B657 Computer Vision

Image Warping, Matching and Stitching

Assignment 2 -Report

Sumit Kumar Dey

Suhas Gulur Ramakrishna

Manikandan Murugesan

Spring 2016

Image Matching

Part 1.1 - SIFT Matching

We implemented the code which finds SIFT descriptors from two images and finds the match between the descriptors in the images and draws a line that connects the descriptors in the images.

To determine the match between the descriptors, for each descriptor in one image, we find the **Euclidean** distance with respect to all other descriptors in the other image. Then, we normalize the best match descriptor with the second best match descriptor and apply a threshold to filter false matches.

The formula we used is

$$D = L2(f1 - f2) / L2(f1 - f2')$$

f1 - Descriptor in Image 1
f2 - Descriptor in Image 2 which has the best match with f1
f2' - Descriptor in Image 2 which has the second best match with f1

If **D < 0.7**, we consider it a match. We experimented with various values to come up with a good threshold value. The correctness of the threshold was image dependent. For a few images, the threshold of 0.4 was good enough to fetch enough descriptors match with more true positives and less true negatives. While in some instances, not enough sift matches were found with 0.4. We increased the threshold to 0.9 and found that there are more false positives in this case.

After repeated trials with various values, we fixed the threshold to 0.7 which gave us a better number of descriptors with negligible false positives.

We had also experimented with Manhattan and Chamfer distance and found after repetitive trials that Euclidean with a threshold of 70 percent gave us better results.

The running time for this part to calculate Sift matches between two images : **m * n * 128 * 128**

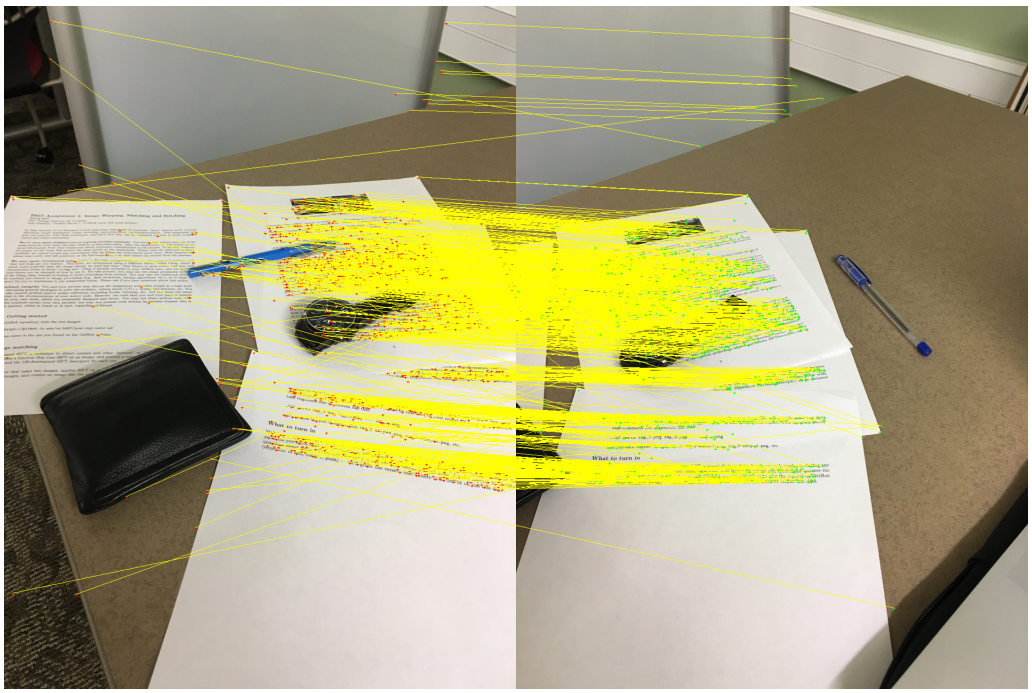
m - number of descriptor in Image 1
n - number of descriptor in Image 2
128 - Dimensions of each descriptor

To execute this part, pass the following command:

./a2 part1 <image1.png> <image2.png>

The output file is saved with the filename '**sift.png**'

A sample output for this part can be found below:



Part 1.2- Image querying

We extended this logic for the next part and ran a query search for the given query image across all the images passed. We found the number of matching descriptors for the query image and all the other images and based on that number, we output the descending order of the images.

To run this part, run the following command line arguments:

./a2 part1 <query.png> <img1.png> <img2.png>

When a similar query is run, <query.png> would be compared using sift descriptors with all the other images that follow it in the query. For each Sift matching between the query image and the <imgX.png>, we calculate the count of sift descriptor matches between them.

Based on the count, we would rank the images with respect to query image. Depending on the number of descriptor matches, the images are ranked. Only the **top 10 results** are displayed

A sample input would be:

./a2 part1 part1_images/bigben_2.jpg part1_images/colosseum_3.jpg part1_images/bigben_3.jpg part1_images/trafalgarsquare_15.jpg

The output would be the list of images sorted in descending order as follows:

```
Normal Sift Matching top results
Image = part1_images/bigben_3.jpg Count = 35
Image = part1_images/trafalgarsquare_15.jpg Count = 7
Image = part1_images/colosseum_3.jpg Count = 1
```

Part 1.3- Image Ranking

We extended the logic for the part 1.2 and made it run across all the images supplied in the sample. This outputs the ranking for each image. To run this, type in the following:

./a2 part1

We have a dataset of 10 images of 10 different locations making it 100 in total. The program takes one image randomly and then ranks it across all the images in the dataset with the image. Only the top 10 results for each image are displayed. The output would be similar to:

```
Query Image = part1_images/colosseum_4.jpg
Normal Sift Matching top 10 results
Image = part1_images/colosseum_5.jpg Count = 75
Image = part1_images/tatemodern_4.jpg Count = 11
Image = part1_images/trafalgarsquare_22.jpg Count = 8
Image = part1_images/sanmarco_3.jpg Count = 8
Image = part1_images/notredame_25.jpg Count = 6
Image = part1_images/louvre_4.jpg Count = 6
Image = part1_images/louvre_14.jpg Count = 6
Image = part1_images/colosseum_15.jpg Count = 6
Image = part1_images/colosseum_11.jpg Count = 6
Image = part1_images/bigben_3.jpg Count = 6

Query Image = part1_images/eiffel_7.jpg
Normal Sift Matching top 10 results
Image = part1_images/eiffel_19.jpg Count = 14
Image = part1_images/tatemodern_6.jpg Count = 10
Image = part1_images/tatemodern_8.jpg Count = 9
Image = part1_images/tatemodern_13.jpg Count = 9
Image = part1_images/tatemodern_11.jpg Count = 9
Image = part1_images/notredame_8.jpg Count = 9
Image = part1_images/louvre_11.jpg Count = 9
Image = part1_images/colosseum_15.jpg Count = 9
Image = part1_images/bigben_12.jpg Count = 9
Image = part1_images/tatemodern_16.jpg Count = 8

…. and so on for the rest of the images
```

The precision depended on the image and it varied across all the images. We calculated the precision for each location and the average precision for 10 trials for each location can be found as follows:

1) BigBen - 61%
2) Colosseum - 33%
3) Eiffel Tower - 25%
4) Empire State - 27%
5) London Eye - 46%
6) Louvre - 46%
7) Notre Dame - 55%
8) San Marco - 59%
9) Tate Modern - 45%
10) Trafalgar Square - 48%

Overall precision for the dataset = **44.5 %**

The easiest location to find were **BigBen** and **San Marco**
The matches that were hard to find were **Eiffel Tower** and **Empire State**.

Part 1.4- Reduced Sift Matching

Since the SIFT implementation computes the distance between the descriptors in 128 dimensions and then selects the one with minimum distance, the process can be quite slow. The run time complexity as given in Part 1.1 is m * n * 128 *128.

To improve this and to reduce time considerably, we implemented a quantized projection function[f(v)] that reduces the number of dimensions in which we compute the distance.

The steps we followed to do this are:

1) We created 'k' number of Gaussian distribution (mean = 0, sigma = 1) each of 128 dimension

2) For each of the image and each of the descriptor, we create a dot product of the 128 dimension descriptors with each of the 'k' Gaussian distribution to obtain 'k' dimension feature of the descriptor.
3) 'w' is another constant used to set the newly created descriptor value.

The run time complexity for this approach is **m * n * k * k**, where **k << 128**

This reduced the runtime extensively and it made the sift matching faster. But this is actually trade off, since the accuracy drops when k is less.

The time required for Normal SIFT matching process in Part 1.1 : **319.93** sec

The time required for Quantized projection function SIFT matching process in Part 1.1 : **89.93** sec

Overall precision for the Quantized projection function SIFT matching: **24**%
To run across specific image :
./a2 part1fast <query.png> <img1.png>

This finds the matching descriptors between the two images and marks them.
This will save the sift matches under the file 'newsift.png'

A sample output is given as follows:

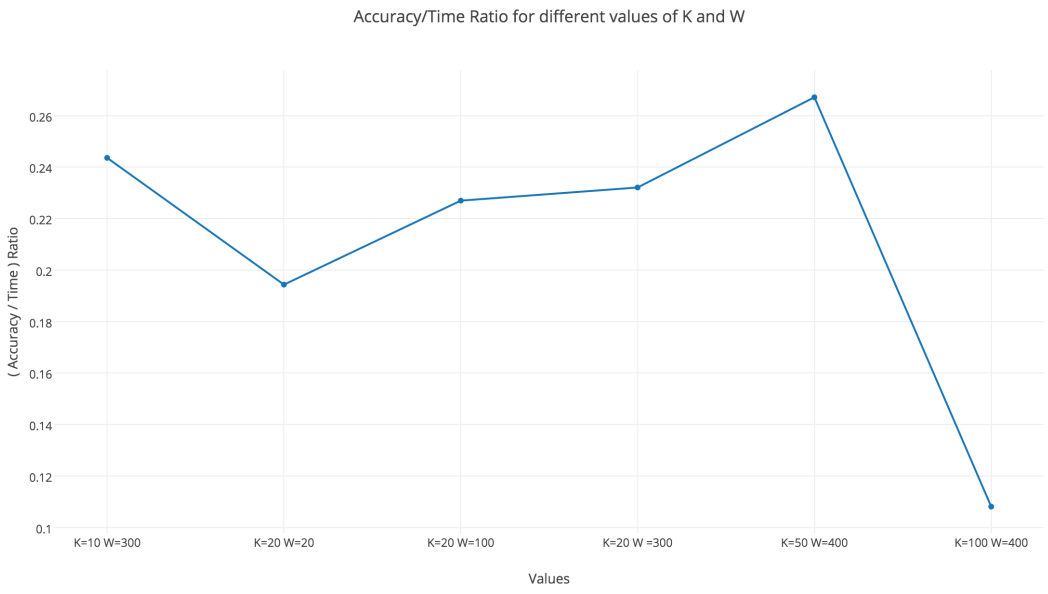
To run across an image set:
./a2 part1fast <query.png> <img1.png> <img2.png> ….. <imgX.png>

This sorts the images based on the number of matching descriptors and displays them to the user.

To run across all sample images :
./a2 part1fast
This ranks all the images with randomly selected query image similar to Part 1.3 and displays the top 10 results

Difficulties:
We had to repeat the process for several values for w and k to come up with the better estimate. As k increases, accuracy also increases since the number of dimensions in which we consider the descriptors also increases. But this in turn slows down the process as the number of computation increases.

We tried by repeating the number of trials by producing different sets of random Gaussian distributions and the result was a little better though the time consumption was also more.
The following graph depicts the (accuracy/time) plotted in the graph for the values we got after repeated trials:



The above graph shows that, the better yield is gotten for values **k = 50 and w = 400**, since that gave us the best (accuracy/time) ratio.

Image Warping and Homographies

Part 2.1 - Image Transformation

Our code takes in the given image and the corresponding transformation matrix. Instead of the straight forward projection approach, we compute the inverse of the transformation matrix and did **Bilinear interpolation**[1] against each pixel in the image to transform it into the new image.

We used the CImg library function to invert the 3x3 projection matrix.

To execute this part:
./a2 part2 lincoln.png

The output is saved under the file '**lincoln_transform.png**'

The output when this component is run against the given image is found as follows:

Part 2.2 - Projection Transformation Estimation

We did projective transformation with the given transformation matrix. But to make it general and to make this run for all the images, the program has to estimate the transformation matrix. So we wrote the function which estimates the homography on its own and applies that transformation to the given image. We did this using **RANSAC** to solve the problem.

The steps we followed are:

1) Finding the SIFT matches between the image and it's projected image. If the number of SIFT matches we find are less than 4, we terminate the process since a minimum of 4 points are required.
2) Collect all the corresponding pairs of sift matches between the image and it's projected image.
3) We repeat the same for **N = 1000** times
    3.1) Pick 4 points at random from all the matches.
    3.2) Fill the points according to the lecture 9 slide 40. Let A be the 8x8 matrix and B is the 8x1 matrix which contains all the points from the transformed image.
    3.3) We used the linear system solver in the CImg to solve the system of linear equation to find the final 8x1 projection matrix.
    3.4) We check the number of inliers for the projection matrix.
    3.5) When the projection has more inliers compared to the existing best value, we save it as the best projection.

4) We use the best projection obtained from the above step and the actual image to get the projected image by calling the code written for Part 2.1

Note: Inliers are calculated by running through all the SIFT descriptor pair matches and checking the points satisfies the projection matrix.
A threshold boundary is kept for the transformed image point to match so say it as an inlier.

Part 2.3 - Image Warping

This part finds the warping of the query image similar to the projected image given. We arrive at the expected result by doing the following steps:

1) For each image in projected image [i .. N]
2) Create a name called mg_i-warped.jpg
3) Call the function in part 2.2 which will find the projection between the query image and the given image and warp the query image to that of the projection.

To execute this step:
./a2 part2 <query.png> <img1.png> <img2.png> …. <imgX.png>

The output would be saved under:
img_1-warped.png, img_2-warped.png and so on.

Sample output is as follows:

Difficulties:

The warping completely depends on the quality of the SIFT matches between the query image and the transformed image. If the sift matching is poor, then the transformation will be not accurate. After repeating the testing process for across a set of images, we found that it works only when we get good SIFT descriptor matches.

Note:

1) According to our folder structure, Part1_images can be accessed by /part1_images/
2) Part2_images can be accessed by /part2_images/seq1/ and /part2_images/seq1/

REFERENCES:

1 - https://en.wikipedia.org/wiki/Bilinear_interpolation