Assignment 2
Image Warping, Matching and Stitching

Snehil Vishwakarma
(snehvish)

Repository:

https://github.iu.edu/cs-b657-sp2017/snehvish-a2

Implementation Details:

All implementation of this assignment is in “a2.cpp” which uses primarily “Sift” and “CImg” apart from the standard libraries

Scripts:

To run all parts sequentially, just use > ./makenrun.sh
1) To just run part1 (Sift Descriptot & Sift Matching): > ./makenrun_part1.sh
2) To just run part2 (Fast Sift Matching & RANSAC): > ./makenrun_part2.sh
3) To just run part3 (Inverse Warping): > ./makenrun_part3.sh

Executable Files:

make clean
make

1) To run part1:
   ./a2 part1 inputimage … **OR** ./a2 part1 inputimage queryimage1 …
2) To run part2:
   ./a2 part1 inputimage queryimage1 …
3) To run part3:
   ./a2 part3 a2-images/lincoln.png **OR** ./a2 part3 inputimage queryimage1 …

Custom Structures

1) K-D Tree structure for Sift Descriptors

```
struct kdtree
{
    int val, index;
    kdtree *l, *r;
    vector < SiftDescriptor > descriptors;
};
```

2) Structure to store matching input image's sift descriptors corresponding to query image's sift descriptors

```
struct ans
{
   SiftDescriptor in_match;
   double val;
   double mined, min2ed;
   bool ismatch;
};
```

3) Structure to hold the information about the top 10 ranked images from the set of tested 100 images for all attractions

```
struct list
{
   string fname;
   int ct_match, ct_descriptors;
   float avg;
};
```

Custom Functions

1) Function to generate a k-d tree from given input image's Sift Descriptors

   *void gen_kdtree(vector < SiftDescriptor >, kdtree **, int, int);*

2) Function to print the k-d tree in in-order format

   *void print_kdtree(kdtree **);*

3) Function to find the euclidean distance between the 128-descriptors of the two Sift descriptors (of input image and query image)

   double euclidean_dist(SiftDescriptor &, SiftDescriptor &);

4) Function to search for the closest and the 2nd closest sift descriptor according to the values of the 128-descriptors

   *void search_kdtree(kdtree **, SiftDescriptor, double &, double, SiftDescriptor &);*

5) Function to find all the matching input image's sift descriptors to a query image's sift descriptor under the threshold value and save them in a vector of struct "ans" objects and also finally return the count of matching sift descriptors

   *int check_out_image(kdtree **, vector < SiftDescriptor >, double, vector < ans > &);*

6) Takes the input image and query image, stitches them together and marks all the matching descriptors and gives us the final image at the output path

   void feature_mark_append(CImg<double> , CImg<double> &, CImg<double> &, *vector <ans> &, vector < SiftDescriptor > &);*

7) Matrix Multiplication for 2 CImg<double> objects

   CImg<double> matrixmult(CImg<double> , CImg<double>);

8) Function to perform inverse warping, on a defined homographic matrix just for the image “a2-images/lincoln.png”. This function is just to warp the “lincoln.png”

   void warping(CImg<double> ,string);

9) Function to generate a k-sized vector of 128 randomized values each, between (0,1)

   vector < vector <double> > generaterandomvector(int);

10) Generating k-length descriptor for each of given image’s Sift Descriptors

    vector < vector <float> > k_descriptorgenerator(vector < SiftDescriptor > &, int, int, vector < vector <double> >);

PART 1 - STEPS

1) *“CImg<double>.get_RGBtoHSI().get_channel(2)”*
   “Sift::compute_sift(CImg<double>.)”

   First we converted input image to gray scale image and computed all the descriptors using the function “compute_sift” of the Sift library.

2) *“CImg<double>.get_RGBtoHSI().get_channel(2)”*
   “Sift::compute_sift(CImg<double>.)”

   Then we computed descriptors in the same way for all the given query images.

3) *“void gen_kdtree(vector < SiftDescriptor >, kdtree **, int, int)”*

   We generate a k-d tree for the input image, because searching in the k-d tree reduces the comparisons to a great extent.

4) Then we compare all the descriptors of every query image to find their corresponding matching descriptors in the input image.

5) *void search_kdtree(kdtree **, SiftDescriptor, double &, double, SiftDescriptor &);*
*int check_out_image(kdtree **, vector < SiftDescriptor >, double, vector < ans > &);*

   To compare we find the query image's closest and 2nd closest sift descriptor in the input image. Then we divide closest distance by 2nd closest and compare it to a thresholding value. It's just a more optimized method rather than just thresholding the closest match, because the current sift descriptor might match to some irrelevant sift descriptor in many cases. So by finding closest and 2nd closest and if they are close by (i.e. less than thresholding value), then we can say that it is a correct match.

6) void feature_mark_append(CImg<double> , CImg<double> &, CImg<double> &, *vector <ans> &, vector < SiftDescriptor > &);*

   We take a count of all the matched descriptors and stitch the input image and the query image and save it in "output-a2-images/part1_images/"

7) While checking every image, we save the top 10 most matching query images to the input image, to find the precision of the system

PART 1 - RESULT (SIFT MATCHING) {IMAGES AT THE END}

1) The best precision which we achieved is 50% (for bigben, notredame, sanmarco, and trafalgarsquare) and the worst being 10% (for louvre)

2) The time taken by user defined operations ranges from (2min 12secs to 3min 2secs) whereas the time taken by actual processing is (2sec 800msecs to 3secs) for one input image in group of 10 attractions

Input Image	Precision
bigben_2.jpg	50%
colosseum_13.jpg	20%
eiffel_6.jpg	40%
empirestate_23.jpg	10%
londoneye_13.jpg	20%
louvre_8.jpg	10%
notredame_1.jpg	50%

Input Image	Precision
sanmarco_3.jpg	50%
tatemodern_8.jpg	40%
trafalgarsquare_15.jpg	50%

PART 2 - STEPS

1) *"CImg<double>.get_RGBtoHSI().get_channel(2)"*
"Sift::compute_sift(CImg<double>.)"

   First we converted input image to gray scale image and computed all the descriptors using the function "compute_sift" of the Sift library.

2) *"CImg<double>.get_RGBtoHSI().get_channel(2)"*
"Sift::compute_sift(CImg<double>.)"

   Then we computed descriptors in the same way for all the given query images.

3) "*vector < vector <double> > generaterandomvector(int);"*

   Then we generate a random distributed (between 0.0 and 1.0) k 128-dimension vector

4) "*vector < vector <float> > k_descriptorgenerator(vector < SiftDescriptor > &, int, int, vector < vector <double> >);"*

   We try to concise the information of the 128-descriptors to k-length descriptors for reducing descriptor dimensionality for implementing fast Sift Matching.
   To reduce 128-dimensions to k-dimensions we use the given projection function:

$$f_i(v) = \text{floor}\ (\ x^i \cdot v\ /\ w)$$

   where x -> randomly generated vector in the former function
   v -> Sift Descriptor's 128-dimension descriptor
   i -> ranges from 0 to k
   k -> number of dimensions to be reduced in (custom value) = 10
   w -> bin value (custom value) = 250

5) Now we can easily compare these vectors for an equality to see for a probable descriptor match. This reduces every Sift descriptor comparison from 128 to k (for us being 10 a good enough number)

6) Once we find a probable match between the 2 k-dimension descriptors we save the actual matching input image's sift descriptors in a vector of "ans" objects corresponding to all query image's sift descriptors

7) Then we compute the homography matrix using RANSAC method. We are taking 400 random 4 - query image's sift descriptors and finding the homography matrix with the maximum inliers and saving that matrix to use it finally

8) Once we have the best possible homography matrix, we try to view the input image from query image's plane (ie. forward wrapping of input image to query image's plane). Then we find the distance between new co-ordinates of the input image descriptors and the existing query image descriptors, and if it is less than a particular thresholding value (trial and tested to find an apt value of 25) we can say to a little bit more precise extent that it is an inlier, thus a probable match.

9) void feature_mark_append(CImg<double> , CImg<double> &, CImg<double> &, *vector <ans> &, vector < SiftDescriptor > &);*

   Now with the new inliers(our new matched descriptors) we take a count of them and stitch the input image and the query image and save it in "output-a2-images/ part1_images/"

10) While checking every image, we save the top 10 most matching query images to the input image, to find the precision of the system

PART 2 - RESULT (FAST SIFT MATCHING & RANSAC) {IMAGES AT THE END}

1) The best precision which we achieved is 60% (for bigben) and the worst being 10% (for empirestate, louvre, sanmarco, and tatemodern)

2) The time taken by user defined operations ranges from (0min 54secs to 1min 51 secs) whereas the time taken by actual processing is (1sec 400msecs to 1sec 509msecs) for one input image in group of 10 attractions

3) The time difference is due to the fast SIFT matching, whereas the high precision is due to RANSAC, as it takes out the outliers easily in images like "bigben", but for some highly varied image groups it cannot find the best homographic plane with maximum inliers and that is why it fails in those cases.

Input Image	Fast Matching Sift & RANSAC Precision
bigben_8.jpg	60%
colosseum_4.jpg	40%

Input Image	Fast Matching Sift & RANSAC Precision
eiffel_7.jpg	20%
empirestate_27.jpg	10%
londoneye_13.jpg	30%
louvre_16.jpg	10%
notredame_14.jpg	30%
sanmarco_14.jpg	10%
tatemodern_6.jpg	10%
trafalgarsquare_15.jpg	30%

PART 3 - STEPS

lincoln.png

1) "*void warping(CImg<double> ,string);*"
   Call the warping function on lincoln.png. Function performs inverse warping, on a defined homographic matrix just for the image "a2-images/lincoln.png".
   It saves it to the file "lincoln-warped.png" after warping according the given homographic matrix

./a2 part3 inputimage queryimage1 …

1) *"CImg<double>.get_RGBtoHSI().get_channel(2)"*
   "Sift::compute_sift(CImg<double>.)"

   First we converted input image to gray scale image and computed all the descriptors using the function "compute_sift" of the Sift library

2) *"CImg<double>.get_RGBtoHSI().get_channel(2)"*
   "Sift::compute_sift(CImg<double>.)"

   Then we computed descriptors in the same way for all the given query images.

3) Then we compute the homography matrix using RANSAC method. We are taking 500 random 4 - query image's sift descriptors and finding the homography matrix

with the maximum inliers and saving that matrix to use it finally

4) Once we have the best possible homography matrix, we wrap it to view the query image from input image's plane (ie. inverse wrapping of query image to input image's plane which is the first camera's coordinate system).

5) Then we save this warped query images (from query image's plane to input image's plane) and save it as "queryimage-warped.png"

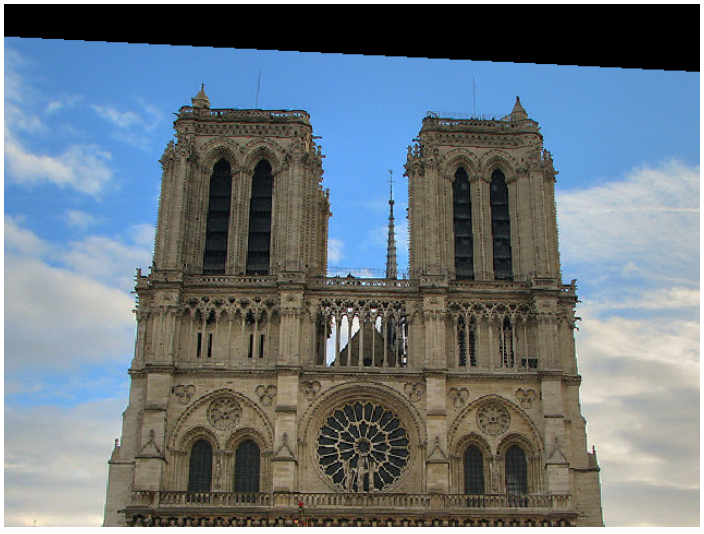
PART 3 - RESULT

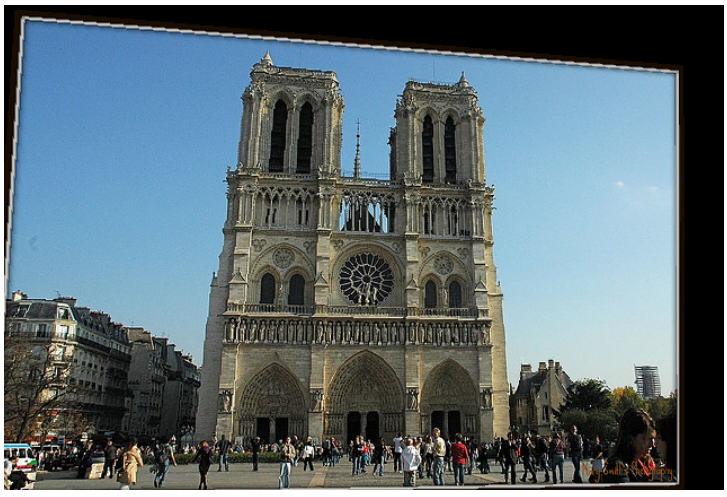
lincoln-warped.png

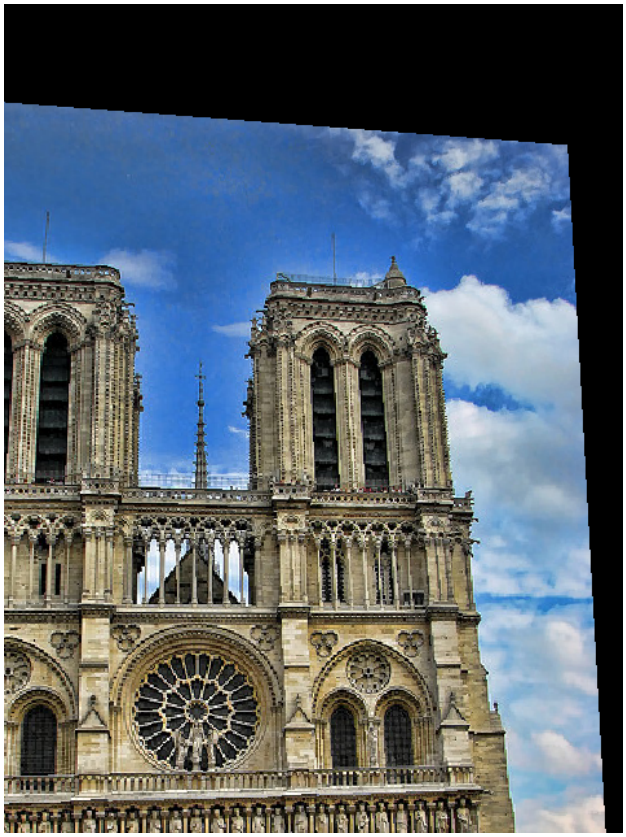
./a2 part3 inputimage queryimage1 …

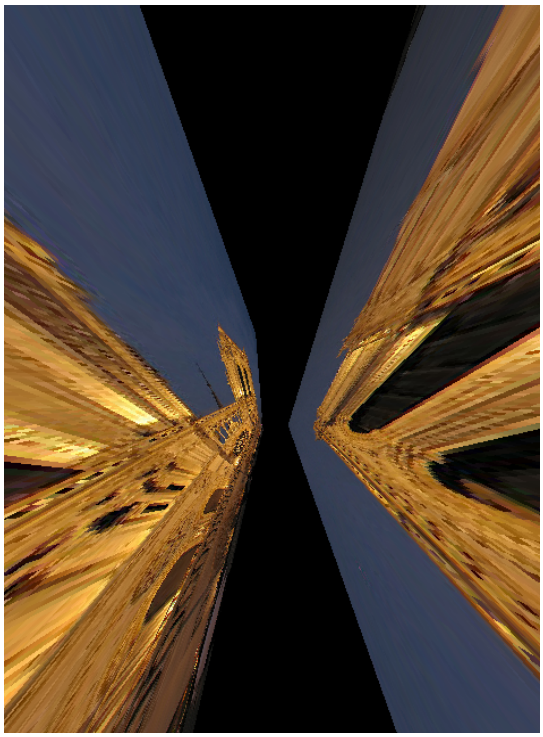
QUERY IMAGE

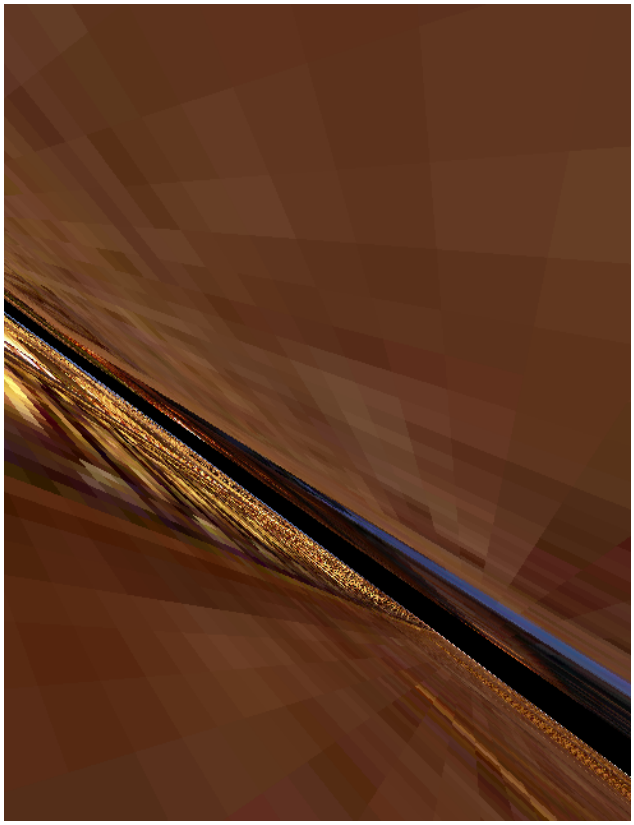
WARPED IMAGES

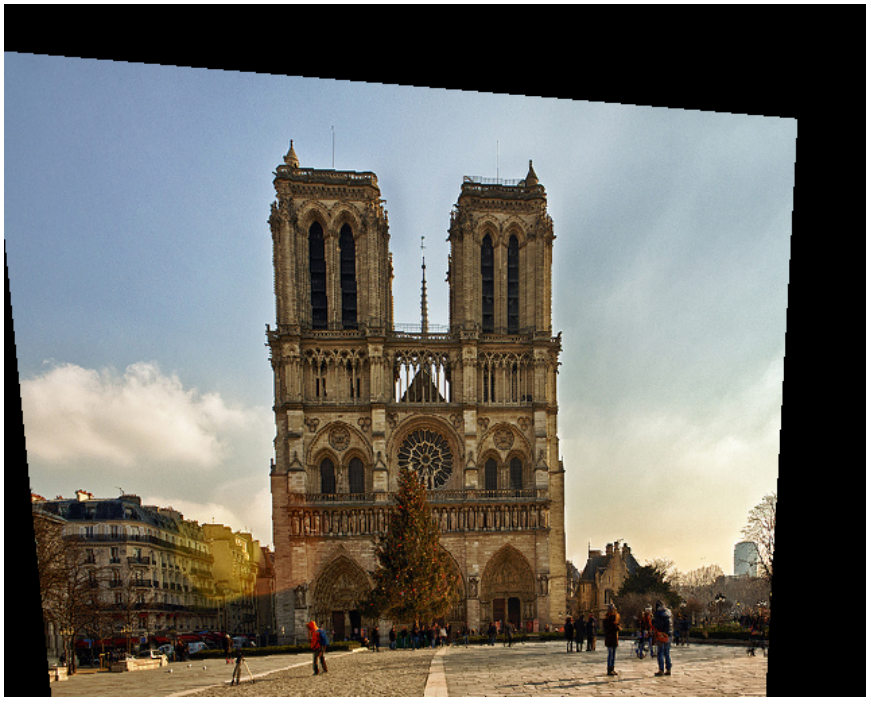


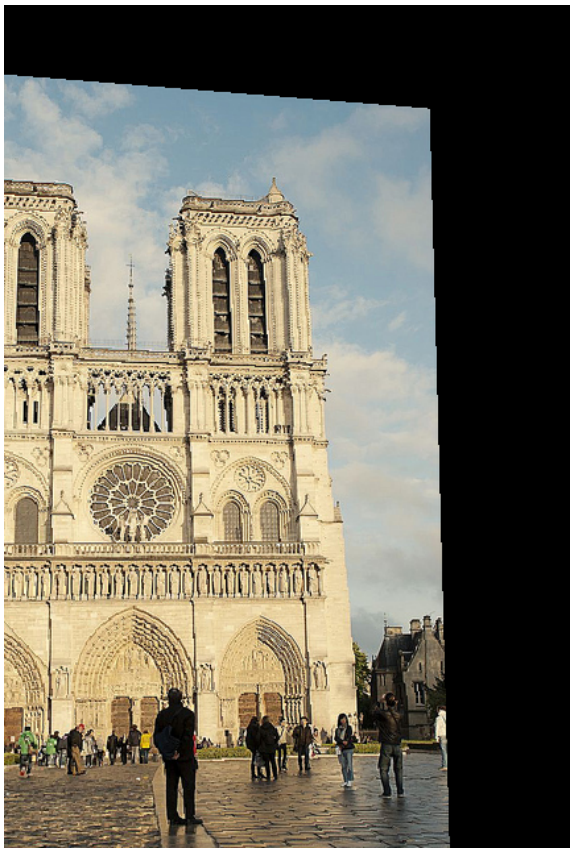








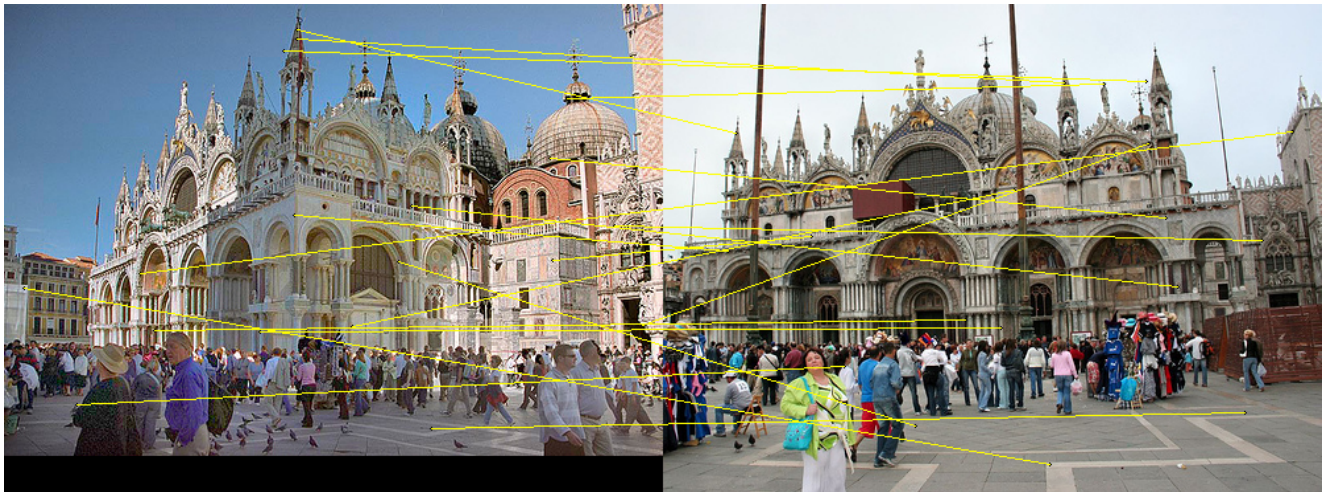


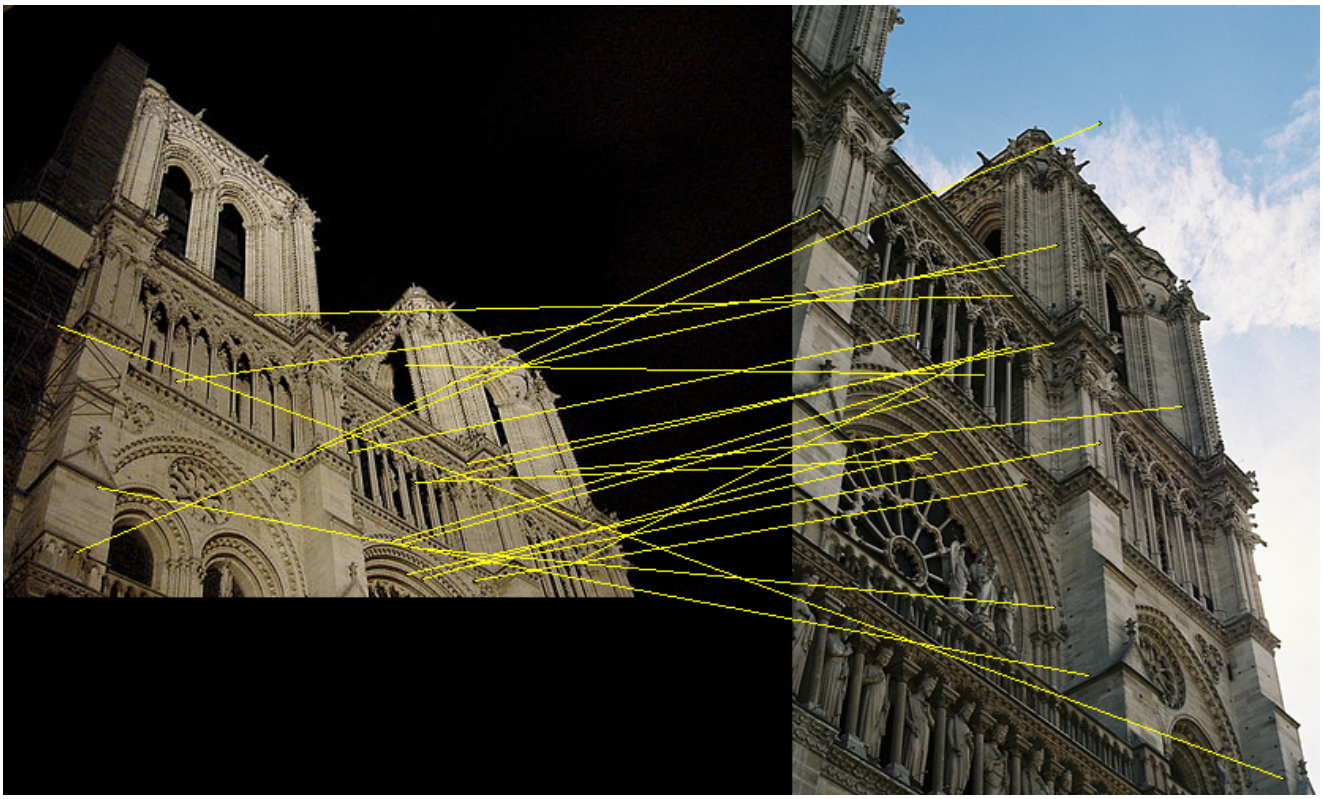


SELF IMAGE
I
V

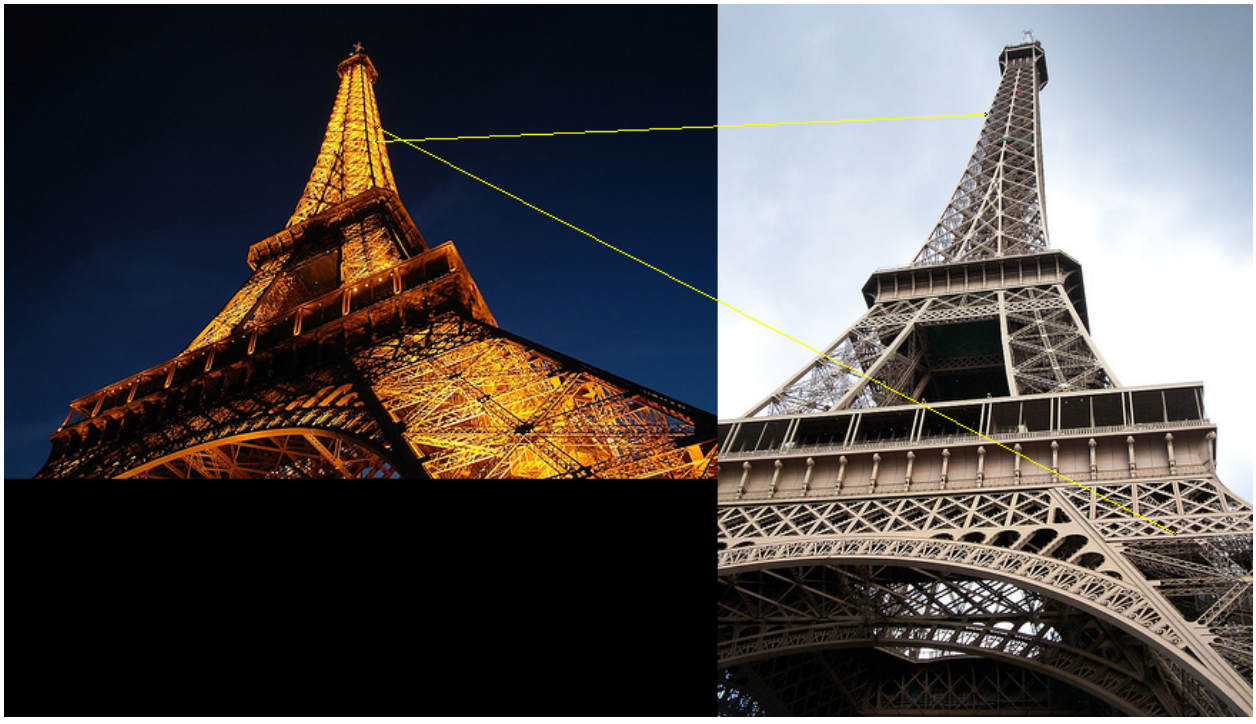
PART 1 RESULT IMAGES

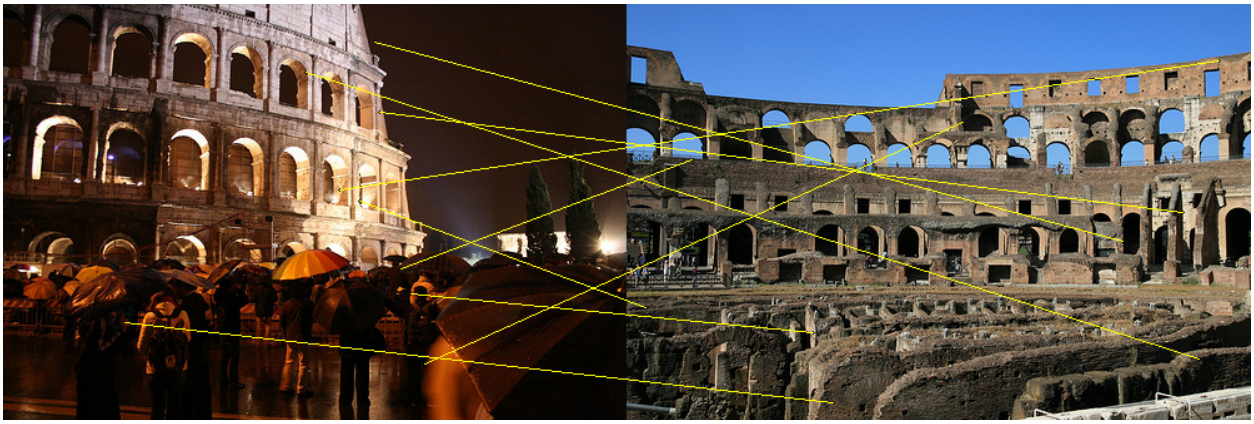
GOOD MATCH



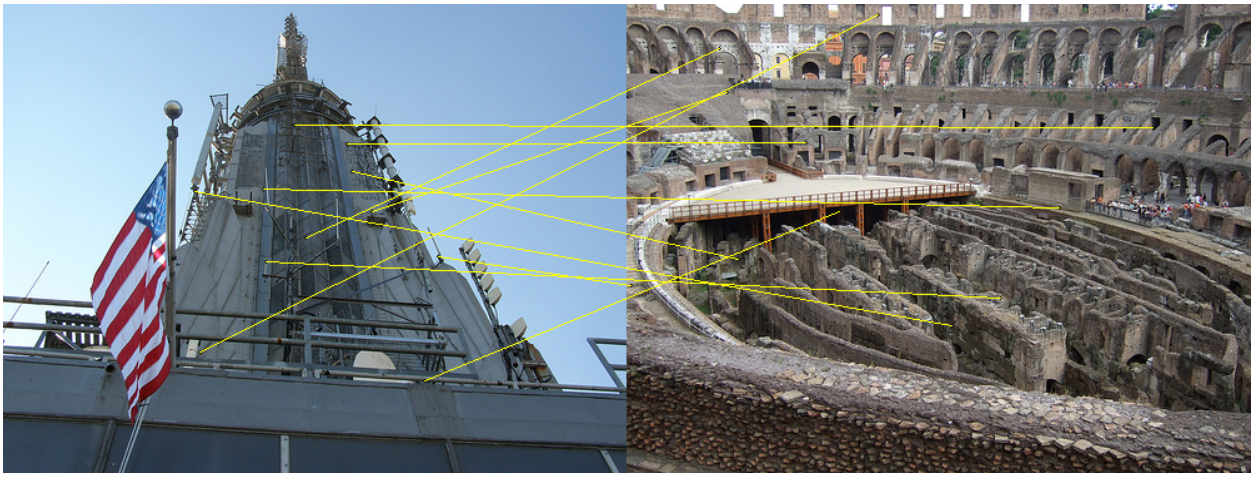


FALSE POSITIVES



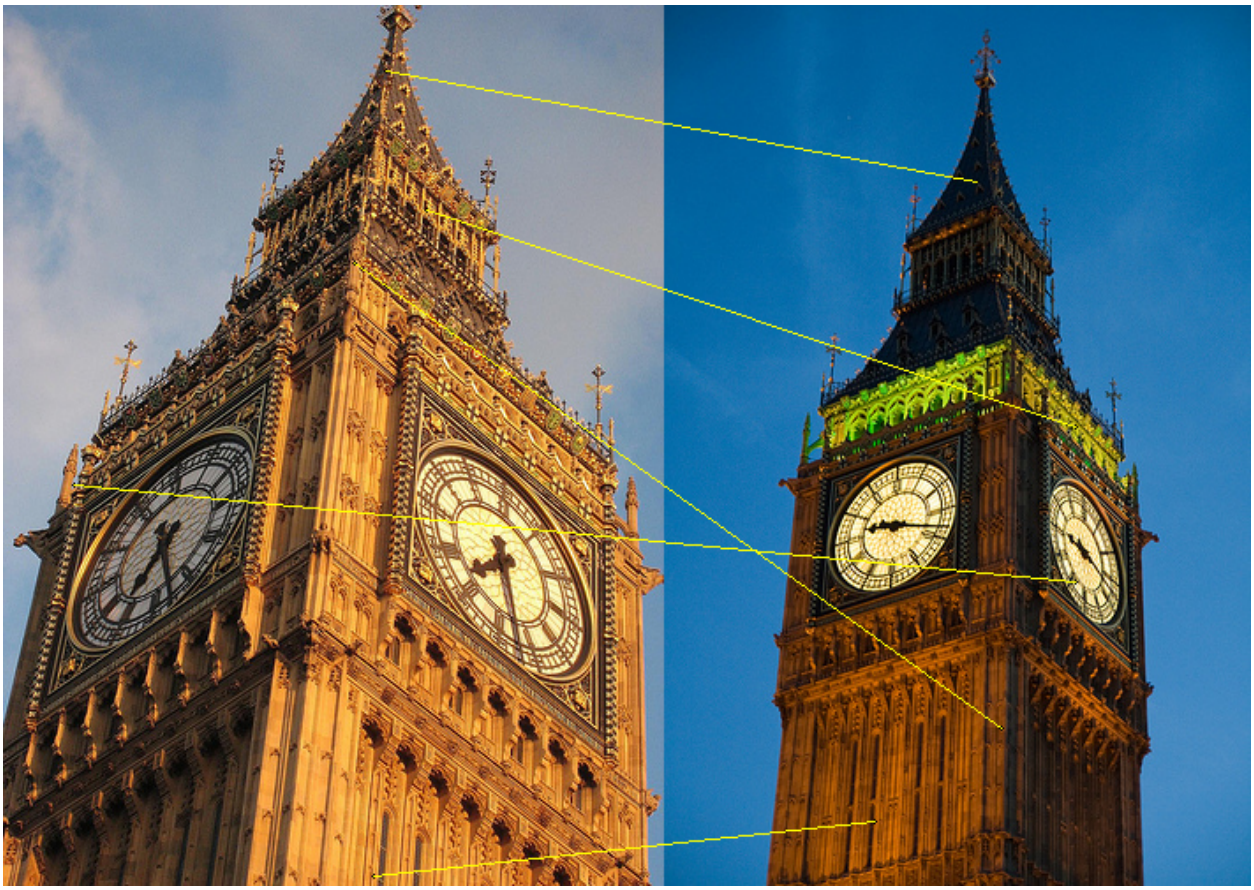


WRONG MATCHES

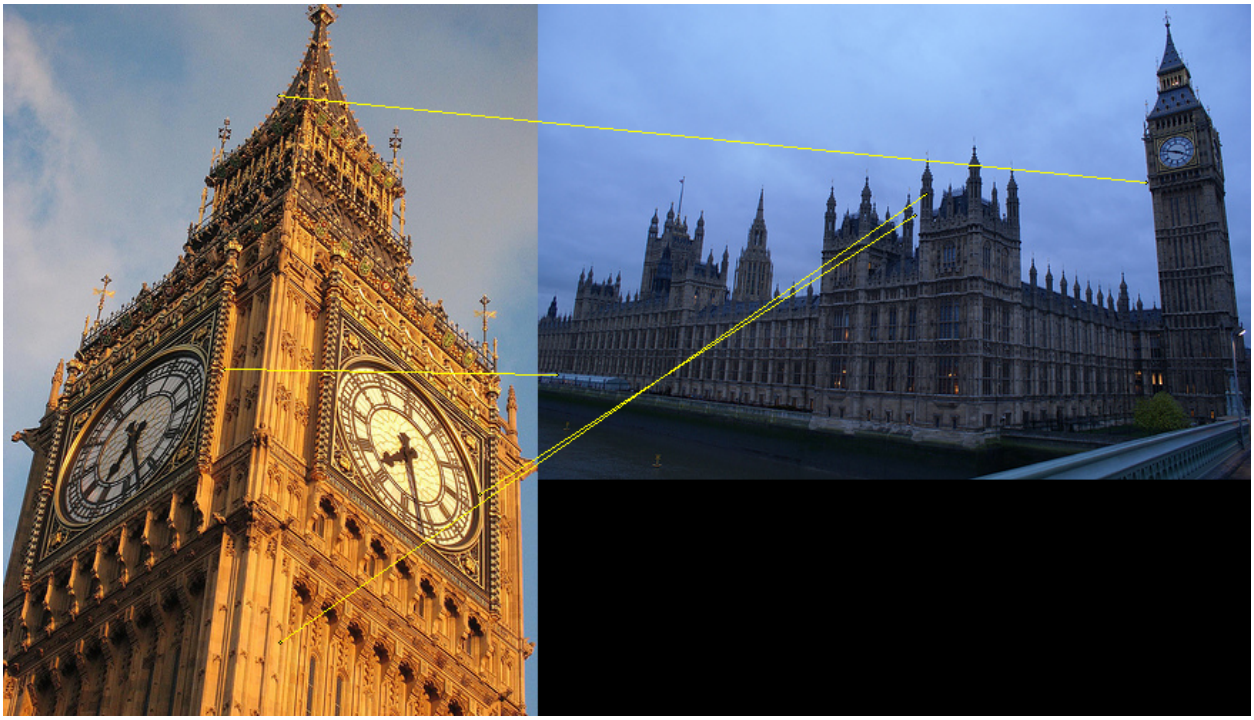


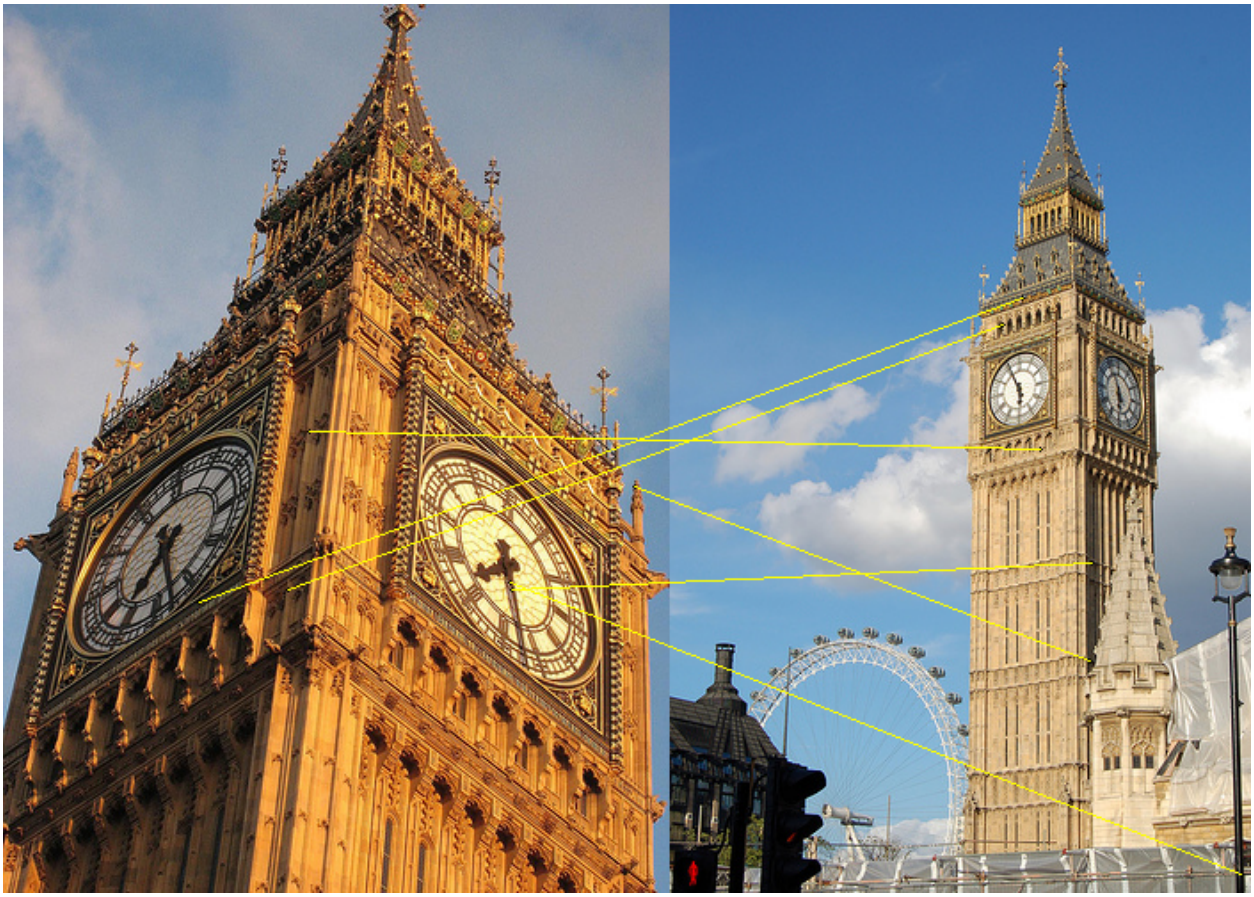
PART 2 RESULT IMAGES

GOOD MATCH



FALSE POSITIVES





WRONG MATCH