AIN433 Introduction to Computer Vision Lab.
Practical 2 - Practical Info

Mehmet Akif Özgür
2200356832
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
ozgurmehmetakif@hotmail.com

Part 1 - Feature Extraction

I extracted key points in the sub-images by a key point extraction method (SIFT/SURF and ORB).

SIFT (Scale-Invariant Feature Transform) and SURF (Speeded Up Robust Features) are both algorithms for detecting and describing local features in an image. These features can be used for tasks such as object recognition, image stitching, and 3D reconstruction. SIFT identifies key points in an image based on scale-space extrema, meaning it looks for points where the difference of Gaussian (DoG) filter response is maximal in scale-space. It then computes descriptors for these key points based on the local image gradients. SIFT is robust to scale, rotation, and illumination changes, making it a popular choice for computer vision tasks.

SURF is a faster version of SIFT that uses a faster algorithm for computing the descriptors. It uses Haar wavelets to compute the image gradients and approximates the Gaussian smoothing using box filters. This allows SURF to be more efficient than SIFT while still being robust to scale, rotation, and illumination changes.

ORB (Oriented FAST and Rotated BRIEF) is another key point extraction method that is designed to be both fast and robust. ORB uses the FAST (Features from Accelerated Segment Test) corner detector to identify key points and then computes binary descriptors using BRIEF (Binary Robust Independent Elementary Features). ORB also incorporates a rotation-invariant version of the FAST algorithm and orientation estimation for each key point.

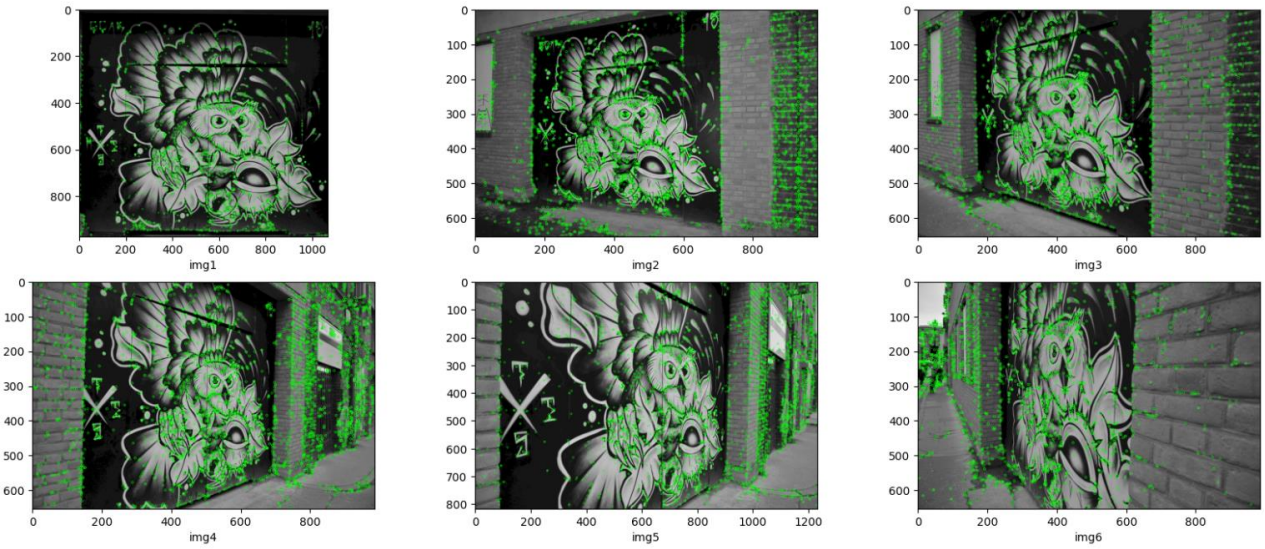
I used OpenCv library for feature extraction. The **detectAndDescribe** function creates a feature descriptor object based on the specified feature detection method using OpenCV's xfeatures2d module for SIFT and SURF and the ORB_create() function for ORB. This object will be used to detect keypoints and compute descriptors for the input image.

The function then uses the detectAndCompute() method of the feature descriptor object to detect the keypoints and compute descriptors for the input image. This method takes the input image as its first argument and an optional mask as its second argument. The output of the detectAndCompute() method is a tuple containing the detected keypoints and their corresponding descriptors.

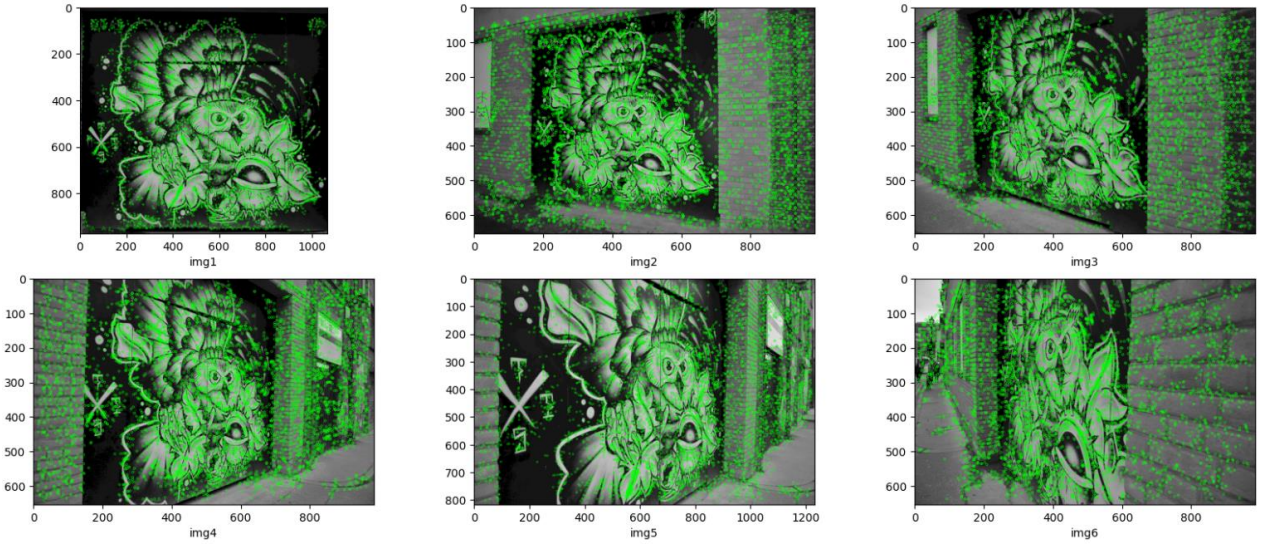
Finally, the function returns the tuple of keypoints and descriptors as its output.

Below are only the photos in the v_bird folder. For other photos, you can make the images variable one of "images_bird, images_boat, images_circus, images_graffiti, images_soldiers, images_weapons" in the code.

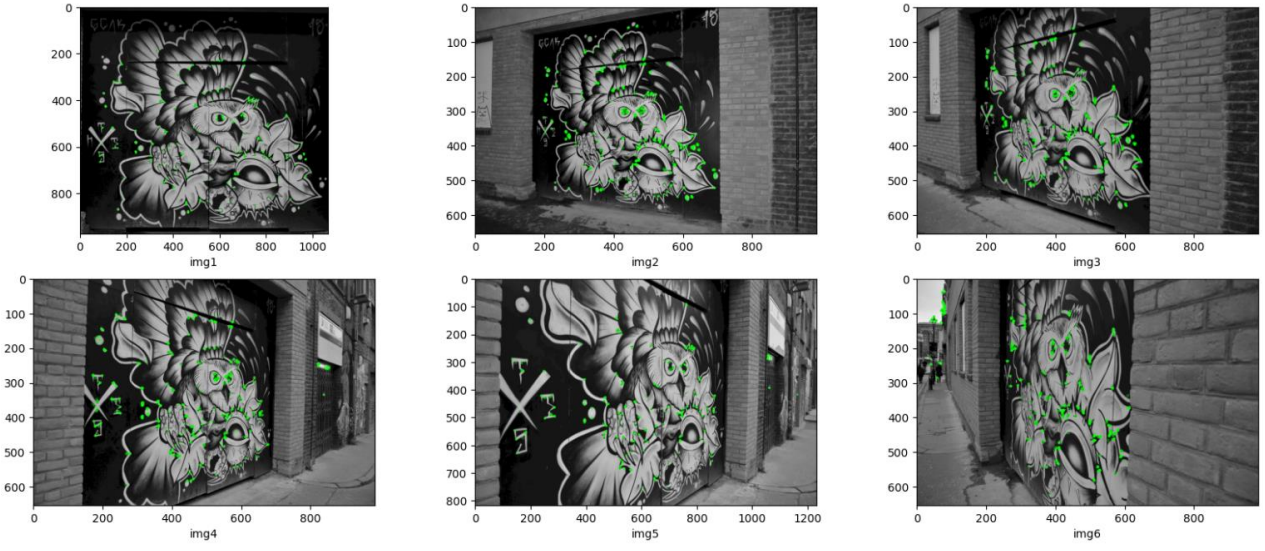
SIFT



SURF



ORB



Part 2 - Feature Matching

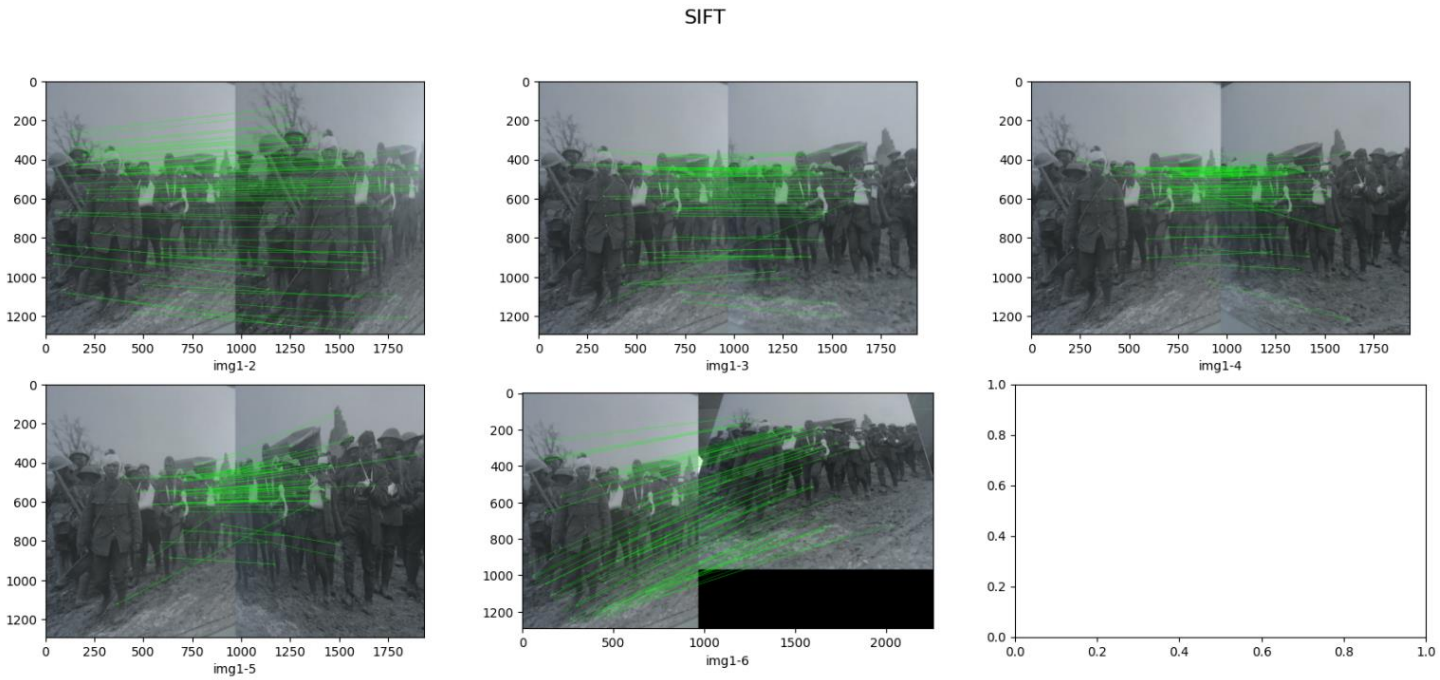
I coded a matching function (I used a function that based k-nearest neighbor method) to match extracted key points between pairs of sub-images. These are pairs of sub-images -> 1-2, 1-3, 1-4, 1-5, and 1-6.

The matchKeyPointsKNN function first creates a matcher object based on the specified matching method using the createMatcher() function. This function returns a matcher object that can be used to match key points based on their descriptors. The matcher object is initialized with the crossCheck parameter set to False, indicating that the matching should not be done in both directions.

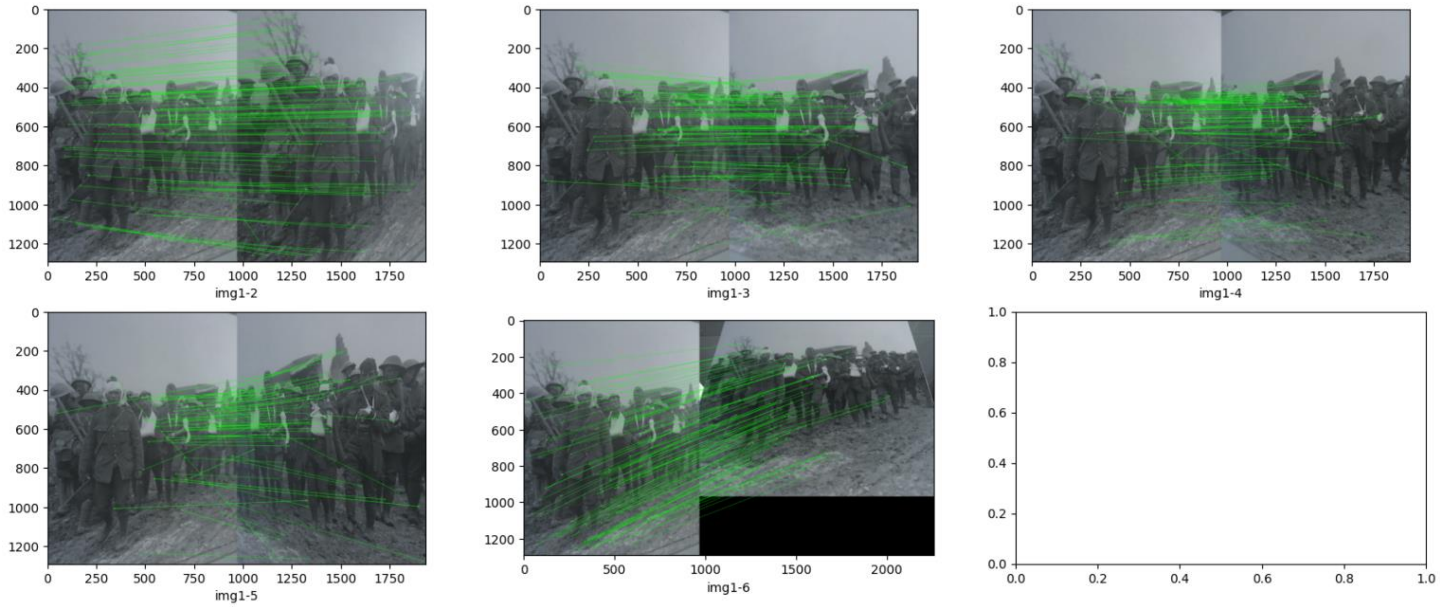
The function then uses the matcher object to compute the crude matches between the two sets of descriptors using the knnMatch() method. This method takes the descriptors for the first image as its first argument, the descriptors for the second image as its second argument, and the number of nearest neighbors to consider as its third argument. The output of the knnMatch() method is a list of lists, where each sub-list contains the indices of the descriptors in the second image that are the closest and second-closest matches to the descriptor in the first image.

The function then loops over the crude matches and applies the Lowe's ratio test to filter out any matches that are not sufficiently distinctive. For each crude match, the function checks if the distance to the closest match is less than a certain ratio times the distance to the second-closest match. If this condition is met, the match is considered to be a good match and is added to the list of actual matches.

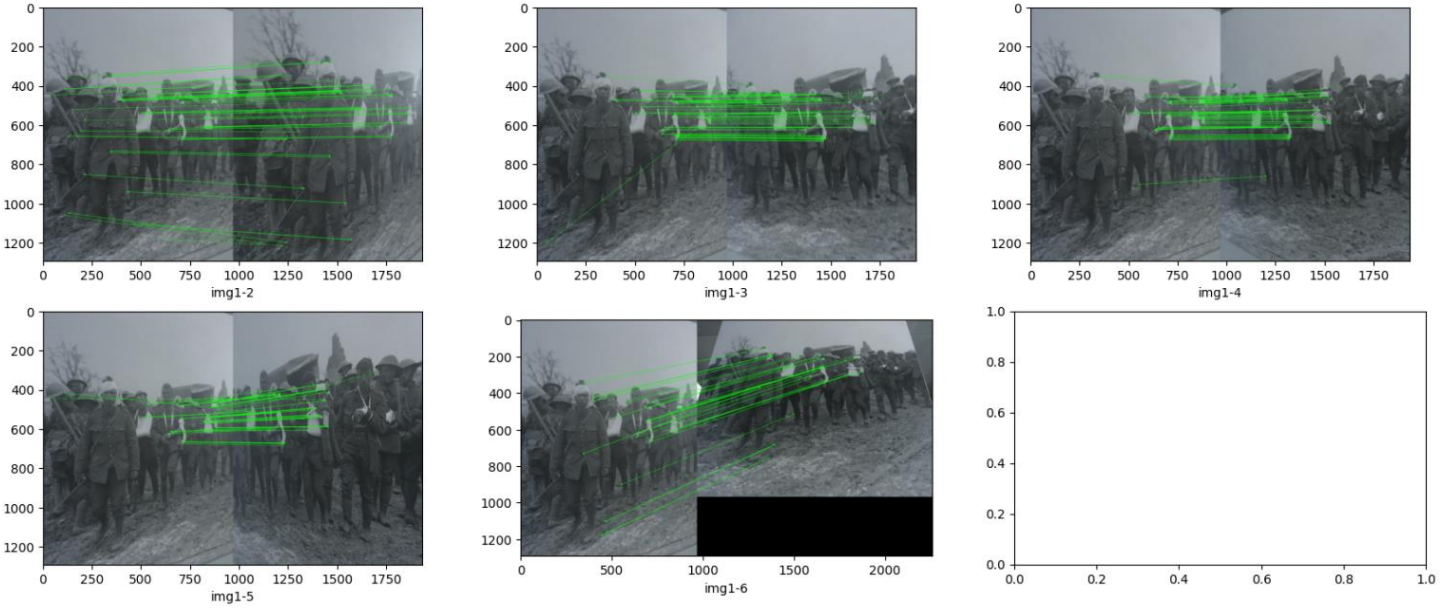
Finally, the function returns the list of actual matches as its output.



SURF


img1-2
img1-3
img1-4
img1-5
img1-6

ORB


img1-2
img1-3
img1-4
img1-5
img1-6

Part 3 - Finding Homography

I calculated a Homography Matrix for each pair of sub-images. (1-2, 1-3, 1-4, 1-5, 1-6).

The getHomography() function first converts the lists of keypoints into NumPy arrays. Then, it checks if the number of matches is greater than 4, which is the minimum number of points required to estimate a homography matrix. If so, it constructs two sets of points using the matched keypoints, and estimates the homography matrix between the two sets of points using the cv2.findHomography() function with RANSAC algorithm and the provided threshold value. The function returns a tuple containing the matches, the estimated homography matrix, and the status of the estimation. If the number of matches is less than or equal to 4, the function returns None.

Part 4 - Merging by Transformation

In this part, we had to combine the pictures given to us according to the homography matrix. But even though I tried many ways, I couldn't get the right result. That's why I just did the joins between pairs 1-2, 1-3, 1-4, 1-5 and 1-6 and finalized my code.