Computer Vision Assignment 2

Part 1:

1) In this part, we take two images as arguments, and we calculate the number of SIFT matches between them. To avoid false matches, we threshold some of these by calculating the ratio between the second closest distance to the first. Once we got our final matches, we visualize them. The execution statement for this part is shown below.

Code:

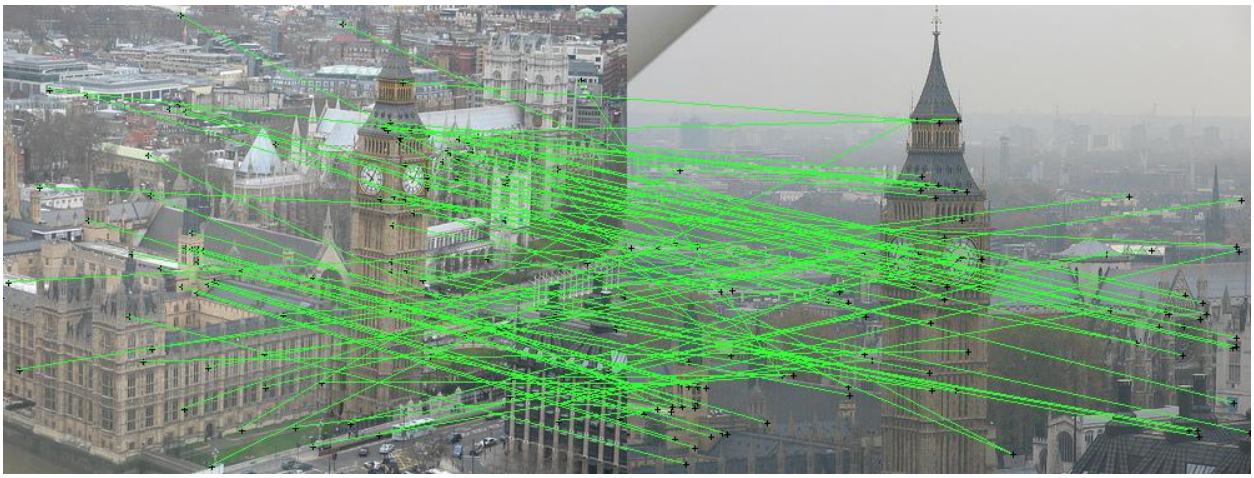
./a2 part1.1 image_1 image_2

Example:

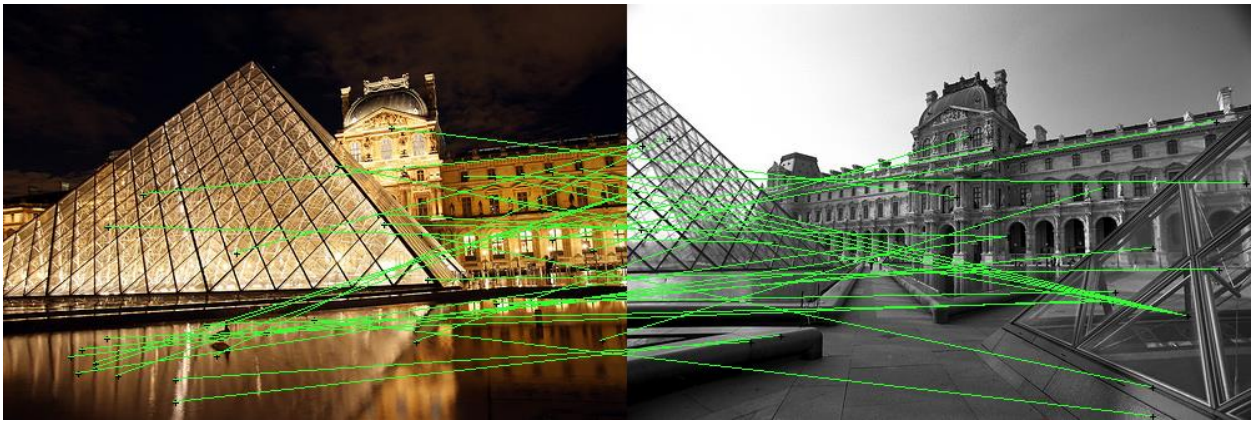
./a2 part1.1 part1_images/bigben_2.jpg part1_images/bigben_3.jpg

A few example outputs are shown below:

-> Bigben_2 and Bigben_3:



-> Lourve_11 and Louvre_13



2) We have our query image and the other set of test images that we need to find which are similar to the query image. Google generally uses something like (more optimized and robust technique) to show “related” images when we click on an image. We implement a basic version of this technique using SIFT descriptors.

We first take the SIFT descriptors for our query image, and find the Euclidean distance between our current descriptor of the query image to all the descriptors in the target image. Then we sort these distances in non-decreasing order. In general, we can use the shortest distance to map the two points. But that can give rise to a lot of false positives. Therefore, we need to implement a simpler, but much more robust method. We take the ratio between the second and first closest distances, and we compare them to a threshold. If this ratio is more than our threshold (hyper-parameter), we include that descriptor as a match, otherwise not. We tried different thresholds (1.5, 1.6, and 1.8) and we finally got settled for 1.7. So, if the ratio between the second closest to the first closest is greater than 1.7, then that SIFT point will be considered a match. Since we consider all the images in this part, all the images will be the output in the order of decreasing SIFT matches.

Code :

./a2 part1.2 query_image directory_of_images

Example:

./a2 part1.2 part1_images/louvre_11.jpg part1_images/*

3) We ran tests for each image category and the best results are shown below. The scores shown in the table are out of ten, and they were not represented as percentage.

Execution Code:

./a2 part1.3 query_image directory-of-images

Example:

./a2 part1.3 part1_images/trafalgarsquare_20.jpg part1_images/*

The trial table is shown below. These are some of the details for the images we tested. The first number is precision, followed by the file extension of the query image.

Tourist Attraction	Precision Score (ID) (queryimage_ID.jpg)
Big Ben	5 (2)
Colosseum	1 (3) 1(4) 1(8)
Eiffel	1(2) 1(19)
Empire State	1 (10) 1 (14) 1 (27) 1 (22)
London Eye	2 [1 (12) 1 (22) 2(23)]
Louvre	1 (10) 1(13) 3(14) 1(4)
Notre Dame	5 (14)
San Marco	2 (13) 2(18) 3(22)
Tate Modern	2(11) 5(14) 3(24)
Trafalgar Square	4(15) 4(5) 6(1)

This is the final table.

Tourist Attraction	Precision Score (10)
Big Ben	5
Colosseum	1
Eiffel	1
Empire State	1
London Eye	2
Louvre	3
Notre Dame	5
San Marco	3
Tate Modern	5
Trafalgar Square	6

The best score came with Trafalgar Square with a precision value of 6, followed by Big Ben and Notre Dame. The number of matching descriptors that will be displayed while executing will be less because our threshold is pretty high, but it also enable us to get some decent results. Sometimes the results were confused, like the Eiffel tower was often confused with Tate Modern, etc.

Sample output when the query image is trafalgarsquare_1.jpg (Precision: 6)
Order of the Table (Rank, Number of matched SIFT features, Image Name)

```
1       35      part1_images/trafalgarsquare_20.jpg
2       26      part1_images/trafalgarsquare_5.jpg
3       24      part1_images/londoneye_9.jpg
4       23      part1_images/tatemodern_8.jpg
5       22      part1_images/londoneye_23.jpg
6       21      part1_images/trafalgarsquare_16.jpg
7       20      part1_images/trafalgarsquare_25.jpg
8       19      part1_images/notredame_14.jpg
9       18      part1_images/trafalgarsquare_6.jpg
10      17      part1_images/trafalgarsquare_15.jpg
```

With this method, as seen from the above results, Trafalgar Square was the easy to find, followed by Big Ben and Notre Dame. The toughest was to find the Colosseum, because sometimes it was not even in the top 10. This might be due to the openness in it, leading it to redundant matches.

Part 2:

2) In this method, we use a vector quantization method to vaguely put, reduce the dimensionality of our SIFT vectors. Our main goal is to represent or "summarize" our 128 – dimensional space into some *k* numbers.

Execution Code:

./a2 part2 query_image directory-of-images

Example:

./a2 part2 part1_images/trafalgarsquare_20.jpg part1_images/*

As in part 1, we tried a few values for k (1, 2, and 5) and w (1, 5, 10), and keeping the speed vs. accuracy point in mind, we decide to limit k to 1. We did try with k = 2 and the results, though okay, took almost the same time as the first part. (It could be due to lack of our knowledge in C++) Assume that we have *x* number of 128-D vectors for an image *A* and *y* number of 128-D vectors for another image *B*. Our goal is to summarize these 128-D vectors into a total of *x* values, and same for image *B*. Once we get the compact representations, we try to find the features on which these values are exactly identical. Once we got those SIFT ID's, we compute the distance between those 128-D space, and we still use a threshold for some better results. Here, instead of second by first, we try to consider only matches that have some distance compared to the threshold.

The time taken for the initial method was on average (scaled across 5 runs) 83 seconds, and for this method is 68 seconds (scaled across 5 runs).The speed improved compared to the first method (almost 1.2x times). When we tried for k = 2, the time taken was 79 seconds on average (scaled across 5 runs). Since k = 1 was working fine, we fine – tuned the code and made it more specific to just one value, so that the implementation can be faster.

Due to the projection quantization, there will be some inconsistencies with some of our SIFT vectors. To overcome that, we set a threshold, and if these values adhere to that value, we take them into consideration. We took some steps to avoid pitfalls, so in the test cases we implemented, everything works good.

The results for this method are shown in the table below. The table follows the same header representation as the one given above.

The trails table is shown first.

Tourist Attraction	Precision Score(ID) (queryimage_ID.jpg)
Big Ben	1 (10) 1(12) 1(16) 1(6)
Colosseum	3 (11) 2(4) 3(13) 2(12)
Eiffel	1 (15) 1(2) 3(22) 1(7)
Empire State	3(10) 2(15) 4(9) 2(14)
London Eye	1(16) 1(17)
Louvre	3 (10) 1(11) 1(13) 2(9) 2(4)
Notre Dame	3 (14) 4(19) 3 (1) 4 (24)
San Marco	1 (13) 1(19) 2 (18) 2 (22)
Tate Modern	1 (4) 1(11)
Trafalgar Square	1 (16) 3(20) 1(25)

The final table is shown below:

Tourist Attraction	Precision Score(10)
Big Ben	1
Colosseum	3
Eiffel	3
Empire State	4
London Eye	1
Louvre	3
Notre Dame	4
San Marco	2
Tate Modern	1
Trafalgar Square	3

This method, after applying vector quantization to reduce the dimensionality of the SIFT vectors, it runs fast, and also generates some decent results. These results are not as high for some compared to the first method, but the positive that this method brings is that the average retrieval precision is mostly more than 2, taking into account we used k = 1.

The Big Ben which had a high retrieval factor in the first part; does not have that here. Here, the average precision for Big Ben is just 1. But for the other which had a very low precision rate in

the first part, they get better with this method. At an average, if we discard the ones with precision value 1, the average precision of retrieval is almost 3, which is a still on par with the first method. On contrast, the thing to notice is that this method also runs faster compared to that. But if we take the overall average precision, the second method has an average of 2.5 compared to the first method, 3.2.

Notre Dame has the maximum retrieval rate, which means it was easy to find. Retrieving Big Ben was the most difficult of all, which was a bit contrasting compared to the first method. But since we are discarding a large number of dimensions, we think these results are pretty decent. Another thing to notice here is that the precision rate is not higher for some as in our initial method. Maximum precision value for any category in our first method is 6 (Trafalgar Square), but for the second method is 4 (Empire State and Notre Dame).

Sample output for this method is given below:

```
1       58      part1_images/empirestate_15.jpg
2       55      part1_images/trafalgarsquare_21.jpg
3       43      part1_images/empirestate_14.jpg
4       40      part1_images/sanmarco_5.jpg
5       39      part1_images/sanmarco_19.jpg
6       38      part1_images/trafalgarsquare_16.jpg
7       35      part1_images/trafalgarsquare_5.jpg
8       33      part1_images/eiffel_7.jpg
9       32      part1_images/notredame_5.jpg
10      30      part1_images/sanmarco_14.jpg
```

Part 3:

Procedure for Q3:

1. Created the homograph matrix and stored it in “matrix”
2. Took inverse of the matrix
3. Used this inverse with the images to apply warp
4. The output of warped images can be seen below. Starting from 10 images from seq1 followed by 10 images of seq2

Seq1 images: