How to run?

First compile the code by using the command ‘make’. Then type ./a2 followed by the part number and then the parameters for the specified part to be examined. In the following steps we have described the instructions for execution for each specific part.

Also our make file makes use of C++11 and has been modified to below code

```
all: CImg.h a2.cpp
	g++ a2.cpp -o a2 -lX11 -lpthread -I. -Isiftpp -O3 siftpp/sift.cpp -std=c++11

clean:
	rm a2
```

For Part 2, we have used an additional header file “dirent.h” to use the basename() method for extracting file names from command line arguments of the program.

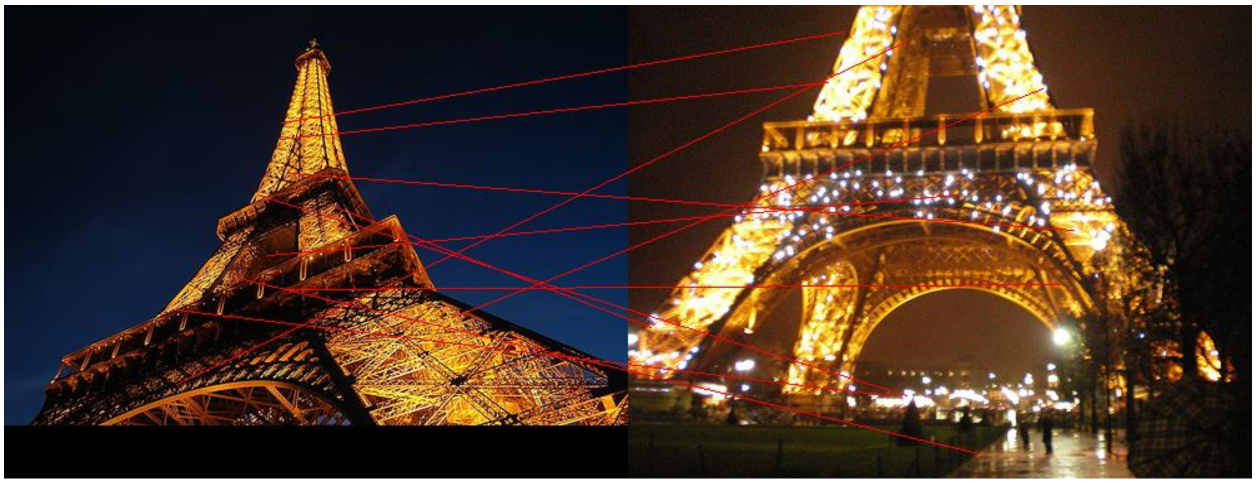
Part 1:Image Matching

1.In this part we have to take two images,apply sift on each one and then finally provide the matching descriptors. We calculate the SIFT descriptors for each image individually and then obtain the distance between the points by using the distance formula.Initially we kept a threshold by trial and error to determine if the matching descriptors obtained were logically correct. On doing this we got the boundary conditions beyond which there were no matching descriptors for any image. Since SIFT requires the ratio of the closest and the second closest match we set two variables to values beyond the boundary values and then obtain the closest and the second closest value which then we take the ratio of the closest match to the second closest match and set the threshold by trial and error. In our case we obtained the optimum threshold to be less than 0.8. The output can be viewed through a file called ‘matched.jpg’.

To run the code first type ‘make’ to compile followed by ./a2 part1.1 and then the two images for which we must compute SIFT match and finally the name of output file where the resultant image can be viewed.

```
Ex: ./a2 part1.1 bigben_2.jpg bigben_3.jpg matched.jpg
```

Shown below is a sample output after applying SIFT match between two images:



2.Here we were required to provide the list of matching images to a given query image and provide the image names along with the number of matching descriptors in the descending order. The function used here is the same as the one used in the previous step. The only addition is the sort function which takes two images and a compare method as a parameter and in the compare method we return the highest matching descriptors. Finally we print out the list of images and its corresponding descriptors in descending order.

To run the code type 'make' to compile and then ./a2 part 1.2 followed by the query image and then the list of images to which the query image has to be compared.

Ex: ./a2 part1.2 bigben_2.jpg bigben_3.jpg,bigben_10.jpg……..

3. Here we calculate the precision of SIFT match function we used in the previous question. From each of the 10 attractions, we choose 1 image of that attraction at random to use as the query image, and then pass all 100 images as the others. Then we print the top 10 ranked images returned by our program, and calculate the percentage of images that are from the correct attraction. This is precision of our system.

To run the code type 'make' to compile and then ./a2 part 1.3. This will output the random images that were chosen as query and then precision for each of the randomly chosen image.

4. The SIFT matching done in part 1.3 is very time consuming as we need to calculate 128D Euclidian distance between each descriptor in both images. To make the process faster we use the method mentioned in assignment.

First we create a multidimensional array x consisting k(taken 28) 128D vectors randomly sampled from Gaussian distribution ~N(0, 1). Now we do dot product with of each of the 28 x vectors on every sift descriptor to convert sift descriptors into a set of 28 numbers. We scale these dot product by dividing it with w(taken as 32). Then we calculate Euclidian distance between each descriptor in both images using these 28D vectors. We stored all these distances in a multidimensional array called distance_values. This will reduce the runtime compared to the 128D space Euclidian distance calculation.

After getting the distance_values we choose a threshold to select certain set of candidate nearest neighbors. Then we calculate Euclidian distance with 128D space on these selected neighbors to find the nearest one.

To estimate w and k we did a lot of iterations with various images. We choose w = 32 and k = 28 so that the values of 28D space distances are almost in the range of 128D space distances.

Runtime for this function is faster compared to the normal SIFT match function as we are doing distance calculation in only 28D space.

To run the code type ‘make’ to compile and then ./a2 part 1.4. This will output the top ten matched images and precision of the selected query image while comparing it with 100 images in the folder.

Part 2: Image warping and homographies

Q# 1.

Method implementing it :
warpImage(CImg<double> input_image,CImg<double> transformationMatrix,string outputFileName);

We first initialized a CImg<double> type result matrix (output image) with the following properties:

- Dimension same as the input image
- 3 channels - Red,Green,Blue
- Depth set to default value of 1
- Pixel value of 255 for all pixels

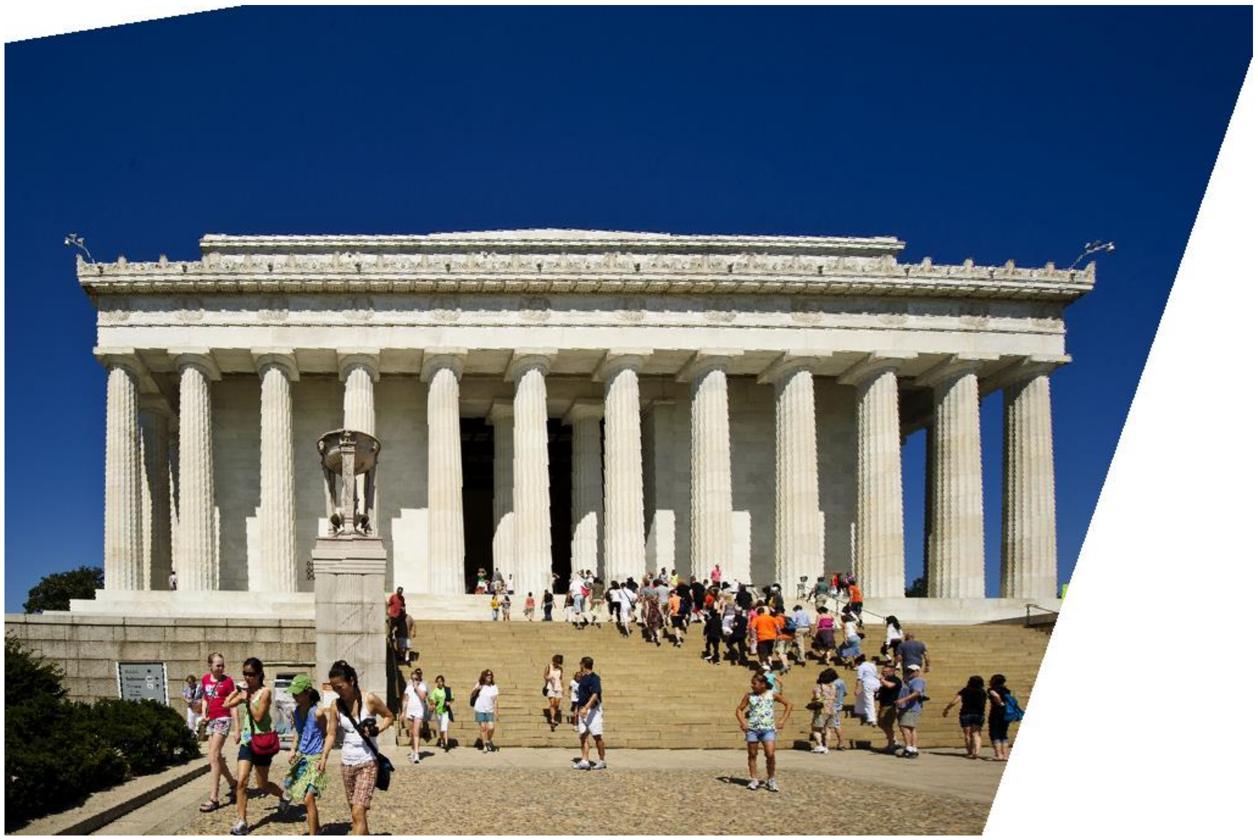
We then used the inverse warping technique by :

- First inverting the transformation matrix given in the assignment using CImg invert() method
- Multiplied the 2 matrices i.e. the output matrix created earlier with this transformation matrix using homographies and calculated the value of corresponding pixel coordinates of input image for each pixel in the output image. Then we copied the pixel values from the input image to the output image accordingly.

Note: We noticed that CImg library doesn't throw any "Out Of Bound" exception even if we try accessing some pixel outside image matrix dimensions. Because of this we were getting partially correct results for lincoln.png. To correct this, we added an additional conditional statement to check if the coordinates computed in the step above lies within the input image dimensions and only then copy the pixel values from input image to output image.

```
if(u_dash>0 && v_dash>0 && u_dash<cols && v_dash<rows){
                    output_image(x,y,0,0)=input_image(u_dash,v_dash,0,0);
                    output_image(x,y,0,1)=input_image(u_dash,v_dash,0,1);
                    output_image(x,y,0,2)=input_image(u_dash,v_dash,0,2);
          }
```

Results After Inverse Warping on lincoln.png:



Q# 2 and 3:

Approach :

1. We first declared a custom class to store feature correspondences between two images:

```
class FeatureCorrespondence{
```

```
            public:
                        double x1;
                        double y1;
                        double x2;
                        double y2;
    };
```

2. Then we generated the sample space for RANSAC by calling the method generateSampleSpace(CImg<double> image1, CImg<double> image2). This method uses the compute_sift() method of Sift.h file to get the matching SIFT descriptors between two images and returns a vector of FeatureCorrespondence class objects.

3. RANSAC algorithm is implemented in the method generateBestModel(CImg<double> image1, CImg<double> image2). There are two parameters to be tuned in this method:

1. variable NO_OF_ITERATIONS : no of iterations RANSAC has to run
2. variable ERROR_THRESHOLD : the acceptable error range to consider a feature correspondence between two images as inliers

To compute a projection transformation matrix(X), we first designed the leftmost matrix below (A) of dimension 8X8 and the rightmost matrix (B) of dimension 8X1. To design this we randomly picked 4 pairs for feature correspondence from the generated sample space and used their coordinates.We used random_shuffle() method to achieve this.

$$
\begin{array}{lccccc}
 & \mathbf{A} & & \mathbf{X} & & \mathbf{B}\\
 & \mathbf{2N \times 8} & & \mathbf{8 \times 1} & & \mathbf{2N \times 1}\\
\begin{array}{l}\textbf{Point 1}\\[1.2em] \textbf{Point 2}\\[1.2em] \textbf{Point 3}\\[1.2em] \textbf{Point 4}\end{array} &
\begin{bmatrix}
x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1\\
0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1\\
x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2\\
0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2\\
x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3\\
0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3\\
x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4\\
0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4
\end{bmatrix} & &
\begin{bmatrix} h_{11}\\ h_{12}\\ h_{13}\\ h_{21}\\ h_{22}\\ h_{23}\\ h_{31}\\ h_{32} \end{bmatrix} & = &
\begin{bmatrix} x'_1\\ y'_1\\ x'_2\\ y'_2\\ x'_3\\ y'_3\\ x'_4\\ y'_4 \end{bmatrix}
\end{array}
$$

Then, to compute the projection transformation matrix(X) we used CImg library's linear solver method:

```
X=B.solve(A)
```

Then, we used the below matrix multiplication (involving homographies) to compute the corresponding pixel coordinates for a given coordinate in the sample space.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \sim \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

As shown below, we have experimented using both euclidean distance and absolute difference between the desired and computed coordinates to count the number of inliers for a given transformation matrix and thereafter evaluated the one that has maximum inlier count.

```
if(abs(u-x2)<=ERROR_THRESHOLD and abs(v-y2)<=ERROR_THRESHOLD){
                                    numberOfInliers++;
                           }

if(sqrt(pow(u-x2,2)+pow(v-y2,2))<=ERROR_THRESHOLD){
                                    numberOfInliers++;
                           }
```

The best model thus generated was further used to inverse warp each image as per the first image’s camera coordinate system, given sequence of two or more images and output the warped image.

Results: Warping Image2 and Image3 to align with camera coordinates of Image1

Image 1

Image 2

Image 2 warped:

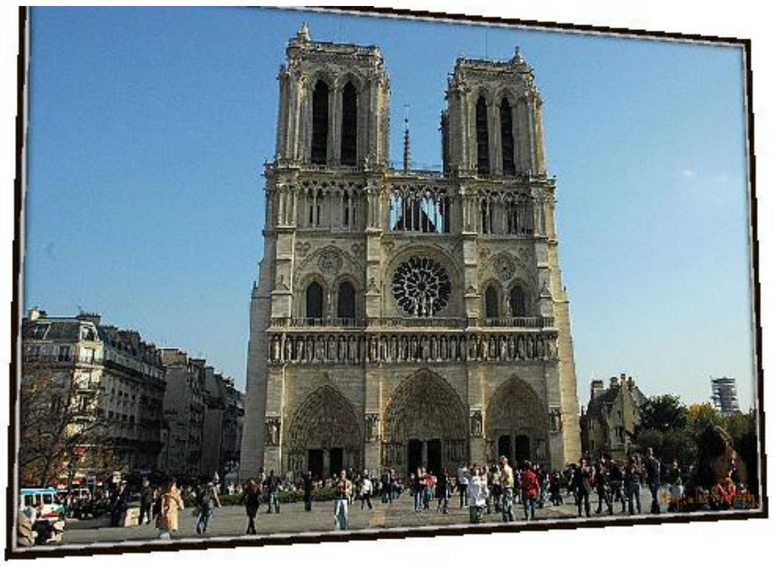


Image 3:

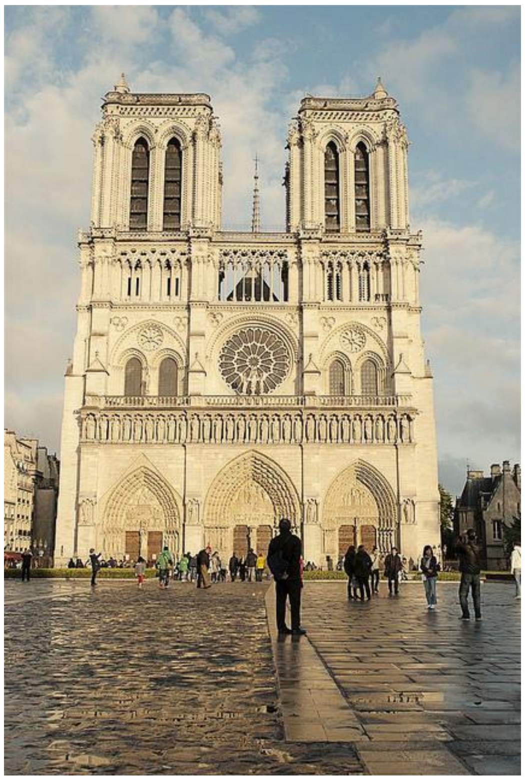
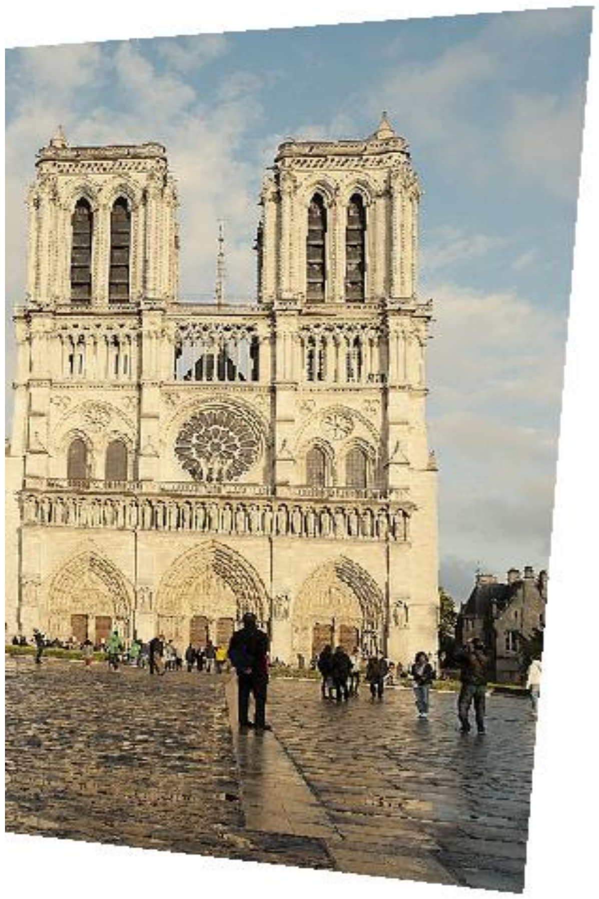


Image 3 warped



To run Part 2 Q1,Q2 and Q3:

```
./a2 part2 img_1.png img_2.png ... img_n.png
```

img_1**-warped**.png is the output for Part 2 Q1
img_2**-warped**.png ... img_n**-warped**.png is the output of Part 2 Q3