November 17th, 2020

Matan Segal

msegal@smu.edu

CS 5390 – Fall 2020

Project 2 – Feature Detection and Matching

Introduction:

In the project we requested to identify pairing between point in one image and a corresponding point in another image. These correspondences can be used to stitch multiple images together into panorama. To achieve this goal, we divided the problem into three steps. First is feature detection – identify points of interest using the Harris corner detection method. Second is feature description – once we have the corner points, we will use the 5x5 frame around the pixels we found earlier in order to describe them and use it later. Lastly is the feature matching – we will use the sum of squared differences (SSD) between the frames in the first image and the frames in the second images so we will be able to find similar points in the two images by finding the least SSD value for each frame.

Implementation Overview:

In my implementation I followed the details in the descriptions. For the first part, after loading the images, I created the function *find_corners()* to find the corners in each image. In this function, first, I convert the image into a grey scale, then I called the *cornerHarris()* function. I set the block size variable, which is the dimensions of the H mat to 2, and the dimension of the Sobel kernel to 3, by the variable apertureSize. The k value I used is 0.04 which refers to the precision of the corners we will get – smaller value will tolerate false corners, but too high value will prevent real corner to show up. The k value used in the equation below:

$$R = det(M) - k(trace(M))^2 = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

Figure 1 – The equation used to find the corners in the cornerHarris function [1]

After having the destination Mat from the *cornerHarris()* function, I normalized it to values between 0 to 255 and search for values higher than the threshold value I chose. I chose 90 for the first image and 110 for the second image after testing different values and receiving the best result for those. When I checked for passing the threshold, I also searched for the maxima value in the 7x7 frame to make sure I have no repetitive corners. Once a point stand with these standards I pushed it to the corners vector.

For the second part, I created the function *set_mat_vector()* which create a vector, named kernels, contains the 5x5 frame around each corner I have in the corners vector. I created it in a such a way that there is a correlation in the position of the corner and its related frame in the kernels vector. For edge handling, I skipped points in frames where they are out of the image (if the corner is very close to the edge its frame might be out of the image) and left them as 205, which is the default value when creating a Mat object.

For the third part, I created the function *set_corralate_corners().* In this function I have an outer loop going through the first kernels vector and inner loop going through the second kernels vector. In this way I checked for each frame in the first vector what is the frame with the least different sum value from the second vector. I did not use a sum of square differences, but the function *absdiff()* which find the absolute value difference between each pixel of two frame. Once I had the different Mat object, *diff,* I calculated the total sum of all values in the Mat. After receiving the point whose frame is the most similar to the first point's frame, first I check If the sum is less than a threshold I set (I chose 800 after testing). Then, I check if it already has a paired point. If it does have a pair, I check the sum value I calculated for both and save the point with the smaller one. If there is no such a point, I save it in my correlated_corners vector. This vector contains the two points of the correlated corner as well as the different sum value I calculated in order to compare with later points. Once I finish looping through all points, I print to the screen all the correlated points with their different sum value.

For demonstration the result, I decided to draw circles around **all** the corners I could find in both pictures. I horizontally concatenated both images into one output image separated by a black line showing the border between the two images. Using the corralate_corners vector, I draw green lines between correlated points to show the user the result on the output image.

Flow Diagram:

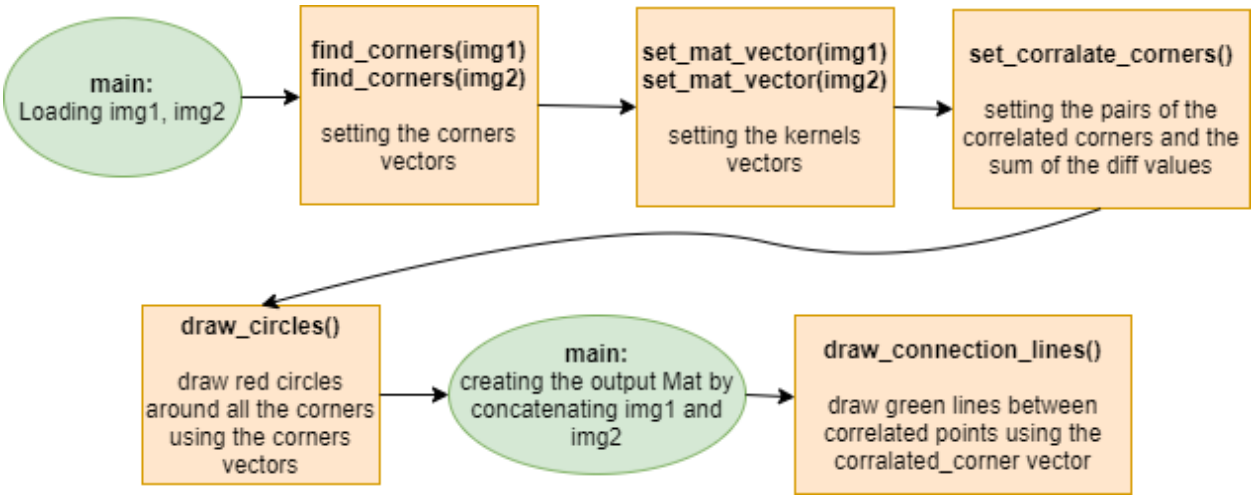

Figure 2 – flow diagram of the program. In bold – the name of the function, below is the description

Results:

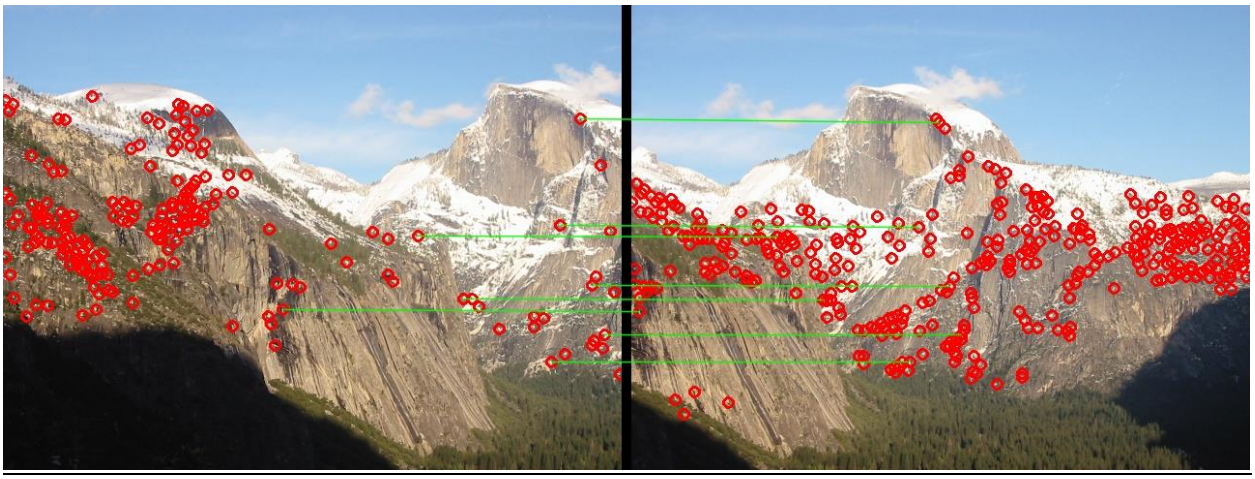

Figure 3 – result correlated point in the Yosemite images

Correlated points:

```
p1: [597, 117] , p2: [318, 121] || diff: 658
p1: [576, 227] , p2: [297, 229] || diff: 762
p1: [429, 238] , p2: [151, 239] || diff: 746
p1: [609, 289] , p2: [329, 290] || diff: 715
p1: [476, 303] , p2: [198, 304] || diff: 793
p1: [289, 314] , p2: [8, 316] || diff: 749
p1: [622, 340] , p2: [341, 340] || diff: 731
p1: [567, 368] , p2: [288, 368] || diff: 668
```

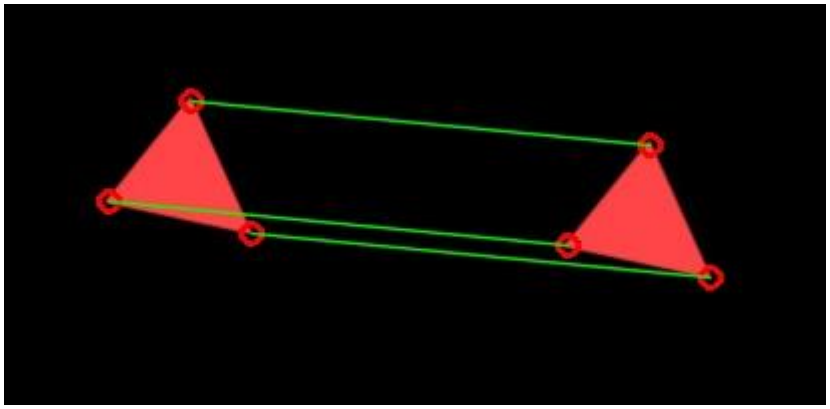

Figure 4 - result correlated point in the triangle images

Correlated points:

```
p1: [93, 48] , p2: [112, 70] || diff: 326
p1: [52, 98] , p2: [71, 120] || diff: 318
p1: [123, 114] , p2: [142, 136] || diff: 250
```

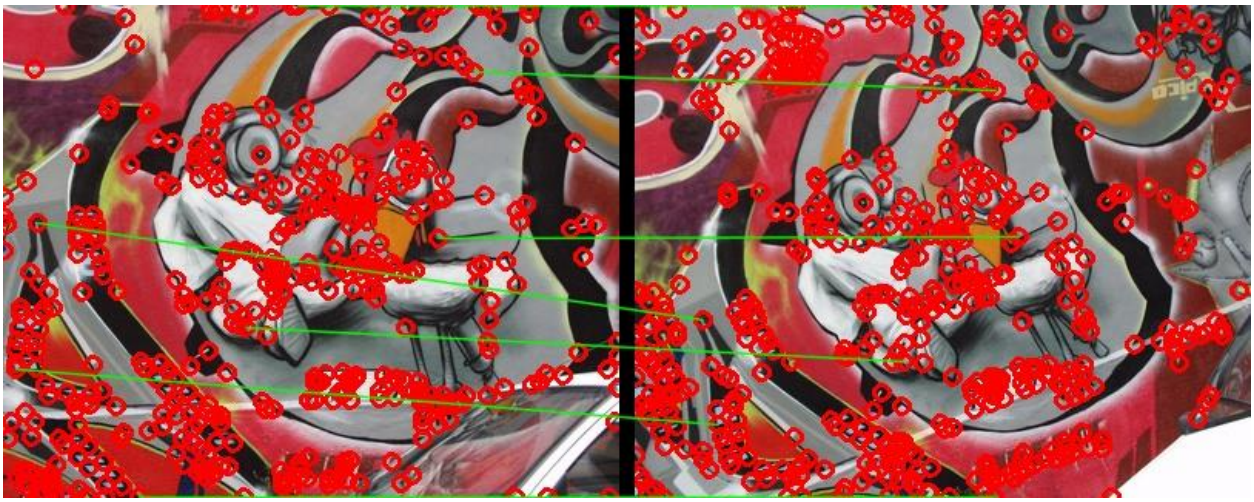

Figure 5 - result correlated point in the graffiti images (img1 and img2)

Correlated points:

```
p1: [305, 43] , p2: [234, 55] || diff: 668
p1: [22, 141] , p2: [44, 204] || diff: 783
p1: [282, 151] , p2: [249, 150] || diff: 613
p1: [157, 209] , p2: [174, 231] || diff: 768
p1: [8, 236] , p2: [54, 272] || diff: 694
p1: [88, 319] , p2: [233, 319] || diff: 778
```

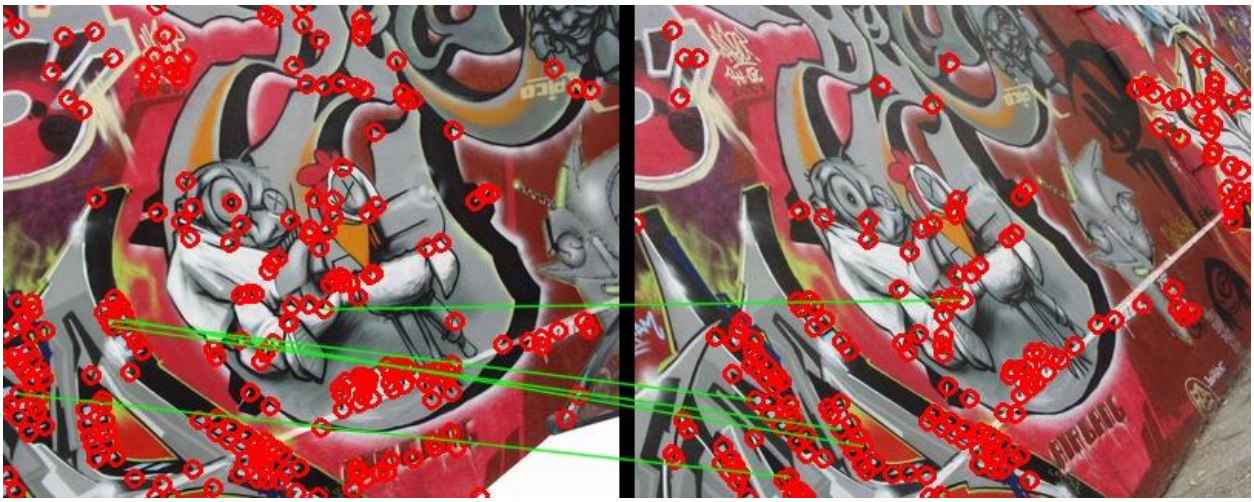

Figure 6 - result correlated point in the graffiti images (img2 and img4)

Correlated points:

```
p1: [208, 197] , p2: [213, 191] || diff: 941
p1: [68, 204] , p2: [78, 256] || diff: 947
p1: [79, 204] , p2: [148, 286] || diff: 818
p1: [70, 208] , p2: [133, 275] || diff: 672
p1: [10, 252] , p2: [100, 306] || diff: 919
```

We can see that in the first three result (figures 3,4, and 5) we receive very accurate results, and the program could find similar points. However, on the last result, with img2 and img4, we see that there are few mistakes in finding the correlated points. Also, I increased the threshold for the sum of the different to 1000, since with the previous value (800) it found just one pair of correlated points. I can assume that it was harder to the program find correlation on the last result since the images are in pretty large difference angels of view.

Discussion:

I am very satisfied with the results I received as well as the process I went through in order to achieve these results. My biggest struggle along this project was creating the frames around each corner. I confused with setting the x's and the y's to assign the values of the new frame.

```
frame.at<Vec3b>(i, j)[0] = src.at<Vec3b>(y, x)[0];
```

After learning about it a little bit more, I figured out that I switched between the x's and the y's since in this assignment they are refers to rows and cols which are the opposite notations. However, along the debugging process I got more familiar with the OpenCV code and error handling.

For future things to explore I want to get more familiar with the optical flow features and the related functions such as *calcOpticalFlowPyrLK()* as well as the math theorem behind them.

References:

[1] https://docs.opencv.org/3.4/dc/d0d/tutorial_py_features_harris.html