BBM418 Computer Vision Laboratory
Assignment 2 - Image Panorama Stitching

Kayla Akyüz
21726914
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
`b21726914@cs.hacettepe.edu.tr`

Overview

"In this assignment, you will merge sub-images provided by using keypoint description methods (SIFT,SURF and ORB) and obtain a final panorama image that including all scenes in the sub-images." [1]

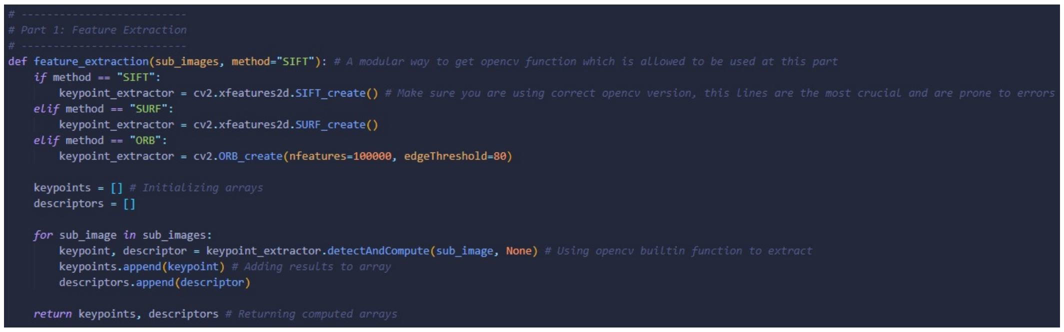
The key point of this assignment is being able to understand and developing a function for finding homography with Ransac method and developing a function for merging by transformation. Since OpenCV can be used for the rest of the assignment, students are also to gain familiarity with the allowed parts of the OpenCV.

For this purpose I have created a solution tool kit in python that has all needs from reading images to warping with homography as well as merging with alpha and much more. For easier using and viewing results I also created a notebook file which I strongly suggest you to check. Notebook file will have more images than here. When the tool kit is run it produces results to disk, the produced results will already be in submitted folders. The most comprehensive results can be found inside those folders.

The code has lots of comments and easy to understand, also code is easy to run especially if the notebook is used. Feel free to check them out too!

Part 1 - Feature Extraction

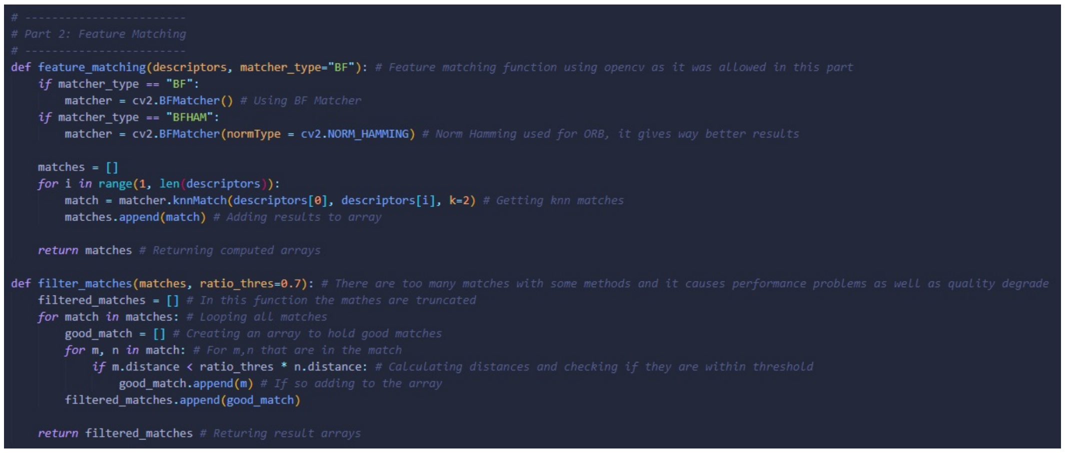
Since we are allowed to use OpenCV functions in this part, I have created a simple function that calls required extractor. However there was a problem with SURF being licensed and new versions of OpenCV not containing it. Thankfully a class mate already asked and answered this problem on Piazza. There is no thing else to mention even the code is so simple I can put it here:



```
# --------------------------
# Part 1: Feature Extraction
# --------------------------
def feature_extraction(sub_images, method="SIFT"): # A modular way to get opencv function which is allowed to be used at this part
    if method == "SIFT":
        keypoint_extractor = cv2.xfeatures2d.SIFT_create() # Make sure you are using correct opencv version, this lines are the most crucial and are prone to errors
    elif method == "SURF":
        keypoint_extractor = cv2.xfeatures2d.SURF_create()
    elif method == "ORB":
        keypoint_extractor = cv2.ORB_create(nfeatures=100000, edgeThreshold=80)

    keypoints = [] # Initializing arrays
    descriptors = []

    for sub_image in sub_images:
        keypoint, descriptor = keypoint_extractor.detectAndCompute(sub_image, None) # Using opencv builtin function to extract
        keypoints.append(keypoint) # Adding results to array
        descriptors.append(descriptor)

    return keypoints, descriptors # Returning computed arrays
```

Part 2 - Feature Matching

We are allowed to use OpenCV functions in this part too, so similarly I have created a simple function that calls required functions. However there was a problem with BF matcher. It was not producing good results for ORB extractions. After testing a lot of parameters I encountered normType, and with research I decided to add ability to chose NORM HAMMING. That way ORB method also got matched good. Since the code is still small I will add it as image, you can browse the code for the other parts as the code has extensive comments and descriptions and easy to understand:



```python
# ------------------------
# Part 2: Feature Matching
# ------------------------
def feature_matching(descriptors, matcher_type="BF"): # Feature matching function using opencv as it was allowed in this part
    if matcher_type == "BF":
        matcher = cv2.BFMatcher() # Using BF Matcher
    if matcher_type == "BFHAM":
        matcher = cv2.BFMatcher(normType = cv2.NORM_HAMMING) # Norm Hamming used for ORB, it gives way better results

    matches = []
    for i in range(1, len(descriptors)):
        match = matcher.knnMatch(descriptors[0], descriptors[i], k=2) # Getting knn matches
        matches.append(match) # Adding results to array

    return matches # Returning computed arrays

def filter_matches(matches, ratio_thres=0.7): # There are too many matches with some methods and it causes performance problems as well as quality degrade
    filtered_matches = [] # In this function the mathes are truncated
    for match in matches: # Looping all matches
        good_match = [] # Creating an array to hold good matches
        for m, n in match: # For m,n that are in the match
            if m.distance < ratio_thres * n.distance: # Calculating distances and checking if they are within threshold
                good_match.append(m) # If so adding to the array
        filtered_matches.append(good_match)

    return filtered_matches # Returing result arrays
```

Part 3 - Finding Homography

This is the first part where we are to use only NumPy and code the requirement ourselves. What I did is researching the required tasks and the formulas to get better understanding. After then I started coding so that I could test out. However since there were some issues with the previous part (image pairs being unclear) for a long time I thought I was doing wrong in this part. It took me a while to clear this part out.

My implementation for this part has total of 5 functions. Since I coded everything in mind also with testing in Jupyter, I created a main function that will call rest of the necessary calls. This way the Jupyter file needed to call only one function.

This first function is called iterate_method in the means it will iterate methods matching to find homography. It takes (method, sub_images, referance_is_0 = True, ransac = 2000, non_filtered = False, matcher="BF", filter_ratio=0.7, verbose=True) as parameter and returns homographies, runtime, keypoints, filtered_matches. These returns later on used to print necessary images and tables, as well as used in the next part where the images are warped and merged. This function it self also measures runtime so that the methods performance can be compared. However just to note my comparison results might have been effected by other programs running.

I am not sure if I should start extensive explaining here, as it is hard to explain functions with text and my code has lots of comments it self, so in order to keep things simple I will move on to the next function.

Second function is find_homography(keypoints, filtered_matches, referance_is_0=True, ransac = 2000). It sets up the points in a loop of filtered_matches then calls the ransac function.

Third function is ransac_homography(src_pts, dst_pts, iterations=2000, threshold=3). I had to look at resources to get grasp of what to be coded [2]. After I understood I started coding. My ransac function gets random 4 indices (non repeated) and calculates homography with them. Then apply the calculated homography to all of the points and checks if they are inlier. The randomly calculated best homography meaning the one with most inlier gets chosen at the end and returned.

I also have compute_homography_matrix(src_pts, dst_pts) function with normalize_points(pts) inside. To create this function I had to look formulas[3][4][5] and resources[6]. I basically normalize points and stack them in matrix as in the formula down below and solve with np.linalg.svd. Then I get homography vector from VT and reshape it to a matrix. Here is the formula:

$$PH = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x'_1 & y_1x'_1 & x'_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y'_1 & y_1y'_1 & y'_1 \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x'_2 & y_2x'_2 & x'_2 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y'_2 & y_2y'_2 & y'_2 \\ -x_3 & -y_3 & -1 & 0 & 0 & 0 & x_3x'_3 & y_3x'_3 & x'_3 \\ 0 & 0 & 0 & -x_3 & -y_3 & -1 & x_3y'_3 & y_3y'_3 & y'_3 \\ -x_4 & -y_4 & -1 & 0 & 0 & 0 & x_4x'_4 & y_4x'_4 & x'_4 \\ 0 & 0 & 0 & -x_4 & -y_4 & -1 & x_4y'_4 & y_4y'_4 & y'_4 \end{bmatrix} \begin{bmatrix} h1 \\ h2 \\ h3 \\ h4 \\ h5 \\ h6 \\ h7 \\ h8 \\ h9 \end{bmatrix} = 0$$

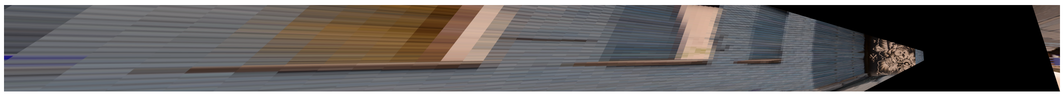
Part 4 - Merging by Transformation

This was the hardest part as it required lots of tips and tricks to be know. For example the warped image would have data mapping to negative region. In order to show as much as data I can I tried to transform images, however it required me to know warped size before hand. While predicting warped size I encountered Integer Overflow error and needed to figure it out. This kind of tiny bits of important insights made Part 4 really hard to overcome, however I accomplished it.

This part is going to have two subsections dedicated to warping image and merging the warped results. Before getting to those subsections let me describe my warped image size prediction function get_warped_image_bounds. This function simply takes image and homography and constructs a vectorized array from the height and width of the image. This vectors represent the corners. And apply the homography to predict where the corners will land.

The hard part of this is that the corners can land anywhere, even sticking from the other side due to Integer Overflow. Of course since I was no expert, a lot of trials and stress passed to get the working solution.

The warped image that got integer overflow so it has parts sticking from right:



I solved these issues by creating a smart solution from understanding abnormalities by which corners are landing where. To make panoramas reasonable I put limit when the image was stretching too much.

Applying Homography Matrix (Warping)

Applying homography matrix as easy as doing a dot product however, doing the correct checks to create meaningful image is hard. One of the main ideas is that instead of applying matrix to the image, a canvas gets created from the predicted corners and the matrix applied in reverse to the canvas. This is actually very logical and can be explained like this; consider there is a pixel and the matrix stretches the image, in the canvas the pixel next to it will have a margin in between, because all of those pixels on canvas actually equal to one pixel from the source image. In the reverse sense you always check the pixels data from the source image so there are no empty pixels or glitches. Of course the correct checks so while looking up the pixels from the source image, trying not to access invalid indices as well as not accessing invalid indices on canvas is crucial.

In order to accomplish that I create a canvas from predicted images. Then populate it with huge arrays of vectors created with np.meshgrid, ranging from the minimum predicted value to maximum predicted value. Then I do dot product with the inverse homography matrix. Then I get a mask from the source image and apply it to both matrices and do the index checks.

After the valid match has been created, I shift the canvas by the minimum value, this is to make negative regions visible. Side effect is it will mess up the alignment of homography matrix that comes from the reference image. However this transform is reversed when merging so that warped images align.

Constructing The Panorama Image (Merging)

Merging function takes images and homographies as parameter. It also takes a boolean called alpha to determine if an experimental merging method will be used. My initial merging method was stacking images on top of each other, this is actually correct for panorama however I saw some glitches so I thought if the panorama can be calculated in a transparent manner.

The solution is keeping a second array sized as big as panorama image and incrementing each pixels integer by one whenever it gets written, and instead of overwriting the pixel color in the actual panorama, I decided to sum it. This way lets say a pixel gets written on three times, if it is white color the pixel value will be 765,765,765, dividing by 3 later on gives 255,255,255 which is the actual white color, this way the pixel interpolates in between all the colors written on to it.

Another trick to accomplish was predicting panorama image size, if given too little warped images would clip or invalid indices might try to be accessed.

I also reverse the warped image transformation that is done by me to fit negative regions so that alignment is ensured, then I transform the merged image all together with the maximum negative region size so that panorama image displays as much as data as it can.

Other then these functions I have created various functions to display, read, write, images, files, as well as a __main__ function so that when the python file is called locally it saves image results to disk.

Also as mentioned I created notebook files, the main one being notebook-interface.ipynb that provides super easy access to code and also has lots of results with in.

Results

The complete results will be shared in image_out folder. There are also .ipynb files with the results and codes to calculate any desired result with in them.

Still I will add some of the results here too, as they were required. I just hope the code and the notebook files gets the view because the real work resides there.

There are more results in folders and notebooks, such as ground truth calculated images etc.

Also one note I want to mention is that in the PDF it says pairs of 1-2, 2-3. As I mentioned before and in other places, the given dataset is more suited for 1-2, 1-3 matching. That is the way I created panoramas. However with my code 1-2, 2-3 matching can also be done.

THIS PDF DOES NOT HAVE RESULTS! This is a safe PDF just in case 800mb+ Report.pdf does not open!

References

[1] Assoc. Prof. Nazlı İkizler Cinbiş. Image Panorama Stitching. BBM 418 - Computer Vision Laboratory, 2023.

[2] Random sample consensus. In Wikipedia. Retrieved April 14, 2023, from https://en.wikipedia.org/wiki/Random_sample_consensus

[3] Homography (computer vision). In . Retrieved April 14, 2023, from https://en.wikipedia.org/wiki/Homography_(computer_vision)

[4] OpenCV. (n.d.). Homography Examples using OpenCV (C++/Python). In OpenCV documentation. Retrieved April 17, 2023, from https://docs.opencv.org/4.x/d9/dab/tutorial_homography.html

[5] How to compute Homography matrix H from corresponding points (2d-2d planar Homography)? In Mathematics Stack Exchange. Retrieved April 18, 2023, from https://math.stackexchange.com/questions/494238/how-to-compute-homography-matrix-h-from-corresponding-points-2d-2d-planar-homog

[6] Socret Lee. (Jan 31, 2022). Understanding Homography a.k.a Perspective Transformation. In Towards Data Science. Retrieved April 18, 2023, from https://towardsdatascience.com/understanding-homography-a-k-a-perspective-transformation-cacaed5ca17