Assignment 2

CSCI B657 – Computer Vision

dipiband-marshalp-sriksrin

Marshal Patel
Srikanth Srinivas Holavanahalli
Dipika Bandekar

1. FILES SUBMITTED:

SR. NO	FILE NAME	INCLUDED/MODIFIED
1	a2.cpp	MODIFIED
2	Makefile	MODIFIED
3	ransac.h	INCLUDED
4	ransac.cpp	INCLUDED
5	Siftmatching.h	INCLUDED
6	Siftmatching.cpp	INCLUDED
7	part_1.txt	INCLUDED

- a2.cpp – Contains the code and core logic for Part 1 and Part 2 Questions of the assignment
- Part_1.txt – Contains image names of the 100 test images to be used in section 2
- Siftmatching.cpp – core functions that are used in the main code like sift matching, Euclidean, k-dimensional sift matching
- Ransac.cpp – Includes functions definitions for executing part 2

2. HOW TO RUN OUR CODE?

1. PART 1:

- **SECTION 1**

1. **(slow)** ./a2 part1 input_image_1 input_image_2
2. **(fast)** ./a2 part1fast input_image_1 input_image_2

- **SECTION 2**

1. **(slow)** ./a2 part1 query_image input_image_1 input_image_2 ... input_image_n
2. **(fast)** ./a2 part1fast query_image input_image_1 input_image_2 ... input_image_n

- **SECTION 3**

1. **(slow**) ./a2 part1 query_image part_1.txt

2. **(fast)** ./a2 part1fast query_image part_1.txt

- **Note**: All 100 images need to be present in the directory where the code is being executed

2. PART 2:

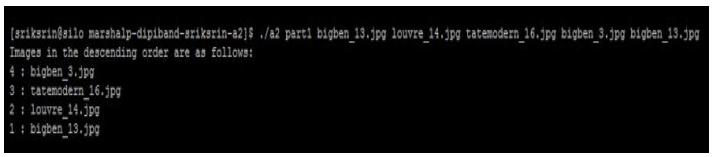
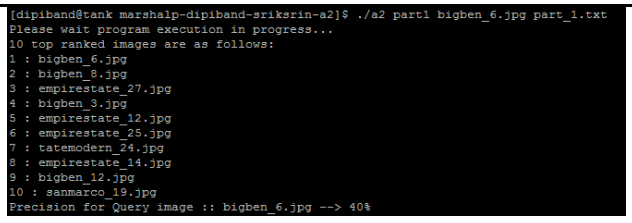
- **SECTION 1**

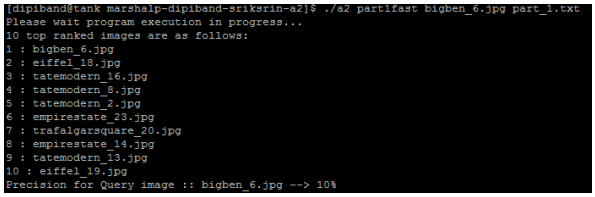
1. ./a2 part2 input_image

- **SECTION 2 & 3**

1. ./a2 part2 query_image input_image_1 input_image_2 input_image_3 … input_image_n

3. OUTPUT FILES GENERATED:

SR. NO	PART – SUBSECTION	OUTPUT
1	Part 1 ---> Section 1 ---> **SLOW**	sift_matching_slow.png sift1.png sift2.png
2	Part 1 ---> Section 1 ---> **FAST**	sift_matching_fast.png sift1.png sift2.png
3	Part 1 ---> Section 2 ---> **SLOW**	
4	Part 1 ---> Section 2 ---> **FAST**	
5	Part 1 ---> Section 3 ---> **SLOW**	

6	Part 1 ---> Section 4 ---> **FAST**	
7	Part 2 ---> Section 1	**img_1-warped.png**
8	Part 2---> Section 2 & 3	**img 2-warped.png, img 3-warped.png, etc.**

4. PART 1 ANALYSIS:

2.1: SIFT MATCHING SLOW:

Function Used: (Section 1, 2 and 3)

1. sift_matching(vector<SiftDescriptor>&, vector<SiftDescriptor>&, string, string, int)
2. euclidean_sift(SiftDescriptor, SiftDescriptor)

Following is the brief outline of the algorithm

a. Sift Matching

- Step 0: sift descriptors of the query image and test image are calculated
- Step 1: for each descriptor in query image, Euclidean distance with descriptors of the test image is calculated
- Step 2: the least and second least distance for each descriptor is calculated and if the ratio of first best and second best is less than 0.75, that particular descriptor is a good match. Else, the descriptor is discarded.
- Referred from the website http://cs.brown.edu/courses/cs143/results/proj2/sz22/ (0.75 → value)
- Step 3: For all the matched descriptors X_0, Y_0 in query image and X_1, Y_1, a line is plotted with a predefined color.
- Below is one of the outputs obtained in our experiments.

SECTION 1: Images (bigben_13 & bigben_16)

SECTION 2:

- Step 0: sift descriptors of the query image and all test images are calculated
- Step 1: for each descriptor in query image, Euclidean distances with descriptors of the test images are calculated
- Step 2: the least and second least distance for each descriptor are calculated and if the ratio of first best and second best is less than 0.75, that particular descriptor is a good match. Else, the descriptor is discarded.
- Step 3: The test image with least sum of Euclidean distance would be ranked 1, second least with rank 2 and so on.
- Below is the screenshot of one the outputs of our experiment

SECTION 2: Descending order

```
[sriksrin@silo marshalp-dipiband-sriksrin-a2]$ ./a2 part1 bigben_13.jpg louvre_14.jpg tatemodern_16.jpg bigben_3.jpg bigben_13.jpg
Images in the descending order are as follows:
4 : bigben_3.jpg
3 : tatemodern_16.jpg
2 : louvre_14.jpg
1 : bigben_13.jpg
```

SECTION 3

- Step 0: sift descriptors of the query image and all 100 test images are calculated
- Step 1: for each descriptor in query image, Euclidean distances with descriptors of the test images are calculated

- Step 2: the least and second least distance for each descriptor is calculated and if the ratio of first best and second best is less than 0.75, that particular descriptor is a good match. Else, the descriptor is discarded.
- Step 3: The test image with least sum of Euclidean distance would be ranked 1, second least with rank 2 and so on.
- Step 4: The top 10 matching images are ordered with least distance at the top of the list.
- Below is the screenshot of one the outputs of our experiment

SECTION 3: Images (bigben_6 compared with all 100 images)

```
[dipiband@tank marshalp-dipiband-sriksrin-a2]$ ./a2 part1 bigben_6.jpg part_1.txt
Please wait program execution in progress...
10 top ranked images are as follows:
1 : bigben_6.jpg
2 : bigben_8.jpg
3 : empirestate_27.jpg
4 : bigben_3.jpg
5 : empirestate_12.jpg
6 : empirestate_25.jpg
7 : tatemodern_24.jpg
8 : empirestate_14.jpg
9 : bigben_12.jpg
10 : sanmarco_19.jpg
Precision for Query image :: bigben_6.jpg --> 40%
```

2.1: SIFT MATCHING FAST:

Function Used: (Section 1, 2 and 3)

1. vector<pair<int, int> > k_dimensional_sift(const vector<vector<float> >&, const vector<vector<float> >&); // to perform sift on k-dimensional vectors
2. float k_dim_sift_matching(vector<SiftDescriptor>&, vector<SiftDescriptor>&, vector<pair<int, int> >, string, string, int); // to perform sift on specific descriptors
3. euclidean_vec(vector<float>, vector<float>)

- Step 0: sift descriptors of the query image and all 100 test images are calculated
- Step 1: Uniform distribution of 128 numbers between 0 and 1 are generated by following formula,

  A[i] = (i-min)/128 ; min = 0, quantizing the range into 128 bins of equal frequency

  (We also checked with randomly generated values between 0 and 100 and multiplied it with .01)
- Step 2: 128D vector of numbers between 0.0 and 1.0 are picked from an uniform distribution
- Step 3: A dot product of the 128D vector from step 2 and the descriptor of image is calculated, divided by a value ‘w’ and this step is repeated for k times
- Step 4: a summary vector of k dimensions for a 128D is generated is generated , this is done for all descriptors of all descriptors of the 100 images
- Step 5: for each k dimensional descriptor in query image, Euclidean distances with descriptors of the test images are calculated

- Step 6: the matched descriptors' descriptor numbers are stored and Euclidean distance of these descriptors of original 128D images of query and test images are calculated to check how similar the images are
- Step 7: Top 10 similar images are chosen and listed in increasing order
- Below is the screenshot of one the outputs of our experiment

SECTION 1: Images (bigben_13 & bigben_16)
(Unique key descriptor point detected)

Section 2 Descending fast

```
[sriksrin@silo marshalp-dipiband-sriksrin-a2]$ ./a2 part1fast bigben_13.jpg louvre_14.jpg tatemodern_16.jpg bigben_3.jpg bigben_13.jpg
Images in the descending order are as follows:
4 : tatemodern_16.jpg
3 : louvre_14.jpg
2 : bigben_3.jpg
1 : bigben_13.jpg
[sriksrin@silo marshalp-dipiband-sriksrin-a2]$
```

```
[sriksrin@silo marshalp-dipiband-sriksrin-a2]$ ./a2 part1fast bigben_6.jpg part_1.txt
10 top ranked images are as follows:
1 : bigben_6.jpg
2 : tatemodern_11.jpg
3 : bigben_12.jpg
4 : bigben_16.jpg
5 : colosseum_13.jpg
6 : notredame_5.jpg
7 : londoneye_23.jpg
8 : bigben_10.jpg
9 : sanmarco_1.jpg
10 : notredame_20.jpg
Precision for Query image :: bigben_6.jpg --> 40%
[sriksrin@silo marshalp-dipiband-sriksrin-a2]$
```

TIME Analysis

- Algorithm speeds up almost 1.5 times as compared to implementation in question 3.
  Time to run part 1 Slow Section 1: **1.6 s**
  Time to run part1fast Section 1: **1.3 s**
  Time to run part 1 Slow Section 3: **119.9 s**
  Time to run part 1 fast section 3: **84.6 s**

K and w consideration and finalizing k = 5 and w = 128

We checked for multiple images and got unique matching descriptors for k = 5 and k =128

Here are few results:

1. K = 5 and w = 16:

K = 5 and w = 64

K = 5 and w = 128

5. PART 2 ANALYSIS:

2.1: Inverse Warping:

Function Used: inverse_warping(input_image, homography matrix, warp_id)

Following is the brief outline of the algorithm

- Step 1: Inverse of homography matrix is computed.
- Step 2: for every point in destination image, a corresponding homogenous co-ordinate in the source image is obtained.
- Step 3: Divide the two co-ordinates by the third coordinate to obtain the Cartesian co-ordinates in the source image; since the co-ordinates obtained are not integers, they are rounded to closest integers.
- Step 4: RGB values of the source pixel are copied to the destination pixel.

Output: warped image: **img_1-warped.png**

2.2: Ransac:

Function Used: perform_ransac (input_image, new_image, warp_id)

Following is the brief outline of the algorithm

- Step 1: A vector of matching pair descriptors is obtained
- Step 2: Randomly select any 4 pairs from the matching descriptor vector.
- Step 3: Matrices A (8 x 8) and B (8 x 1) are filled with values of descriptor co-ordinates respectively.
- Step 4: Values of matrix H (8 x 1) are solved and stored in a 3 x 3 matrix. The value at position (2, 2) in the resultant matrix is 1.
- Step 5: Using the H matrix, count the number of inliers by running it through the vector of matched descriptors. If the x and y co-ordinates obtained is less than 10 then we consider it an inlier.
- Step 6: Repeat steps 2 through 5 for N number of times where N = 1000. We chose N=1000 as it was giving the best results out of various values tried (N=200, 500).
- Step 7: The H matrix is chosen as the final homography matrix which gives the maximum number of inliers.
- Step 8: Inverse_warping is invoked and the warped image is obtained.

2.3: Image Sequence warping application:

Following is the brief outline

- Multiple images passed and compared with the first image of the command line which generates sequences of warped images

OUTPUT ANALYSIS FOR PART 2 SECTION 2 & 3:

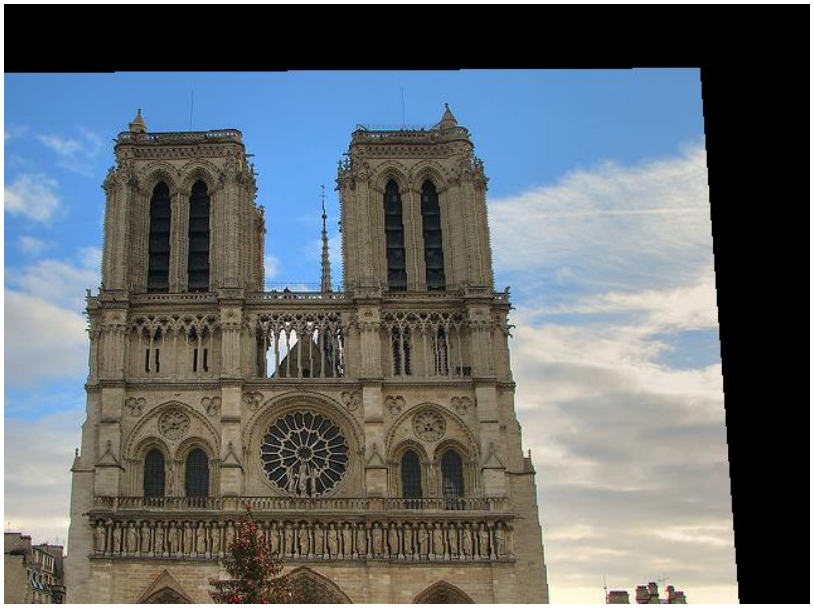
- **Part2_images/seq1**

Query image: 2nd image given for testing

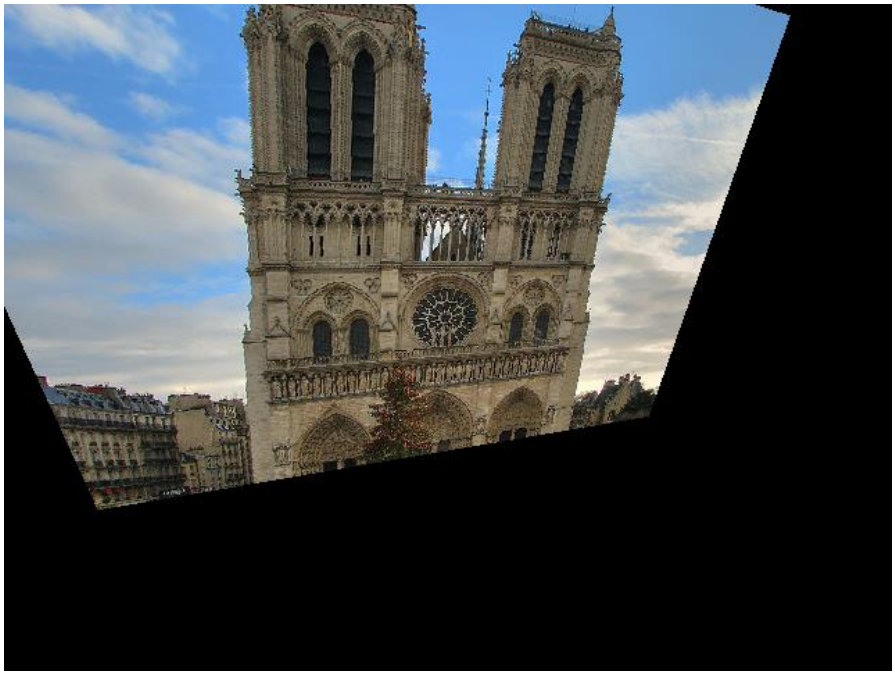
Before Warping

1. Loop (n = 500)

Effect on 3rd image given for testing purposes



2. **Loop (n = 1000)**



3. **Loop (n = 200)**