

## Part-1

### 1. Sift Matching – Brute force

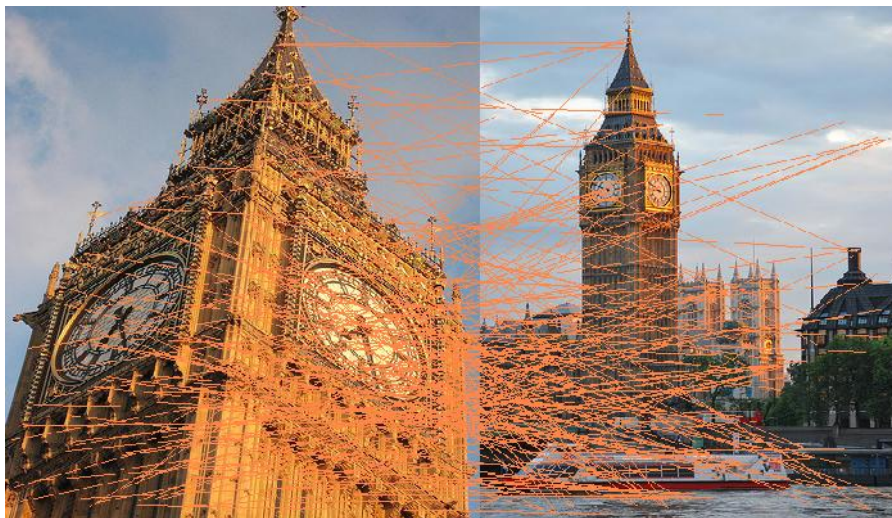
We define two images as match if the ratio of euclidian distance between the closest match and the second closest match is above a threshold. Intuitively if the threshold is small, fewer points are considered and if the threshold is large, closer images are also considered a match. The trade-off is achieved by changing the ratio.

We tried matching using different trade-offs,

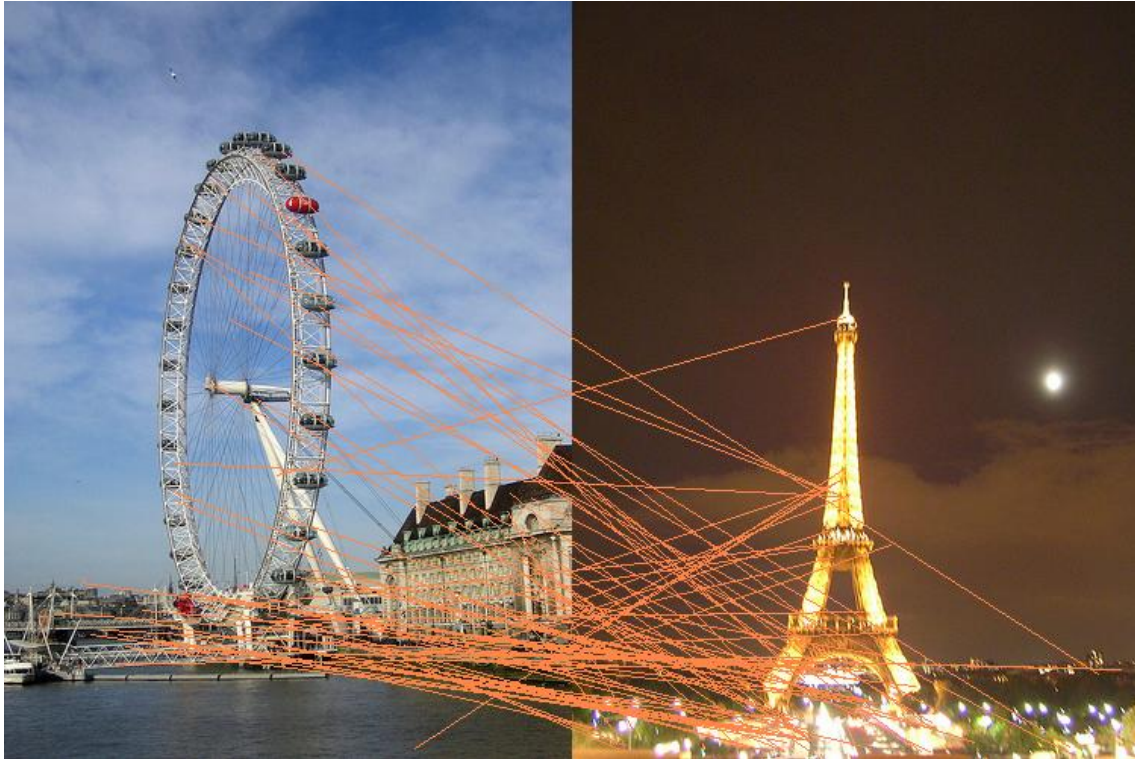
For the ratio of 0.7, below are the images and sift descriptors. It can be seen that there are just a couple of sift matched found between these images even though they from the same scene.



For the ratio of 0.9 however, it can be seen that there are much more SIFT descriptors than the image shown above.



For the threshold of 0.6, when a different scene is used it seems to find a reasonable amount of sift vectors to describe the image.



### 3. Precision of the Image Retrieval Program

In order to find calculate the performance of the algorithm, we wrote a wrapper python code which would call the C++ executable with the required parameters. The program initially decipheres the image category by file name and iterates through each category thereby randomly sampling the query image and excludes the query image from the search image parameter. Thus the input image can be never exactly matched, however approximate nearest match in the library is found and precision is calculated.

This process was ran for multiple ratios, and following are the results.

Image Category	Ratio of 0.7 / Precision	Ratio of 0.9 / Precision
Notredame	0.44	0.33
Tatemodern	0.11	0.22

Sanmarco	0.33	0.33
Bigben	0.11	0.11
Effiel	0.22	0.11
Trafalgar	0.22	0.11
Louvre	0.25	0.0125
Colosseum	0.11	0.11
Empirestate	0.11	0.22
Londoneye	0.11	0.11

Overall, the ratio of 0.7 from sift descriptors performed better than the 0.9. However since the query images were randomly sampled, the query image could have also be a bad selection.



## Part 2

1.

### Steps

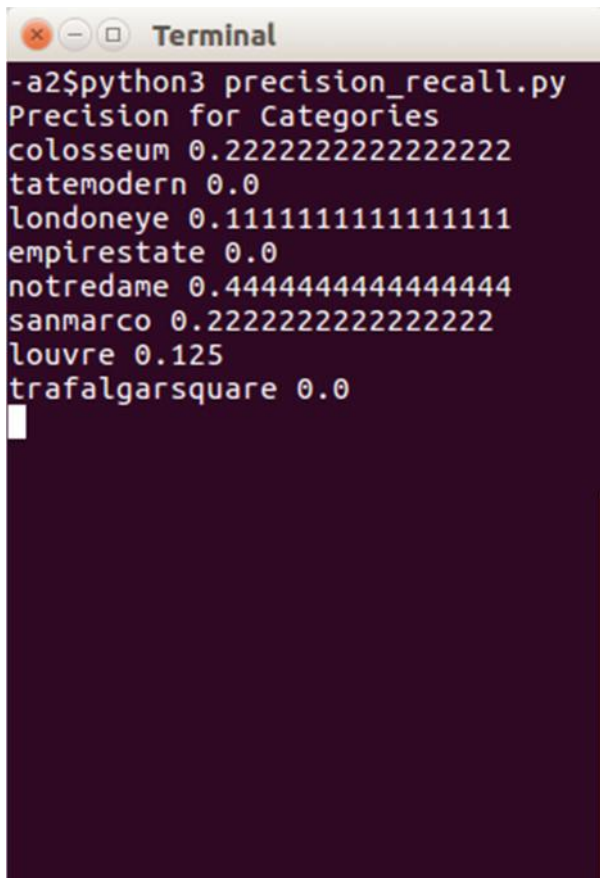
1. We pick four points at random from the set of matched points that we have and we calculate the homography
2. Using that homography we do the translation for all the matched points.
3. We calculate the error using Euclidean distance
4. We repeat the same steps for many iterations and then we pick the homography with least error

Note : We had implemented RANSAC , since we noticed the requirement for this question at the last moment. Hence we could not complete it as per the expectation. However the code is available and commented out in part 2 code section. We have attached the results for this but we are not sure about the correctness of the results.

Here is the example of Homography and SIFT matching.



The Precision is calculated by the same method as part-1,q3 and the performance metrics are as follows.

A terminal window titled "Terminal" with a dark purple background. It shows the execution of a Python script and its output. The output lists precision values for nine categories: colosseum, tatemodern, londoneye, empirestate, notredame, sanmarco, louvre, and trafalgarsquare. The values range from 0.0 to 0.4444444444444444.

```
-a2$python3 precision_recall.py
Precision for Categories
colosseum 0.2222222222222222
tatemodern 0.0
londoneye 0.1111111111111111
empirestate 0.0
notredame 0.4444444444444444
sanmarco 0.2222222222222222
louvre 0.125
trafalgarsquare 0.0
```

While comparing to the 1-3, the brute force seems to have better precision than the new method. Trafalgar and tatemodern are the most difficult images to match on.

2.

We did repeated trials to find the optimum value for k and w. We found the value for w between 40 and 50 to be optimal. The below results show the results got for varied w values with k kept constant at 50

### Steps

#### 1. Varying W values

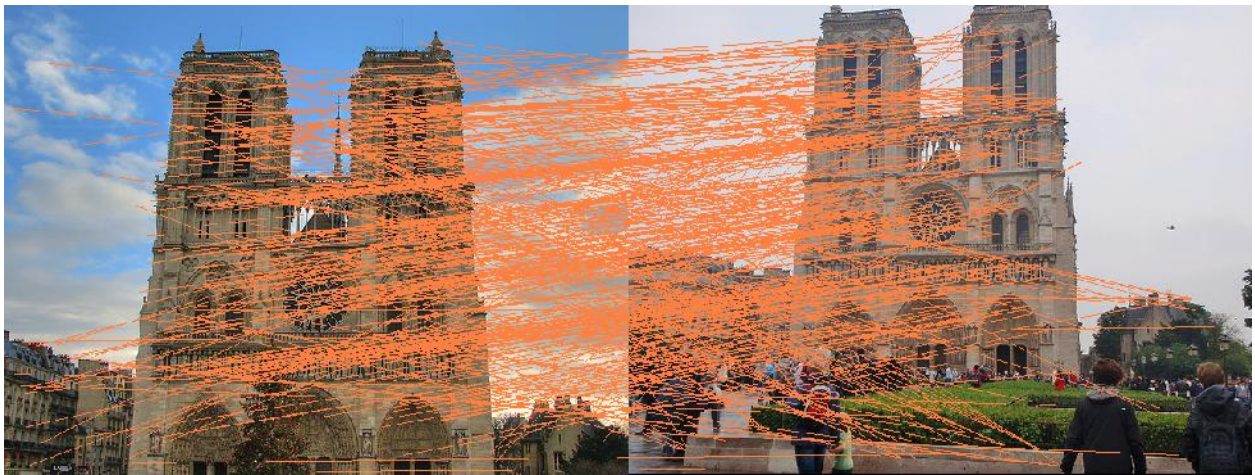
##### Trial 1

**k= 50 , w= 41 , Time taken - 32.75**



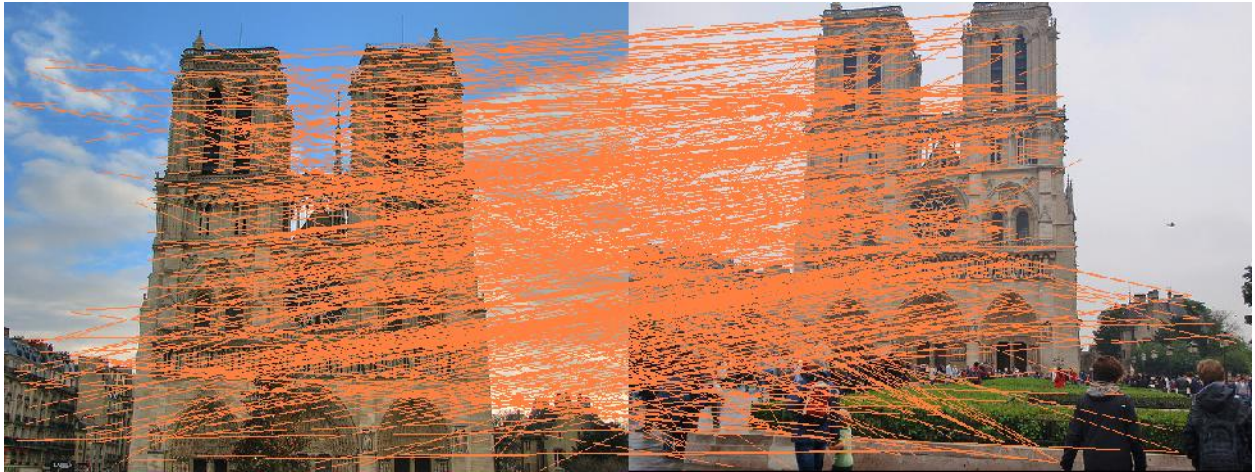


**K=50 , w=45, Time taken - 37.1**





**K=50 , w=50, Time taken - 39.48**



We found from experiments that the value for w to be optimal at 41

### Varying K values

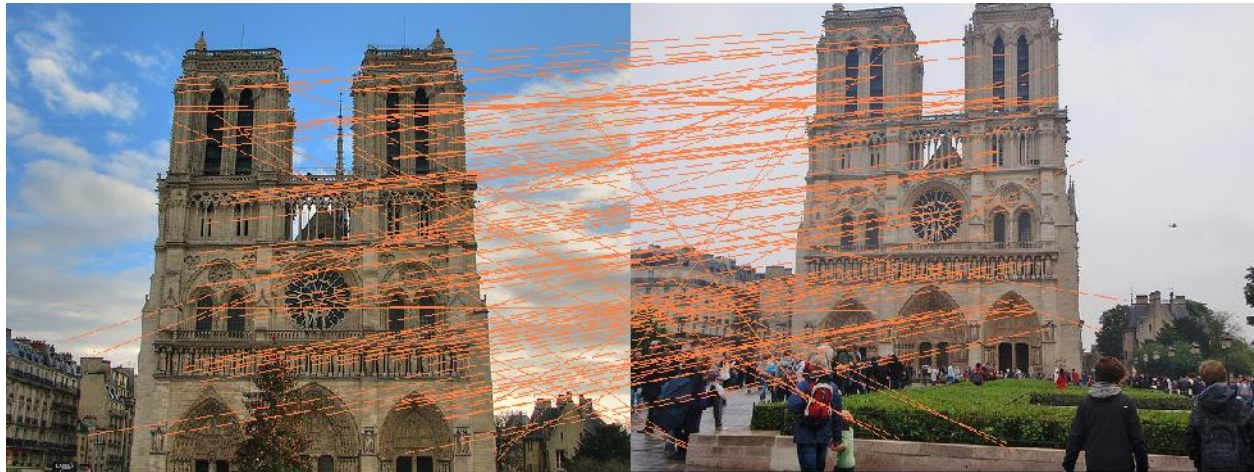
#### Trial 1

**K=55 , w=41, Time taken - 35.42**



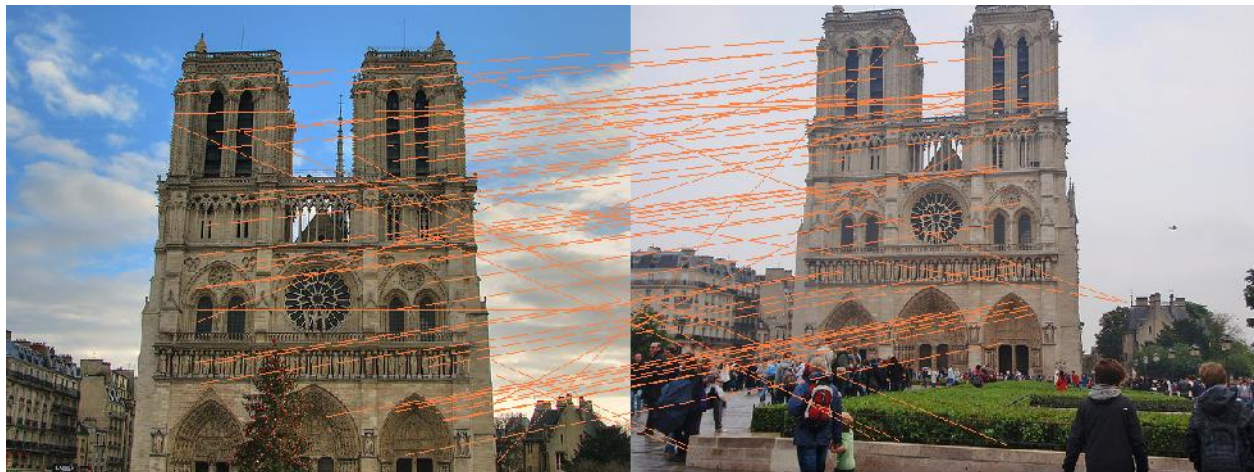
### **Trial 2**

**K=60 , w=41, Time taken - 34.81**



### **Trial 3**

**K=70 , w=41, Time taken - 34.19**



From our trials we find that the value of K as 70 seems to be optimal for our case.



## Part 3

1.

### Image Warping

The task was to take an input image and apply a transformation matrix to produce a corresponding warped image. We used Lincoln.png that was given to produce the below warped image



2.

### Steps

1. We use an image from Sequence 1 and produce a transformation using the transformation matrix given.
2. CalculateHomography() - The function calculates the homography between each image and the other images inputted. The first image serves as the base image while the other images in the sequence are warped to an image to have been taken in the first image camera's coordinate system.

### Results

#### Warped images

**Base Image** , The other images are warped to this image's camera's coordinate system

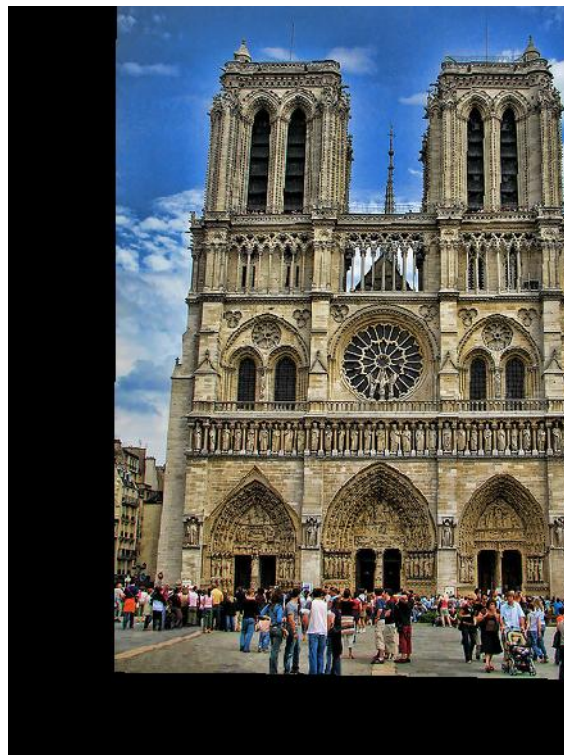


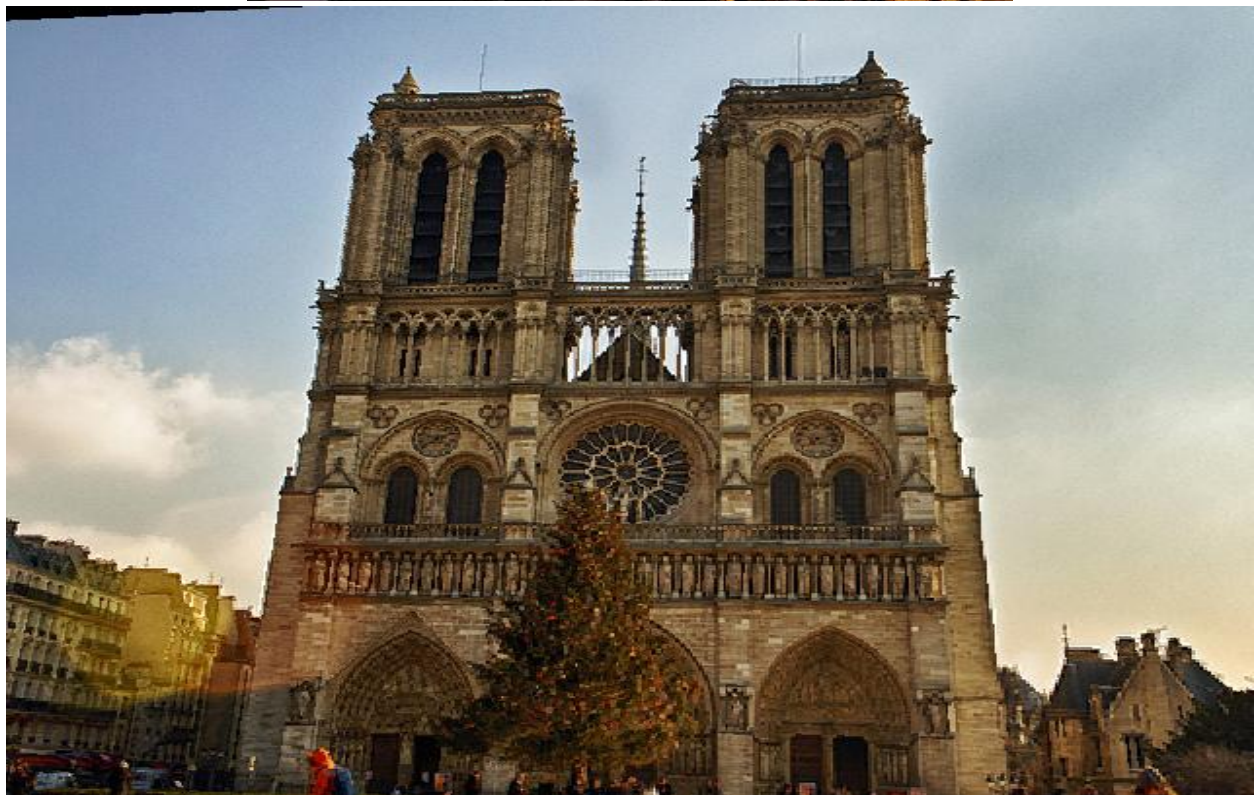
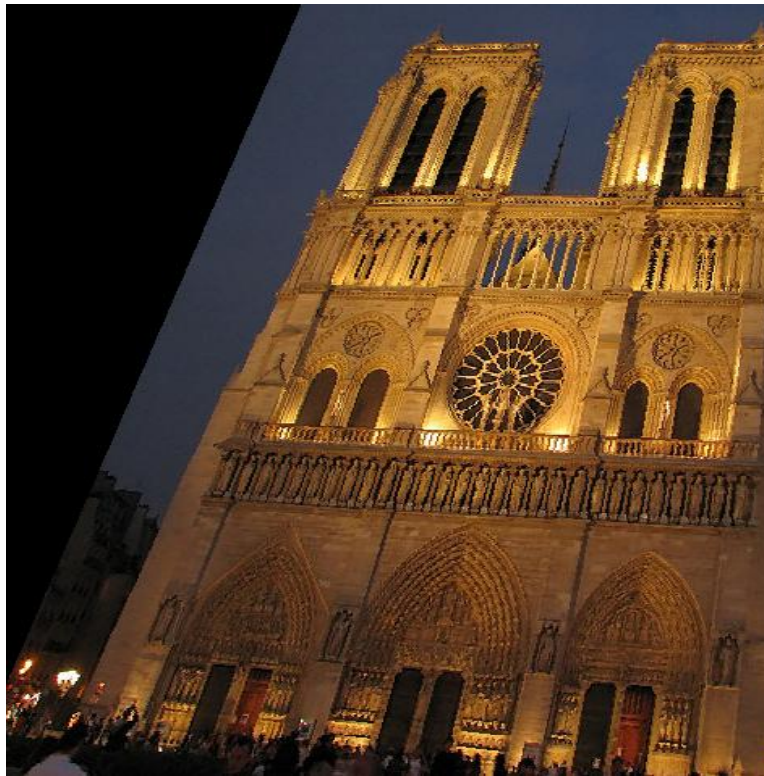
### **Warped Images**

The following are the list of warped images. We set a threshold to the value to 0.8 to obtain the following warped results. We found that setting a threshold of 0.7 worked better for sequence one but didnot work that well for sequence 2. On analysis we found that there might not be enough matches between the two images for it to produce any concrete results.

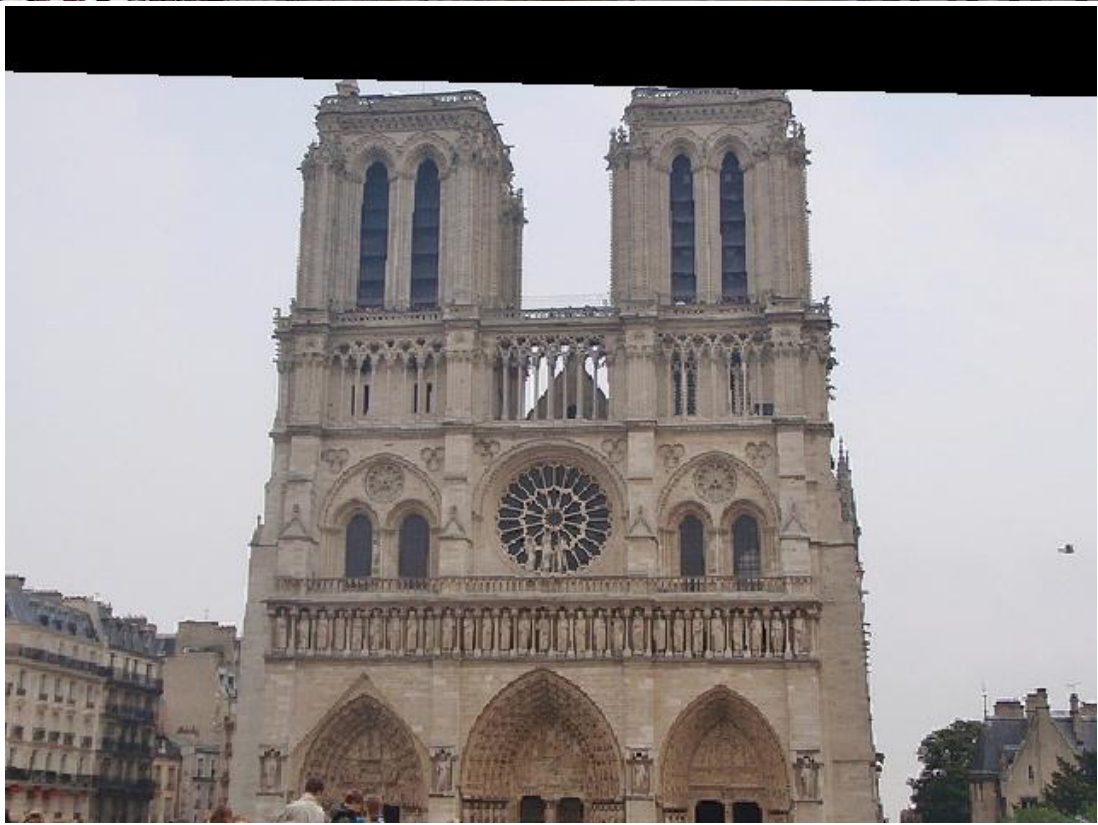
**Sequence 1 - Time taken - 31.88**







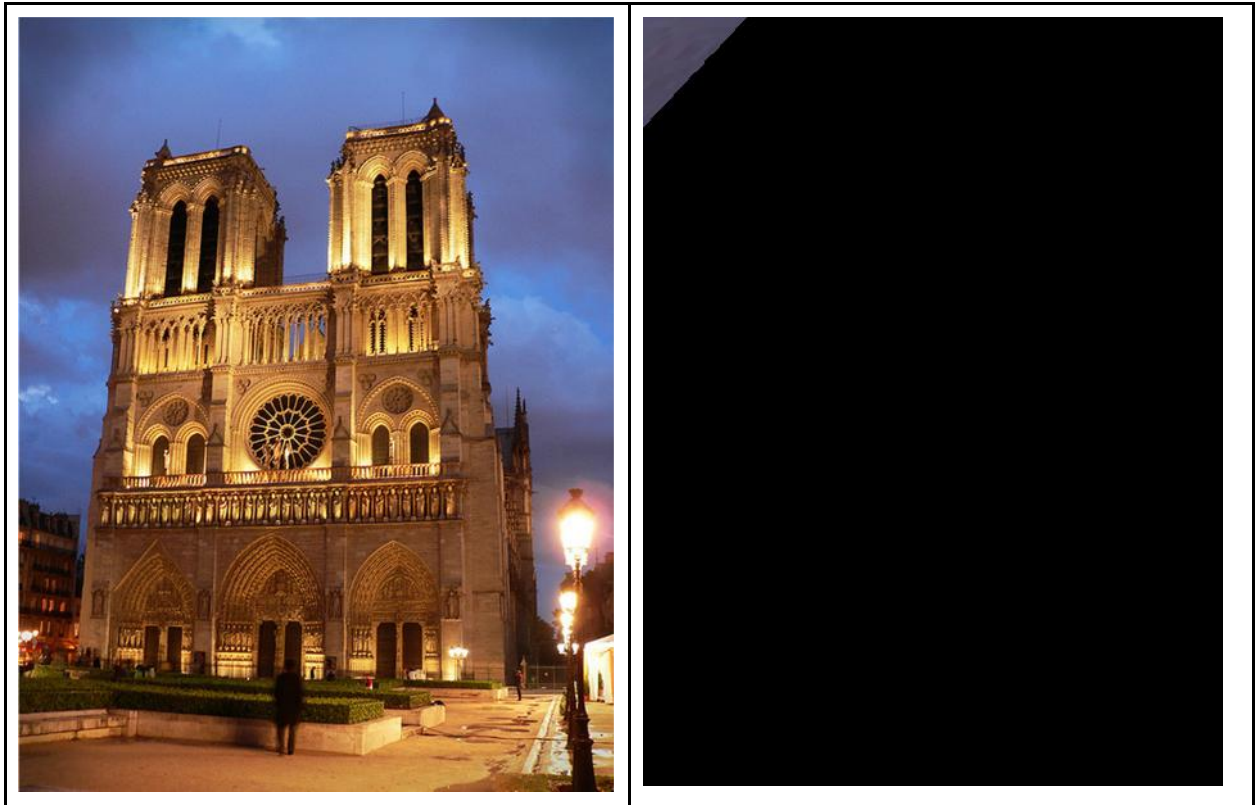






Noisy warped images





The image on the left is the actual image and the image on the right is the warped image.



Even in this image we could see that the image was not warped as expected. The above two images gave better results when we ran multiple times. But the image at most times gave bad fuzzy results. We felt that this could be because of the random point matching technique that we are following to acquire matches between the base image and the above two images.

### **Sequence 2**

The results were similar for sequence 2. We applied the same threshold while warping for sequence 2. However when we changed the threshold to 0.7 we found that not enough matching points were not found for certain images. Hence we had to increase the threshold and on increasing the threshold we detected a lot of false positives which led to the image being distorted. Hence the results for this sequence was not as expected.