IN4393-16 Computer Vision Final Assignment Report
Group 27

Jiaao Dong
Student ID: 4756851

Yuan Tian
Student ID: 4716159

Abstract

This report describes the final assignment of the Group 27 in Computer Vision course (IN4393-16) arranged by Delft University of Technology. The assignment is about the 3D reconstruction task of a castle given a set of images. The first part is about a basic Structure from Motion(SfM) task, followed by the second part of bundle adjustment and ambiguity removal.

1. Feature Points Detection and Descriptors Generation

1.1. Preprocessing of the images

The images are taken from a model of a castle in different perspectives. Before they are fed into a feature detector, we did some preprocessing. The most important one is to segment the castle from the background. If the background is kept, the feature detector will detect so many feature points from that background, which deteriorates the matching result. The matches after RANSAC with background is shown in Figure 1. The outliers are even more than inliers. That's why RANSAC is not capable of removing those points on the background. So we decided to use mean shift clustering to segment the object from the background. The mean shift implementation is in **MeanShiftCluster.m** and **im_meanshift.m**. There is only one parameter for the mean shift segmentation. It is called bandwidth. A large bandwidth yields less segments per image, but it shouldn't be so large that the object will not be segmented from the background. We used a program to select the proper bandwidth for each image. The images after segmentation are shown in Figure 2. The matches between images after segmentation are shown in Figure 3. From the example it is obtained that the background points are suppressed. The data file **lab.mat** contains the segmentation labels. It can be directly loaded by: `load_im_v2(im_folder,1,lab);`

1.2. Feature points detection

The implementation of this part is in the MATLAB function **Feat_extract_v2.m**.

In this step we tried three different methods to find the features:

- Scale-invariant Harris Corner Detector
- Harris-affine Detector
- Scale-invariant Feature Transform (SIFT)

The scale-invariant Harris corner detector is an expansion of the basic Harris corner detector by looking for the extrema of the corner responses in scale-space. The implementation of the detector is provided in the MATLAB function **myharris_ScaleInv.m**. It returns two column vectors $[r, c]$ denoting the rows and columns of each feature point.

The Harris-affine Detector can be extracted by the program provided by the websiteHarris-affine. The algorithm details are described by [7]. The features generated by the Linux binary file downloaded from the website are stored in folder **model_castle_seg_haraff**. The features can be extracted in a MATLAB array by MATLAB function **loadFeatures.m**.

The SIFT feature detector can be extracted by the VLFEAT library[10], which has a MATLAB Toolbox API and it's easy to use. The feature points of the first image by these three methods are shown in Figure 4.

SIFT detector searches for blobs which are found by Laplacian of Gaussian (approximated by DoG). Harris corner detector and its variants search for corner points which yield significant changes in all directions if applied a shift window. From our experiment, SIFT is better to find points on textured surfaces while Harris detectors are better to find corner points. To make them complement each other, we choose to concatenate the features found by Harris-affine detector and SIFT detector. The features are stored in cell array named **f**.

1.3. Generating descriptors

After detecting feature points of each image, the descriptors of these feature points are generated. SIFT descriptors

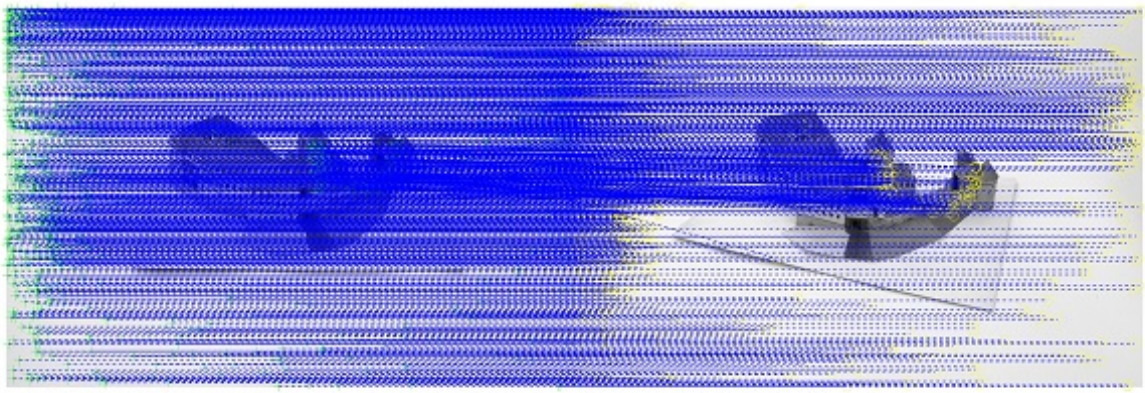


Figure 1: The matches between the 1st and the 2nd image after RANSAC without background subtraction

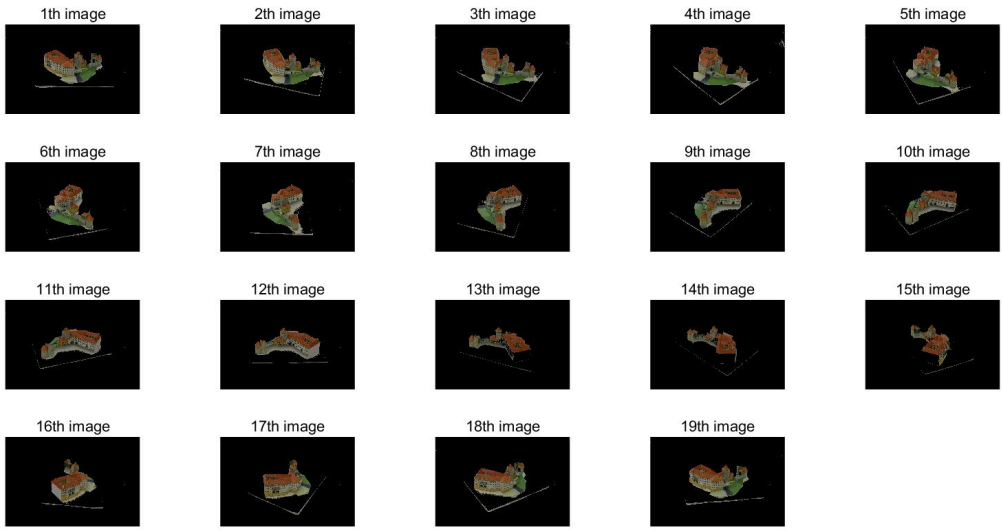


Figure 2: The castle images after segmentation

are invariant to image scale and rotation, and are shown to provide robust matching across a substantial range of affine distortion, change in 3D view-point (In our SfM case) and addition of noise[6]. So we generated SIFT descriptors from feature points found by SIFT detector and Harris-affine detector.

2. Normalized 8-point RANSAC

The implementation of this part is in the MATLAB function **Norm_EightPoints_RANSAC_by_d.m**.

The normalized 8-point algorithm is according to [4]. The parameters and their values are listed in Table 1.

The default matching threshold of **vl_ubcmatch.m** is 1.5. A descriptor d_1 is matched to a descriptor d_2 only if the distance $D(d_1, d_2)$ multiplied by the threshold is not greater than the distance of d_1 to all other descriptors[6]. The higher the threshold, the less feature points are matched but the matches become more precise. In our case, the matching threshold is lower than the default value in order to match as many points as possible. A Lower threshold yields many outliers. Then we adopted a larger RANSAC iteration time to cope with these outliers. More RANSAC iterations will ensure robustness to outliers but take more computational

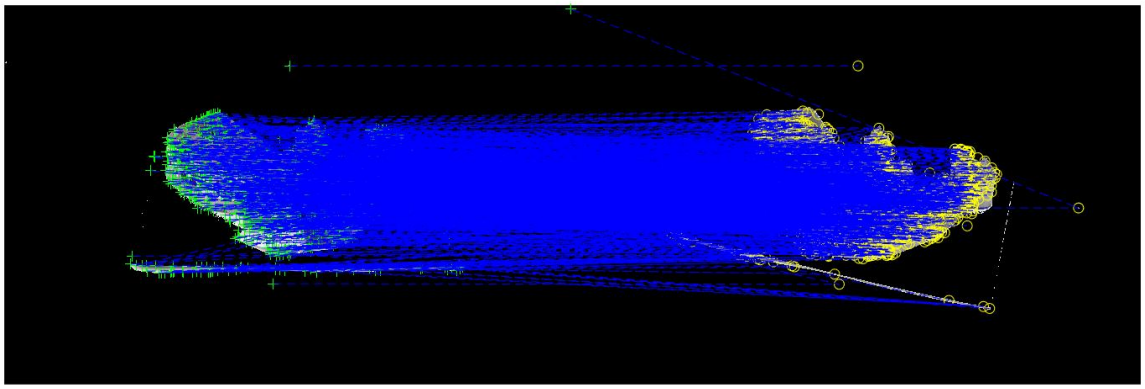
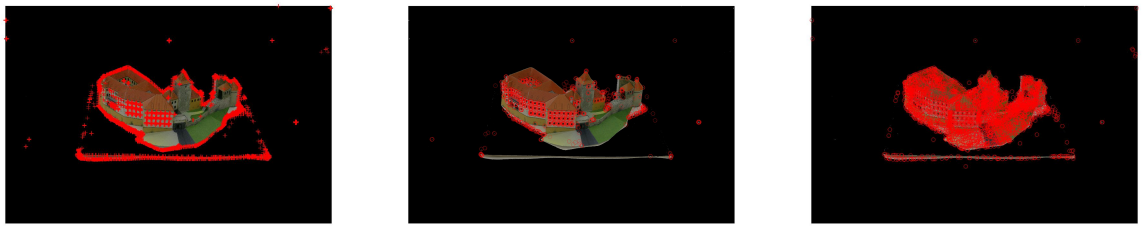


Figure 3: The matches between the first and the second and image before RANSAC after background subtraction



(a) Scale-invariant Harris detector (b) Harris-affine detector (c) SIFT detector

Figure 4: The result on the first image of different detectors

energy, thus we tried to adapt RANSAC iteration times for different images. After some experiments, the image 15 and image 16 are the most difficult image pair to be matched precisely, so we chose to use the highest RANSAC iteration time for this pair. The criterion for inliers and outliers are Sampson distance, which is the geometric distance from the first order approximation of a curve (in our case, a line)[3]. In order to get more points, we set a large threshold (50 pixels). Also the outliers introduced by this large threshold will be removed by a large RANSAC iteration time. The matches are stored in cell array **good_matches**. Some of the matching results are shown in Figure 5. The number of the good matches are shown in Table 2.

In this step, the fundamental matrix is very important. If the fundamental matrix is not correct, the RANSAC will be meaningless. We verify the fundamental matrix by visualizing the epipolar lines of some randomly selected points. The implementation is in **visualize_F.m** and **epipolar.m**. The selected points in the 1st frame and the epipolar line where these points should be found in the 2nd frame are shown in Figure 6.

3. Chaining

The implementation of this part is in the MATLAB function **PointViewMat.m**.

The input of the point-view matrix is the matches after RANSAC. It is constructed according to the instructions provided by the manual of assignment6 in this course. Each row stands for a camera view and each column stands for a unique 3D point. The entries are indices of the feature points in its corresponding image. The last row of the point-view matrix not only contains the matching points between the last two images but also contains the matching points between the last and the first image. (The indices of the last image is in the last row. The indices of the first image is in the first row. Common matches are in the same columns). The common points shared with the previous view of each new camera view in this point-view matrix are listed in Table 3.

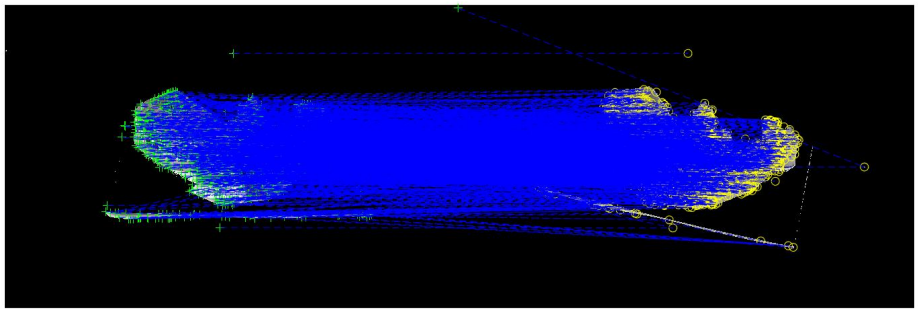
The point-view matrix is of great significance for the next stage. It is verified by finding some points which are seen from several different views at the same time, which

Parameter	value
ubc_match threshold	1.2
Sampson distance threshold	50
RANSAC iteration times	1×10^4 for image 1-4
	2×10^5 for image 15
	2×10^4 for image 16-19

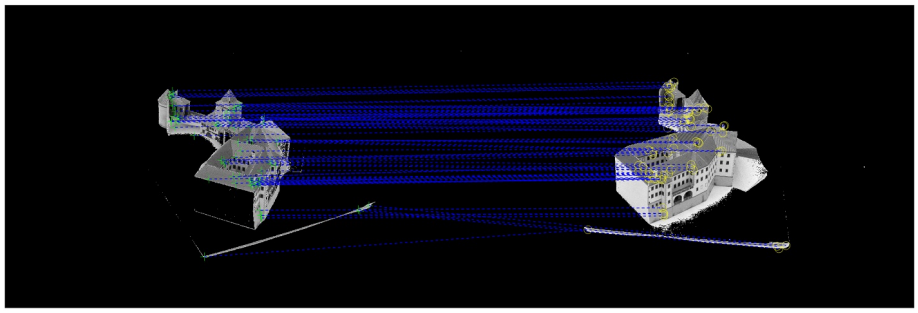
Table 1: Parameters for normalized 8-point RANSAC

Image	1	2	3	4	5	6	7	8	9	10
Number	970	1273	1332	1175	1415	810	558	447	545	666
Image	11	12	13	14	15	16	17	18	19	
Number	658	175	320	297	78	307	500	557	1579	

Table 2: The number of good matches



(a) Matches between 1st and 2nd image



(b) Matches between 15th and 16th image

Figure 5: The matching results of some image pairs

Image	1	2	3	4	5	6	7	8	9	10
Common	970	515	684	627	638	482	209	191	191	279
Image	11	12	13	14	15	16	17	18	19	
Common	328	80	50	101	13	11	113	139	343	

Table 3: The common matches between the previous match set

(a) Selected Points

(b) Epipolar line in the next frame

Figure 6: The selected points in the 1st frame and the epipolar line in the 2nd frame where those selected points should be found

means selecting a column and plot the points at that column. If the points are the same points in different views, our point-view matrix is correctly composed. The implementation is in **visualize_CommonPoints.m**. An example is shown in Figure 7.

4. Stitching

The implementation of this part is in the MATLAB function **mySfM_v2.m**.

The input of this part is the point-view matrix from the previous step.

4.1. Matching points between consecutive images

From the derivation in [9], the rank of the measurement matrix should be at least three. The conclusion indicates at least two consecutive images from point-view matrix should be included in the measurement matrix. More images will generate more reliable and robust motion and structure. But more consecutive images will have less common points (common points are points that can be seen from several consecutive images, as shown in Figure 7). In our case, we choose to make three images in a group (Group 1 contains 1st-3rd image. Group 2 contains 2nd-4th image. Group 3 contains 3rd-5th image, etc.).

4.2. Estimate 3D coordinates

The Structure from Motion task is performed using factorization method described in [9]. Given three consecutive images, the measurement matrix has 6 rows. The number of columns depends on how many 3D points are commonly seen by the three images in each group. The output of this step is a MATLAB cell struct **points_data** containing 6 fields. "PointsIndx" is the column index in the point-view matrix, which is an identification of the unique 3D points in space. "Points_3D" has the 3D coordinates of the point cloud. "Points_2D" has the 2D coordinates of each view in this image set. "Pose" has the camera motion matrix of each view in this image set. "mean_meas" is the mean value subtracted from the measurement matrix, which is useful when re-projecting the 3D points. "Color" is the pixel RGB value of the points in the first frame of each group, which is useful for visualization. Given 19 images, the Structure from Motion yields 17 point sets.

The Structure from Motion result of each image group is verified by visualization. The implementation is in the MATLAB function **visual_SfM.m**. It use 3D point cloud with RGB values to depict the 3D reconstruction from each image group. The Pose is also illustrated in different colors. It also generates the feature points that are seen by the 3 images in each group and the reprojection of these points. Some of the visualization examples are in Figure 8 and Figure 9.

4.3. Procrustes

The implementation of this part is in MATLAB function **my_procrustes.m**.

The mathematical details can be found in [8]. The output of **my_procrustes.m** is a MATLAB struct containing orthogonal transformation, scaling factor and a translational vector.

4.4. 3D Stitching

The implementation of this part is in MATLAB function **CalcTrans.m** and **sti_recursive_overlap.m**.

The transformation matrix between 3D points from each consecutive image group is calculated using **CalcTrans.m** and the transformations are stored in a MATLAB cell. To calculate the transformation between two 3D point cloud, common 3D points and their correspondences are needed for procrustes analysis. Then the points of each image

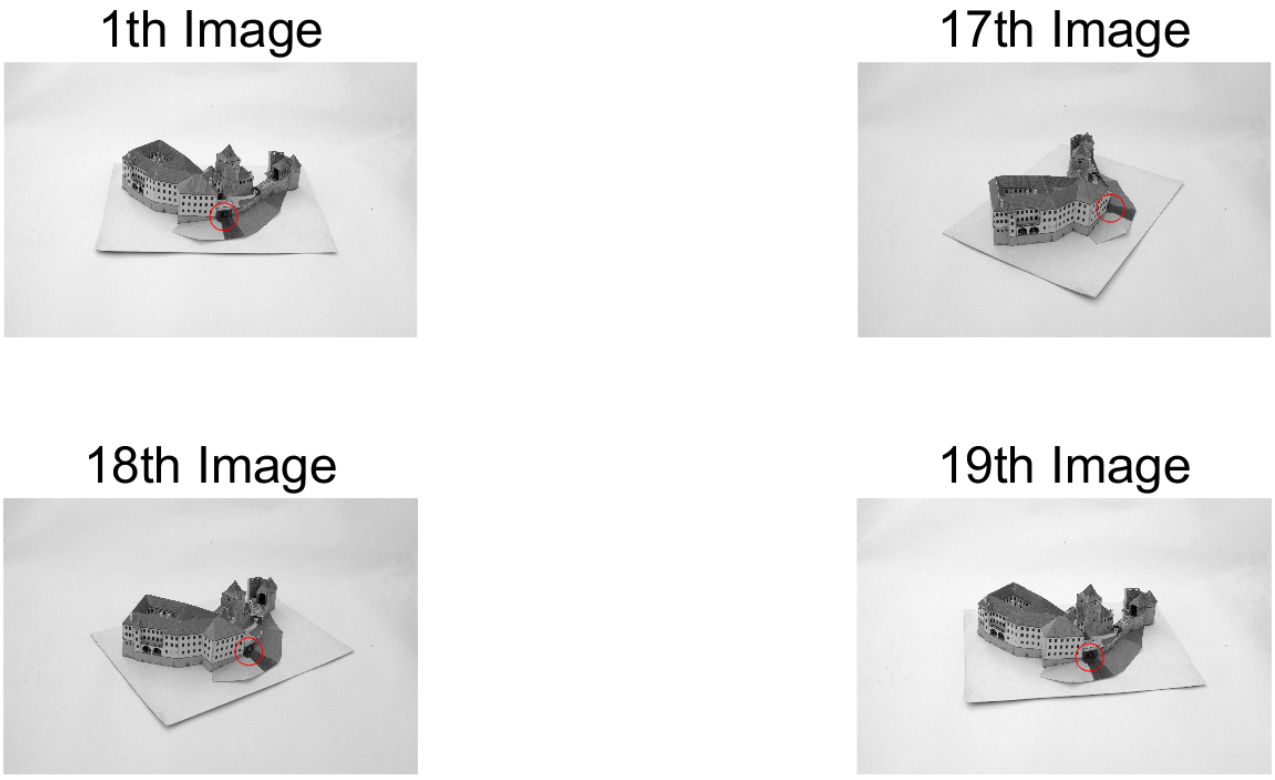
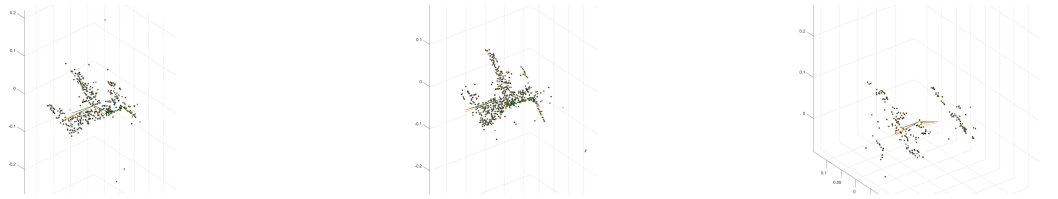


Figure 7: An example point that can be seen from 6 images at the same time



(a) SfM of the 1st to 3rd image (b) SfM of the 2nd to 4th image (c) SfM of the 10th to 12th image

Figure 8: The SfM results of several image groups

set stored in **points_data.Points_3D** are transformed recursively to the first frame by **sti_recursive_overlap.m**. Given 17 point sets from the previous step, 16 transformation between consecutive point sets are calculated.

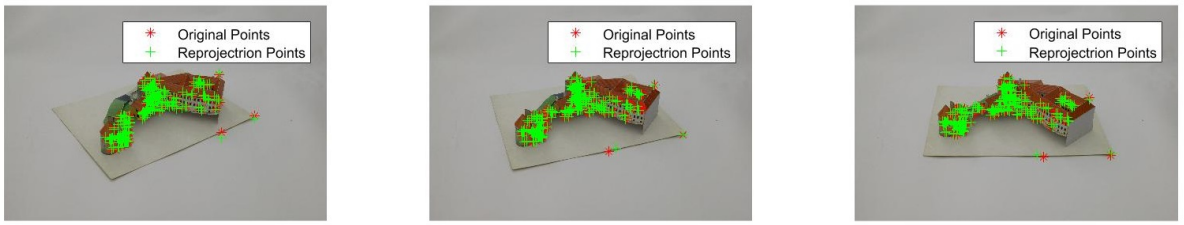
5. Eliminate affine ambiguity

The implementation of this part is in MATLAB function **remove_ambiguity.m** and **visualize_remove_ambiguity.m**.

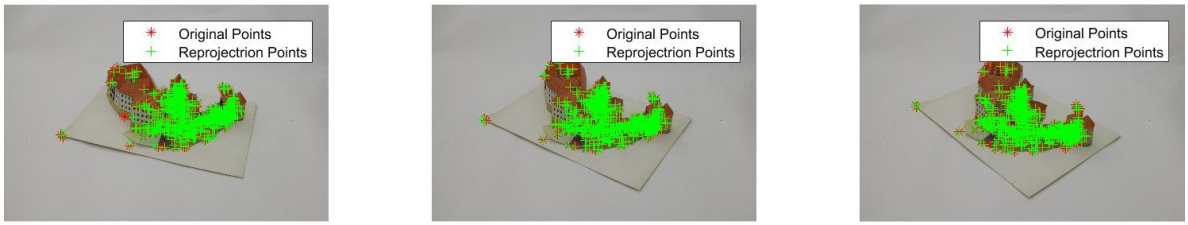
We eliminated affine ambiguity by enforcing Euclidean constraints, which means forcing the camera pose vector to be orthogonal. We try to compute and visualize some poses before and after removing the ambiguity, which is shown in Figure 10.

From structure from motion stage, we can get the poses for each image. But the poses are 2×3 vectors A_i. We want to make sure the first and second row of A_i is perpendicular with each other. Which means the the Equation 1 should be satisfied. But A_i is not square matrix, we use cross product of its first two rows to fill it with the third row. Which means we use a vector perpendicular to A_1 and A_2 at the same time as the third row of A_i. Then we use Equation 2 to calculate L. L should satisfy Equation 3. Then we can calculate C from L using Cholesky Decomposition. This method of filling the matrix with the third row will guarantee the matrix to be positive definite.

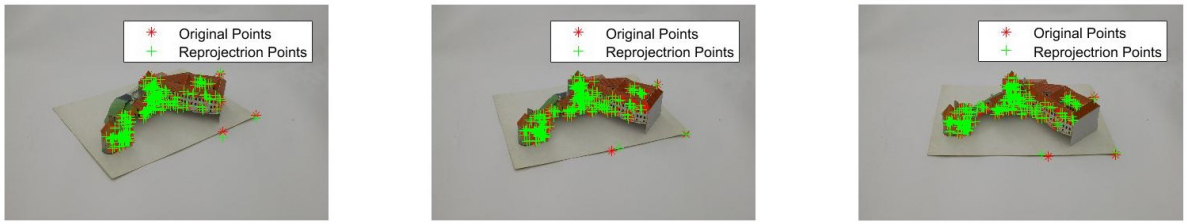
After we have transformation matrix C, we can use it to



(a) Reprojection error of the 1st to 3rd image



(b) Reprojection error of the 2nd to 4th image



(c) Reprojection error of the 10th to 12th image

Figure 9: The reprojection results of several image groups

transform the poses and 3D points in each image groups. The refined points data are stored in **points_data_refine**.

$$A_i L A_i^T = Id \tag{1}$$

$$L = (A^T A)^{-1} \tag{2}$$

$$L = C \times C^T \tag{3}$$

6. 3D model plotting

The implementation of this part is in MATLAB function **surf_3D_PCL_v2.m** and **main.m** line 92-104s.

We tried to use MATLAB **surf** function to surf a 3D surface. **surf** needs to meshgrid the domain, so we need to round our reconstructed points to its nearby grid. This will make some points belong to the same grid. If the grids are finer, the problem can be partly solved. But if there are too many grids, the surface will become harsh. The 3D surface is shown in Figure 12.

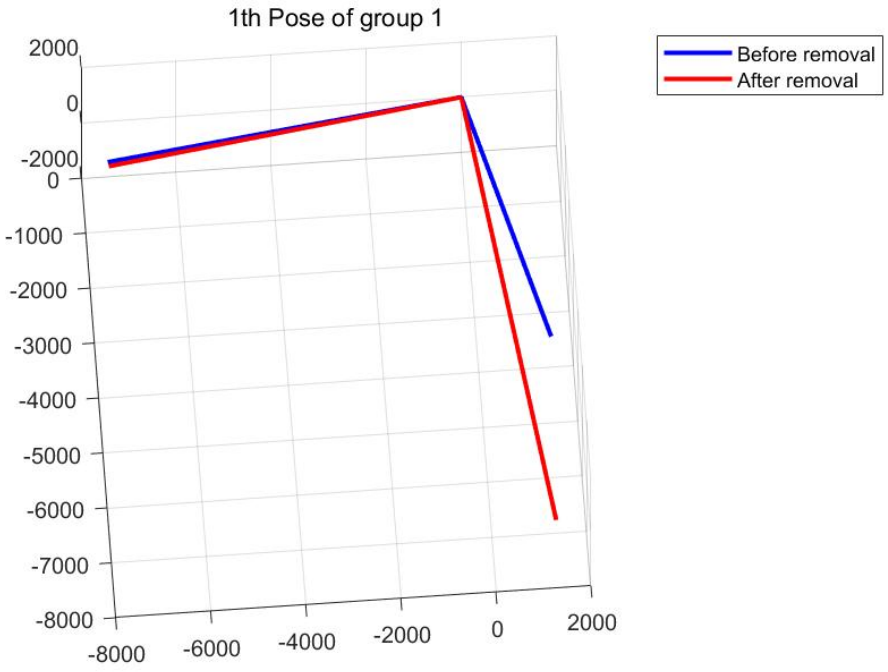
As an alternative, we highly recommend to use **pcshow** in MATLAB computer vision toolbox to visualize the point clouds. We added the color map which is mapped from the original images. The color map is already stored in the **points_data** struct when doing SfM. The point cloud is shown in Figure 11.

7. Bundle adjustment

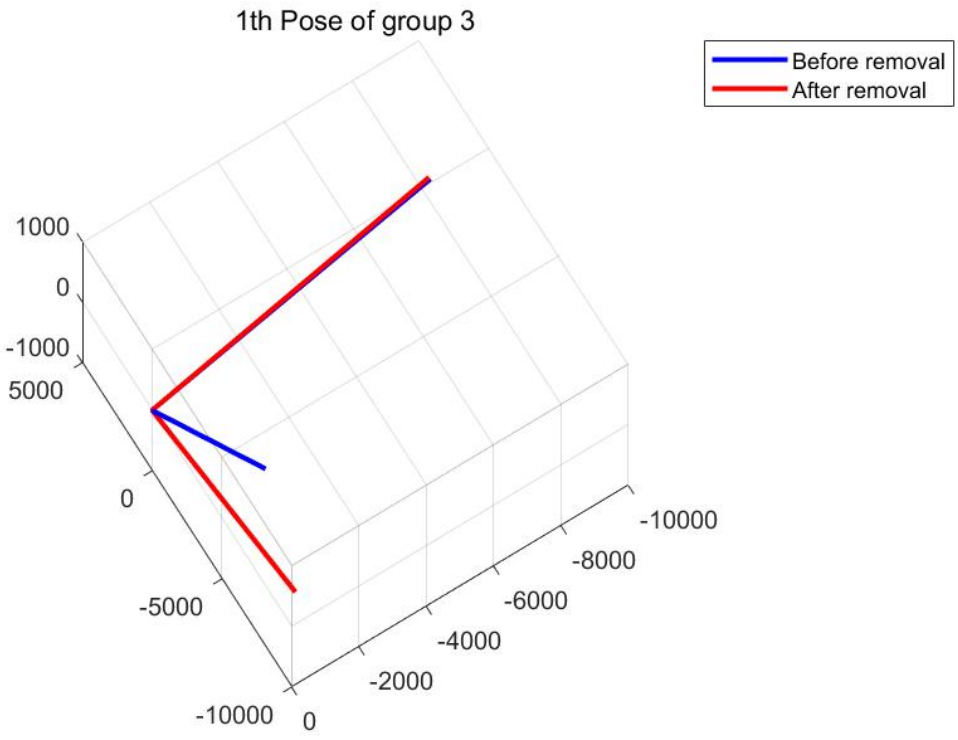
The implementation of this part is in MATLAB function **bundle_adjustment.m**. In the bundle adjustment stage, we try to optimize the overall reprojection error of all the image sets. Each image set contains 3 image. There are 17 image sets in total, so we need to take into account 17 reprojection errors.

MATLAB optimization toolbox has different optimization methods to do the task. First we need to pass the value of poses and 3D points to the optimization function. The value is stored in **points_data** in several different matrices. To facilitate the operation, we concatenate the matrices and reshape them into a vector. The optimization loss function is defined in the scope of the bundle adjustment function to utilize the information when reshaping the vector back into matrix multiplication.

The options for the optimization problems are chosen empirically. There are some arguments to propose Levenberg-Marquardt algorithm, which is one of the most successful methods due to its ease of implementation and its use of an effective damping strategy that lends it the ability to converge quickly from a wide range of initial guesses[12]. In this case, we consider the time and complexity, so we use a basic quasi-Newton algorithm.



(a) 1st Pose in Group 1



(b) 1st Pose in Group 3

Figure 10: The comparison of pose vectors before and after removing ambiguity

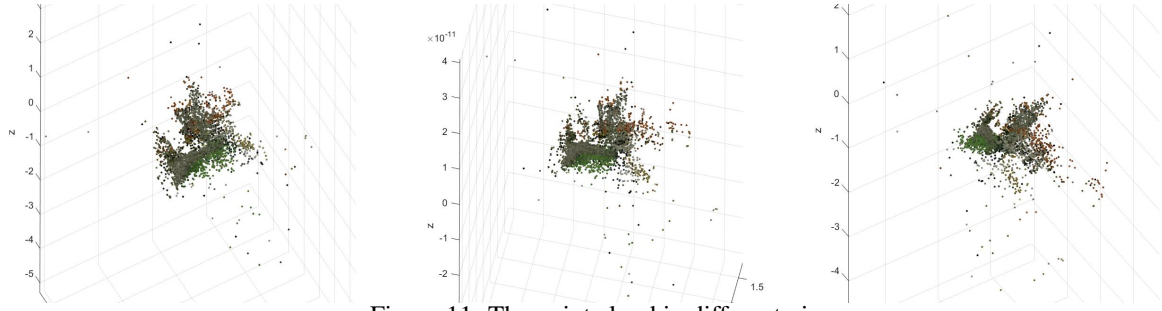


Figure 11: The point cloud in different views

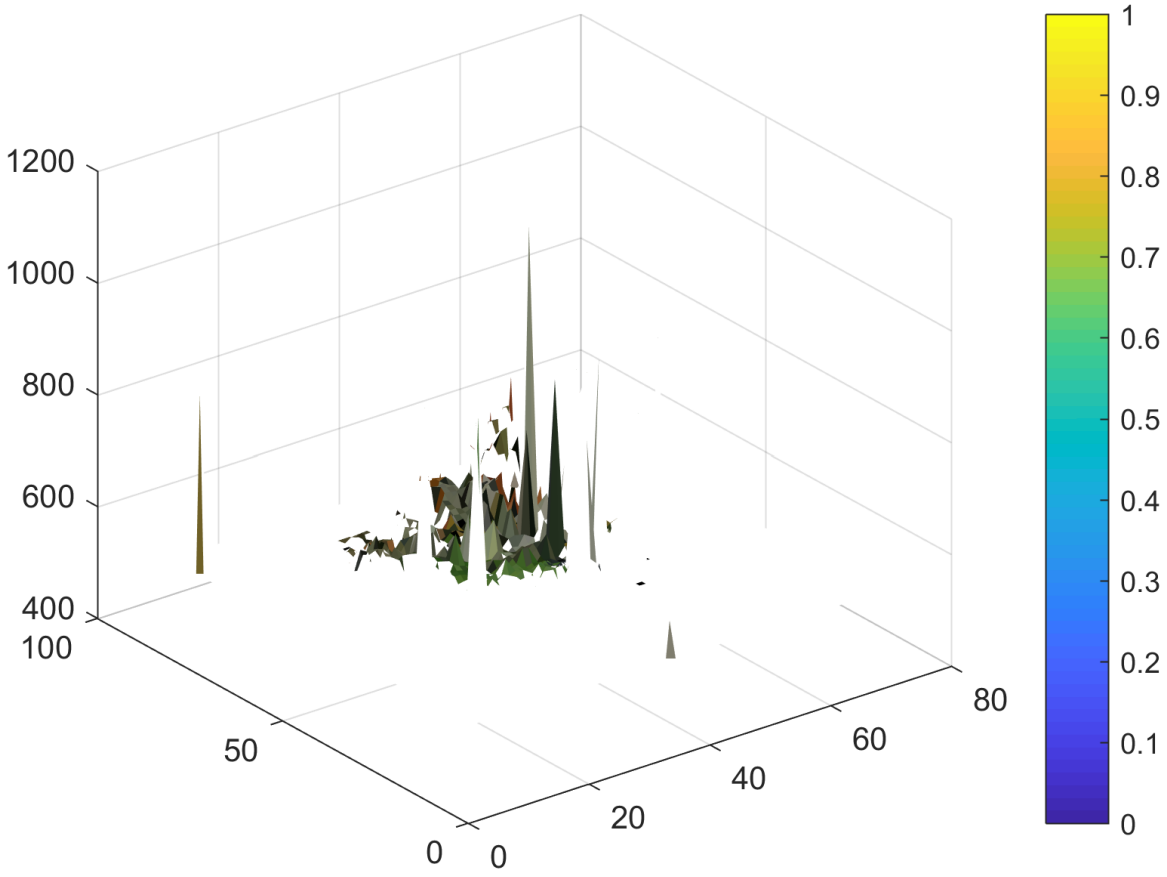


Figure 12: An example of 3D surf by meshgrid

8. How to Evaluate the accuracy of the reconstruction

The most reliable method to evaluate the accuracy of a general 3D reconstruction algorithm is to evaluate by a real world object. This sounds tricky but there are indeed some researches performing such analyses such as described in [5]. There are also some dataset for evaluation that saves the time and money to use some real objects. For example, the dataset described in [2].

If we want to analyze the accuracy only for this castle model, the reprojection error is still a good trade-off between complexity and feasibility. If we reproject the 3D points back into the original 2D images, the distance error

is not the only indicator. There are three metrics proposed by[1] which are signal-to-noise ratio (SNR), the inverse fuzzy image metric (IFIM), and the modified image quality index (Q_m). This test uses calibrated sequence of images captured for the object under reconstruction as ground truth data. Here we didn't have calibrated images, but the principles are still valid. Also we could use the fundamental matrix to estimate the intrinsic matrix as proposed by [11].

In conclusion, the most reliable method is to use the 3D ground truth, either the real object or the dataset with 3D scanning points. The alternative method is to use reprojections with different metrics.

References

[1] A. Eid and A. Farag. On the performance evaluation of 3d reconstruction techniques from a sequence of images. *EURASIP Journal on Advances in Signal Processing*, 2005(13):986306, 2005.

[2] Z.-H. Feng, P. Huber, J. Kittler, P. Hancock, X.-J. Wu, Q. Zhao, P. Koppen, and M. Rätsch. Evaluation of dense 3d reconstruction from 2d face images in the wild. In *Automatic Face & Gesture Recognition (FG 2018), 2018 13th IEEE International Conference on*, pages 780–786. IEEE, 2018.

[3] M. Harker and P. O'Leary. First order geometric distance (the myth of sampsonus). In *BMVC*, pages 87–96, 2006.

[4] R. I. Hartley. In defense of the eight-point algorithm. *IEEE Transactions on pattern analysis and machine intelligence*, 19(6):580–593, 1997.

[5] A. Koutsoudis, B. Vidmar, G. Ioannakis, F. Arnaoutoglou, G. Pavlidis, and C. Chamzas. Multi-image 3d reconstruction data evaluation. *Journal of Cultural Heritage*, 15(1):73–79, 2014.

[6] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[7] K. Mikolajczyk and C. Schmid. Scale & affine invariant interest point detectors. *International journal of computer vision*, 60(1):63–86, 2004.

[8] M. B. Stegmann and D. D. Gomez. A brief introduction to statistical shape analysis. *Informatics and mathematical modelling, Technical University of Denmark, DTU*, 15(11), 2002.

[9] C. Tomasi and T. Kanade. Shape and motion from image streams under orthography: a factorization method. *International Journal of Computer Vision*, 9(2):137–154, 1992.

[10] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. `http://www.vlfeat.org/`, 2008.

[11] A. Whitehead and G. Roth. Estimating intrinsic camera parameters from the fundamental matrix using an evolutionary approach. *EURASIP Journal on Advances in Signal Processing*, 2004(8):412751, 2004.

[12] Wikipedia contributors. Bundle adjustment — Wikipedia, the free encyclopedia, 2018. [Online; accessed 27-June-2018].