that the alignment of first image can be forwarded to the subsequent images for the perfect stitching.

<u>FINAL IMAGE</u>

Panorama

CODE:

```
import numpy as np
import cv2
import sys
import glob
import matplotlib.pyplot as plt

image_paths = glob.glob(r"D:\USC\Advanced Computer Vision\Assignment 2\*.jpg")
if not image_paths:
    sys.exit("Error: No images found in the directory")

print(image_paths)

# Load images
images = []
for path in image_paths:
    img = cv2.imread(path)
    if img is None:
        print(f"Error: Could not load image {path}")
        continue
    images.append(img)

# Convert to grayscale, detect keypoints and descriptors
sift = cv2.SIFT_create()
keypoints = []
descriptors = []
count = 0
for img in images:
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    kpts, dpts = sift.detectAndCompute(gray, None)
    # show the detected features overlaid on the images. Also give out the number
of features detected in each image.
    img1=cv2.drawKeypoints(img,kpts, None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    count += 1
    plt.imshow(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB))
    plt.title(f"Image {count} with keypoints detected")
    plt.axis('off')
    plt.show()
    print(f"Number of features detected in image {len(keypoints)}: {len(kpts)}")
    keypoints.append(kpts)
    descriptors.append(dpts)

# Match features
```

```
bf = cv2.BFMatcher()
good_matches = []

for i in range(len(descriptors) - 1):
    matches = bf.knnMatch(descriptors[i], descriptors[i + 1], k=2)
    print(f"Number of matches between images {i} and {i+1}: {len(matches)}")
    good = []
    for m, n in matches:
        if m.distance < 0.5 * n.distance:  # Ratio test
            good.append([m])
    good_matches.append(good)

    #draw top 10 matchesKNN
    #sort matches based on distance
    good = sorted(good, key = lambda x:x[0].distance)
    img_matches = cv2.drawMatchesKnn(images[i], keypoints[i], images[i+1],
keypoints[i+1], good[:10], None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    plt.imshow(cv2.cvtColor(img_matches, cv2.COLOR_BGR2RGB))
    plt.title(f"Top 10 matches before RANSAC between images {i} and {i+1}")
    plt.axis('off')
    plt.show()

# Find homography
homographies = []
for i in range(len(good_matches)):
    if len(good_matches[i]) >= 4:
        src_pts = np.float32([keypoints[i][m[0].queryIdx].pt for m in
good_matches[i]]).reshape(-1, 1, 2)
        dst_pts = np.float32([keypoints[i + 1][m[0].trainIdx].pt for m in
good_matches[i]]).reshape(-1, 1, 2)
        H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
        #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.
        inlier_count = np.sum(mask)
        print(f"Number of inlier matches between images {i} and {i+1}:
{inlier_count}")

        # Reprojection error calculation for each inlier
        def reprojection_error(src_pt, dst_pt, H):
            src_pt_proj = np.dot(H, np.array([src_pt[0], src_pt[1], 1]))
            src_pt_proj /= src_pt_proj[2]  # Normalize
            error = np.linalg.norm(src_pt_proj[:2] - dst_pt)
            return error
```

```
        inlier_matches = [m for j, m in enumerate(good_matches[i]) if mask[j] ==
1]
        errors = []
        for m in inlier_matches:
            src_pt = keypoints[i][m[0].queryIdx].pt
            dst_pt = keypoints[i+1][m[0].trainIdx].pt
            error = reprojection_error(src_pt, dst_pt, H)
            errors.append((m, error))

        # Sort based on reprojection error and get the top 10 matches with least
error
        errors.sort(key=lambda x: x[1])
        top_10_matches = [e[0] for e in errors[:10]]

        # Display top 10 matches with minimum reprojection error
        img_top_matches = cv2.drawMatchesKnn(images[i], keypoints[i],
images[i+1], keypoints[i+1], top_10_matches, None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
        plt.imshow(cv2.cvtColor(img_top_matches, cv2.COLOR_BGR2RGB))
        plt.title(f"Top 10 inlier matches with minimum error between images {i}
and {i+1}")
        plt.axis('off')
        plt.show()

        print(H)
        homographies.append(H)
    else:
        print(f"Warning: Not enough matches between images {i} and {i+1}")
        homographies.append(None)

count = len(images)
idx_center = count // 2

cumulative_homographies = [np.eye(3) for _ in range(count)]
for i in range(idx_center-1, -1, -1):
    cumulative_homographies[i] = np.dot(homographies[i],
cumulative_homographies[i+1])
for i in range(idx_center, count-1):
    cumulative_homographies[i+1] = np.dot(np.linalg.inv(homographies[i]),
cumulative_homographies[i])


min_x = min_y = max_x = max_y = 0.0
for i in range(len(images)):
```

```
    # Get the height and width of the original images
    h, w, p = images[i].shape
    # Create a list of points to represent the corners of the images
    corners = np.array([[0, 0], [w, 0], [w, h], [0, h]], dtype=np.float32)
    # Calculate the transformed corners
    transformed_corners = cv2.perspectiveTransform(corners.reshape(-1, 1, 2),
cumulative_homographies[i])
    # Find the minimum and maximum coordinates to determine the output size
    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 width and height of the stitched image
output_width = int(max_x - min_x)
output_height = int(max_y - min_y)

# blend the transformed images
panorama = np.zeros((output_height, output_width, 3), dtype=np.uint8)
warped_images = []

for i in range(len(images)):
    # create offset transformation
    offset_x = int(-min_x)
    offset_y = int(-min_y)
    transformation = np.array([[1, 0, offset_x], [0, 1, offset_y], [0, 0, 1]])
    # warp images
    warped = cv2.warpPerspective(images[i], np.dot(transformation,
cumulative_homographies[i]), (output_width, output_height),
flags=cv2.INTER_LINEAR, borderMode=cv2.BORDER_CONSTANT)
    warped_images.append(warped)
    # create logical mask for warped image and previous panorama
    mask = np.where(warped > 0, 1, 0).astype(np.uint8)
    panorama_mask = np.where(panorama > 0, 1, 0).astype(np.uint8)
    # alpha blending
    panorama = np.where(mask == 1, warped, panorama)
    alpha = 0.5
    panorama = np.where((mask == 1) & (panorama_mask == 1), (warped * alpha +
panorama * (1-alpha)).astype(np.uint8), panorama)

# display the panorama
fig = plt.figure(figsize=(10, 8), dpi=200)

plt.imshow(cv2.cvtColor(panorama, cv2.COLOR_BGR2RGB))
plt.title("Panorama", fontsize=8)
```

```
plt.axis('off')
plt.show()

for i in range(len(warped_images)):
    fig = plt.figure(figsize=(10, 8), dpi=200)
    plt.imshow(cv2.cvtColor(warped_images[i], cv2.COLOR_BGR2RGB))
    plt.title(f"Transformed Image {i+1}", fontsize=8)
    plt.axis('off')
    plt.show()
```