CS B657 Spring 2016

Assignment 2 Report

Usernames:

1. Debasis Dwivedy (ddwivedy)
2. Srivatsan Iyer (srriyer)
3. Akash Ram Gopal (agopal)

Brief description of how our code works

The code follows below sequence –

Part 1.1: run code as:

./a2 sift image.jpg image2.jpg

1. In our implementation of SIFT, we take 2 input images, the part of the question to execute. We take the part of the question as input because part 1.1 requires us to form a resulting image matching the sift points.
2. First, we take the two input images and convert it into grayscale images if not already in grayscale.
3. Then, we call the given compute_sift method to obtain the descriptors of each of the image.
4. We create a merged output image since we need it to visualize the matching sift points on the two images.
5. We use the formula to compute the distance between the two images using the descriptors of the two images. This is nothing but the ratio of the Euclidean distances between the closest match and the second-closest match and if the resultant distance ratio is less than the magic number of the dist_ratio which is found to be 0.8 after due experimentation, we draw a line on the output merged image.
6. The output of this part this then written to a file named “sift.png”.
7. This image is also displayed to the user. Use this to check the correspondences detected by Sift Matcher.

Part 1.2, Part 1.3: run code as:
./a2 part1 image.jpg image2.jpg image3.jpg, …

Comparison between two images

1. In our implementation of SIFT, we take 2 input images, the part of the question to execute. We take the part of the question as input because part 1.1 requires us to form a resulting image matching the sift points.
2. First, we take the two input images and convert it into grayscale images if not already in grayscale.

3. Then, we call the given compute_sift method to obtain the descriptors of each of the image.
4. We create a merged output image since we need it to visualize the matching sift points on the two images.
5. We use the formula to compute the distance between the two images using the descriptors of the two images. This is nothing but the ratio of the Euclidean distances between the closest match and the second-closest match and if the resultant distance ratio is less than the magic number of the dist_ratio which is found to be 0.8 after due experimentation, we draw a line on the output merged image.
6. The output of this part this then written to a file named "sift.png".

Running above algorithm against multiple images.

1. Given to us are 10 images from 10 well known tourist attractions for a total of 100 images. Therefore, first we stored all these images in the form of a vector called imageNames and sorted the vector so that the group of these 10 images are together.
2. Then, we randomly picked one image from each of these groups and stored it in another vector called queryImages.
3. Now, for each of the images in the vector queryImages, we apply SIFT with every other image including itself and store the results in the vector of class which we created on part 1.2.
4. We then sort these vectors based on their matches, and print out the 10 best for the particular query image. We observed that this process takes an average of 450-500 seconds to execute. We repeat this for every query image.
5. We calculate the precision by looking at the number of test images we obtained in the top 10 that came from the same group of the tourist attraction.
6. The precision obtained is as shown below and a more detailed output is in the text file called **output_queryimage.txt**.

Execution Command:
./a2 part1 a2-images/part1_images/queryImage.jpg a2-images/part1_images/*.jpg | tee output-tatemodern_14.txt

From the table below, we can see that Notredame and Tatemodern is the easiest to recognize whereas Sanmarco is the toughest.

Query Image	Precision(%)
Bigben_10.jpg	40
Colosseum_8.jpg	20
Eiffel_3.jpg	30
Empirestate_16.jpg	20
Londoneye_12.jpg	40

Louvre_15.jpg	20
Notredame_3.jpg	70
Sanmarco_19.jpg	10
Tatemodern_14.jpg	50
Trafalgarsqure_16.jpg	40

Part 1.4: run code as
./a2 part1Fast a2-images/part1_images/tatemodern_14.jpg a2-images/part1_images/*.jpg | tee output-tatemodern_14.txt

1. Since the previous implementation was taking a higher amount of time because of the process of finding the nearest neighbors in the a huge set of higher dimensional points.
2. We follow the exact process as SiftMatcher, but instead of comparing all pairs of two sets of descriptors, we try to reduce the number of comparison by comparing only against a tiny subset of descriptor.
3. The projection function which performs the following:
$$f_i(v) = [x^i.v/w]$$
gives us a way of approximately comparing the descriptors in lower dimensions.
4. We calculate the projection value for the first image. This results in a k-vector. Similarly, for the second image, we have another k-vector. Ideally we would be checking for a complete equality of k-vector, but we felt that having an approximate way would be far better. The Euclidean distance threshold is represented as "max_dist".
5. From the subset returned, we perform full matching in 128 dimension to return results.

Through the course of experimentation, we have found that the algorithm works best with: k=10, w=255, max_dist=5
Below are the results

Query Image	Precision(%)	Fast Match Precision(%)
Bigben_10.jpg	40	40
Colosseum_8.jpg	20	20
Eiffel_3.jpg	30	30
Empirestate_16.jpg	20	20
Londoneye_12.jpg	40	40
Louvre_15.jpg	20	20

Notredame_3.jpg	70	70
Sanmarco_19.jpg	10	10
Tatemodern_14.jpg	50	50
Trafalgarsqure_16.jpg	40	40

Part 2.3: How to run the code:

./a2 part2 a2-images/part2_images/seq2/445303794_898b486744_z_d.jpg a2-images/part2_images/seq2/*

1. We take one initial image, and pair it up with every other image. We try to compute the sift descriptors and then finally compute the list correspondences (x1, y1) => (x2, y2).
2. We use the correspondences in RANSAC algorithm and come up with a set of models. For each of the models, we calculate the projection matrix by solving 8 simultaneous equations and check how many of (n-4) correspondences agree with the model. We pick the matrix formed out of the model that has the largest inlier count.
3. For each of the pair, the program outputs an image called: "<filename>-warped.png". This image represents the warped image that has been transformed to match the perspective in the inital image.

Challenges faced:

1. Set of 4 points chosen were too close for a real equation to be formed. We calculated the condition number of the linear equation and randomly sampled it until the value was less than 10
2. The transformation matrix used in inverse warping was producing jagged images. We used linear interpolation for inverse warping. This produced better looking images.
3. We tried randomly setting the the number of trials. To make it more robust, we plugged it into the expression: $1-(1-(1-e)^s)^N \geq p$, where $p = 0.99$, $s = 4$, $e = 0.5$.

References:

1. https://www.youtube.com/watch?v=NPcMS49V5hg
2. www.inf.fu-berlin.de/lehre/SS09/CV/uebungen/uebung09/SIFT.pdf
3. https://www.cs.cornell.edu/courses/cs664/2008sp/handouts/cs664-6-features.pdf
4. http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/AV0405/MURRAY/SIFT.html