

## 浙江大学实验报告

课程名称: 专题研讨 指导老师: Ping TAN

实验项目名称: Project 3: 3D Reconstruction

学生姓名: 顾继庠 专业: 计算机与科学技术 学号: 3150105385

实验日期: 2019 年 5 月 26 日

### 1. Experiment Purpose

In this assignment, there are two programming parts: sparse reconstruction and dense reconstruction. Sparse reconstructions generally contain a number of points, but still manage to describe the objects in question. Dense reconstructions are detailed and fine-grained. In fields like 3D modelling and graphics, extremely accurate dense reconstructions are invaluable when generating 3D models of real world objects and scenes.

#### 1.1 Part 1

In part 1, you will write a set of functions to generate a sparse point cloud for some test images we have provided to you. The test images are 2 renderings of a temple from two different angles. We have also provided you with a mat file containing good point correspondences between the two images. You will first write a function that computes the fundamental matrix between the two images. Then you will write a function that uses the epipolar constraint to find more point matches between the two images. Finally, you will write a function that will triangulate the 3D points for each pair of 2D point correspondences.

#### 1.2 Part 2

In Part 2, we utilize the extrinsic parameters computed by Part 1 to further achieve dense 3D reconstruction of this temple. You will need to compute the rectification parameters. We have provided you with testRectify.m (and some helper functions) that will use your rectification function to warp the stereo pair. You will then use the warped pair to compute a disparity map and finally a dense depth map.

## 2. Experiment Task and Result (Brief description)

### 2.1 Sparse reconstruction

#### 2.1.1 Implement the eight point algorithm

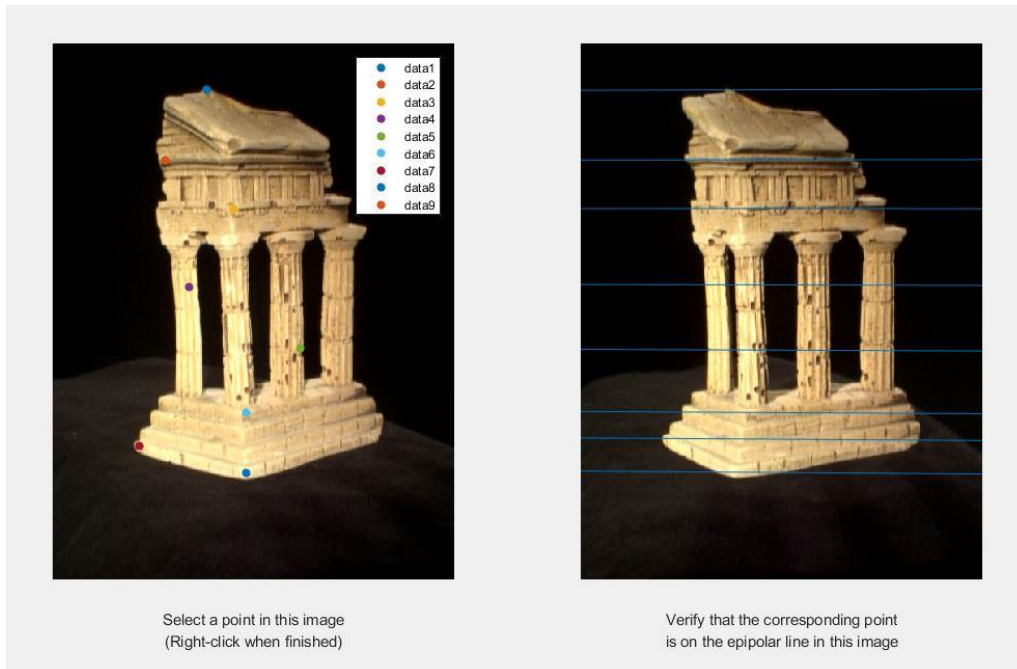
Please use the point correspondences provided in someCorresp.mat.

Write a function with the following form:

function F = eightpoint(pts1, pts2, M)

**#Result:**

$$\mathbf{F} = \begin{bmatrix} -0.0000 & -0.0000 & 0.0000 \\ -0.0000 & 0.0000 & 0.0031 \\ 0.0000 & -0.0030 & -0.0121 \end{bmatrix}$$

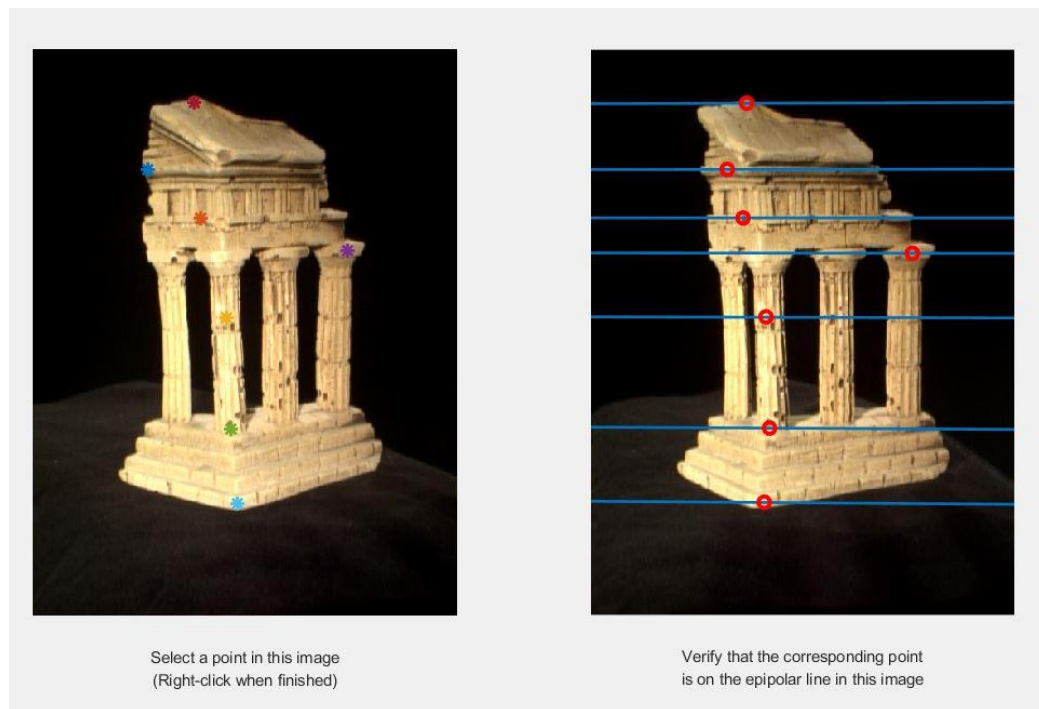


#### 2.1.2 Find epipolar correspondences

Write a function with the following form:

function pts2 = epipolarCorrespondence(im1, im2, F, pts1)

You can use epipolarMatchGui.m to visually test your function. Your function does not need to be perfect, but it should get most easy points correct, like corners, dots etc...

**#Result:**

First the using all the above constructed functions, the epipolar line in the second image was determined. Knowing that in the following images the corresponding point lies in a region fairly close to the corresponding point in image 1, I implemented a search space on the epipolar line that was plus or minus 50 pixels within the range of the original point on the epipolar line. This sped up the process fairly well. Also, a 9 window limit was chosen for the descriptors which were then matched with the corresponding descriptor of the point in the left image. A 9 window seemed to do fairly well as compared to a case of 3 or 5 window range. Also, further increasing the size of descriptor would not show marginal improvement as it did to slow down the implementation. Therefore, a 9x9 window range was chosen. A search along the required range was carried out and the euclidean distance metric was chosen to be the similarity metric between the descriptors.

**2.1.3 Write a function to compute the essential matrix**

In order to get the full camera projection matrices we need to compute the Essential matrix.

So far, we have only been using the Fundamental matrix.

Write a function with the following form:

```
function E = essentialMatrix(F, K1, K2)
```

**#Result:**

$$E = \begin{bmatrix} -0.1539 & -0.0020 & 0.0121 \\ 0.0568 & -1.1031 & 4.5982 \\ -4.6073 & -0.1472 & -0.0014 \end{bmatrix}$$

#### 2.1.4 Implement triangulation

Write a function to triangulate pairs of 2D points in the images to a set of 3D points with the form

```
function pts3d = triangulate(P1, pts1, P2, pts2)
```

**#Result:**

pts1 and pts2 are the  $N \rightarrow 2$  matrices with the 2D image coordinates and P is an  $N \rightarrow 3$  matrix with the corresponding 3D points per row. M1 and M2 are the  $3 \rightarrow 4$  camera matrices.

Once I have implemented a method of triangulation, check the performance by looking at the reprojection error.

#### 2.1.5 Write a test script that uses templeCoords

You now have all the pieces you need to generate a full 3D reconstruction. Write a test script testTempleCoords.m that does the following:

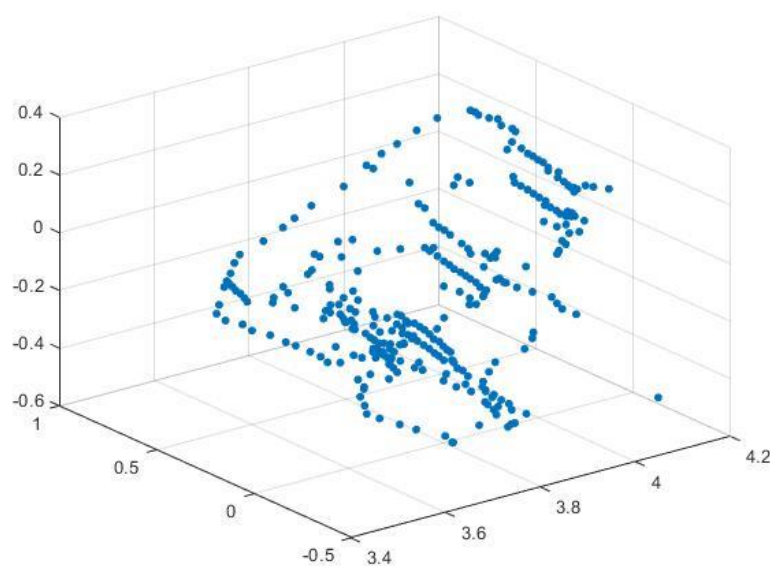
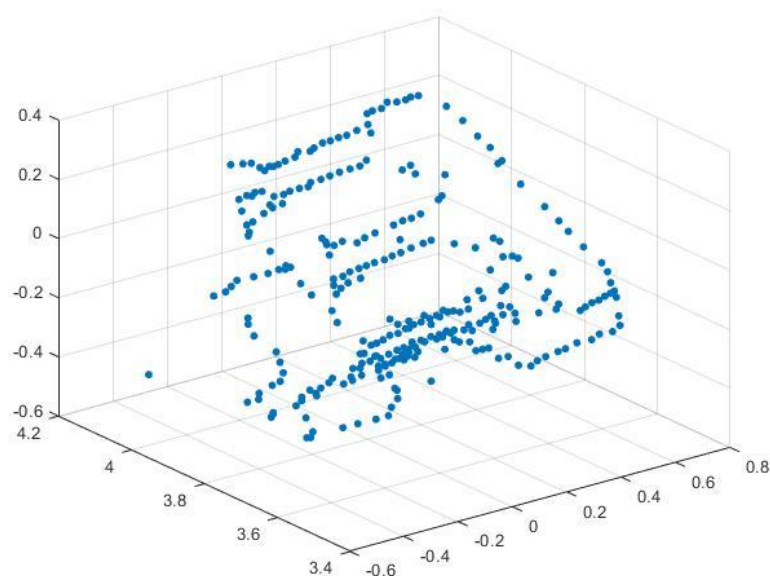
1. Load the two images and the point correspondences from someCorresp.mat
2. Run eightpoint to compute the fundamental matrix F
3. Load the points in image 1 contained in templeCoords.mat and run your epipolarCorrespondences on them to get the corresponding points in image
4. Load intrinsics.mat and compute the essential matrix E.
5. Compute the first camera projection matrix P1 and use camera2.m to compute the four candidates for P2
6. Run your triangulate function using the four sets of camera matrix candidates, the points from templeCoords.mat and their computed correspondences.
7. Figure out the correct P2 and the corresponding 3D points. Hint: You'll get 4 projection

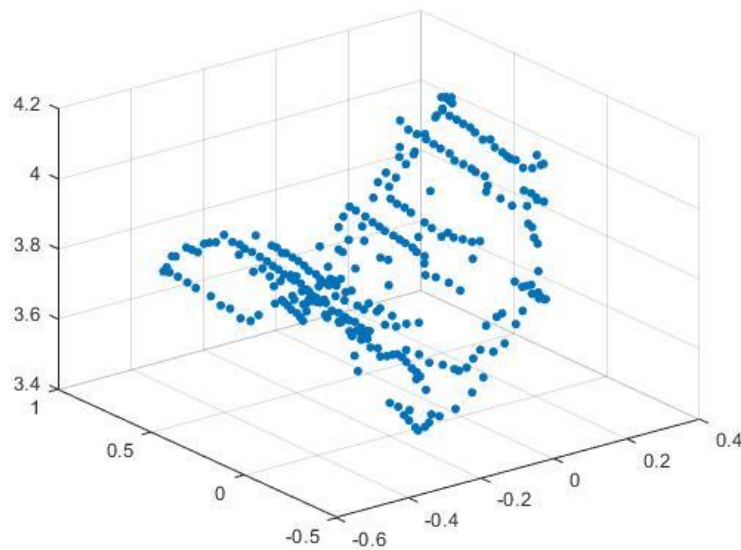
matrix candidates for camera2 from the essential matrix. The correct configuration is the one for which most of the 3D points are in front of both cameras (positive depth).

8. Use matlab's plot3 function to plot these point correspondences on screen

9. Save your computed rotation matrix ( $R_1$ ,  $R_2$ ) and translation ( $t_1$ ,  $t_2$ ) to the file `../data/extrinsics.mat`. These extrinsic parameters will be used in the next section.

### #Result:





## 2.2 Pose estimation

### 2.2.1 Estimate camera matrix $P$

Write a function that estimates the camera matrix  $P$  given 2D and 3D points  $x$ ,  $X$ .

function  $P$  = estimate\_pose( $x$ ,  $X$ ),

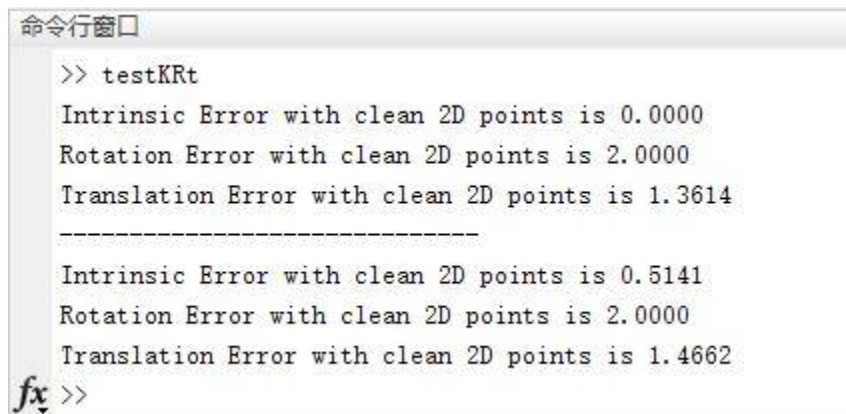
**#Result:**

```
命令行窗口
>> testPose
Reprojected Error with clean 2D points is 0.0000
Pose Error with clean 2D points is 0.0000
-----
Reprojected Error with noisy 2D points is 2.2585
Pose Error with noisy 2D points is 0.2174
fx >>
```

### 2.2.2 Estimate intrinsic/extrinsic parameters

Write a function that estimates both intrinsic and extrinsic parameters from camera matrix.

```
function [K, R, t] = estimate_params(P)
```

**#Result:**A screenshot of a MATLAB Command Window titled '命令行窗口'. It shows the execution of the 'testKRt' function. The output displays intrinsic, rotation, and translation errors for two different sets of clean 2D points. The first set shows an intrinsic error of 0.0000, a rotation error of 2.0000, and a translation error of 1.3614. The second set shows an intrinsic error of 0.5141, a rotation error of 2.0000, and a translation error of 1.4662. The prompt 'fx >>' is visible at the bottom.

```
命令行窗口

>> testKRt

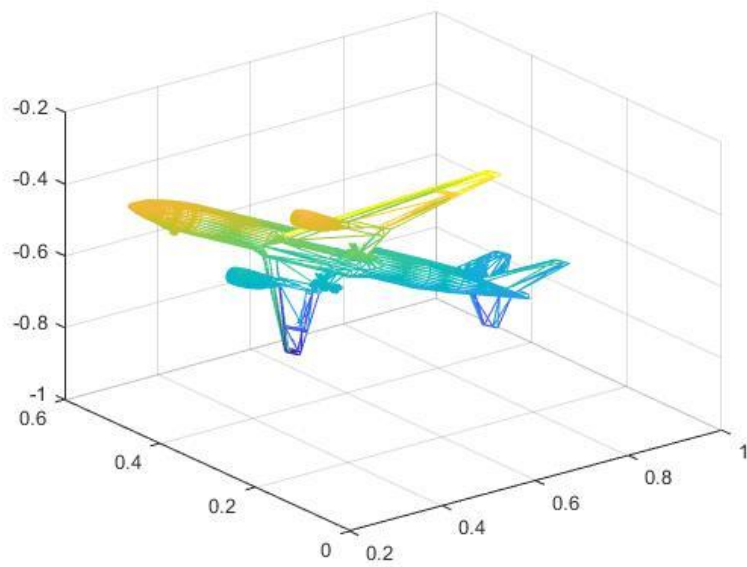
Intrinsic Error with clean 2D points is 0.0000
Rotation Error with clean 2D points is 2.0000
Translation Error with clean 2D points is 1.3614
-----
Intrinsic Error with clean 2D points is 0.5141
Rotation Error with clean 2D points is 2.0000
Translation Error with clean 2D points is 1.4662
fx >>
```

**2.2.3 Project a CAD model to the image**

Write a script projectCAD.m, which does the following:

1. Load an image image, a CAD model cad, 2D points x and 3D points X from PnP.mat.
2. Run estimate\_pose and estimate\_params to estimate camera matrix P, intrinsic matrix K, rotation matrix R, and translation t.
3. Use your estimated camera matrix P to project the given 3D points X onto the image.
4. Plot the given 2D points x and the projected 3D points on screen. An example is shown at the left below. Hint: use plot.
5. Draw the CAD model rotated by your estimated rotation R on screen. An example is shown at the middle below. Hint: use trimesh.
6. Project the CAD's all vertices onto the image and draw the projected CAD model overlapping with the 2D image. An example is shown at the right below. Hint: use patch.

**#Result:**







## 2.3 Dense reconstruction (extra credit)

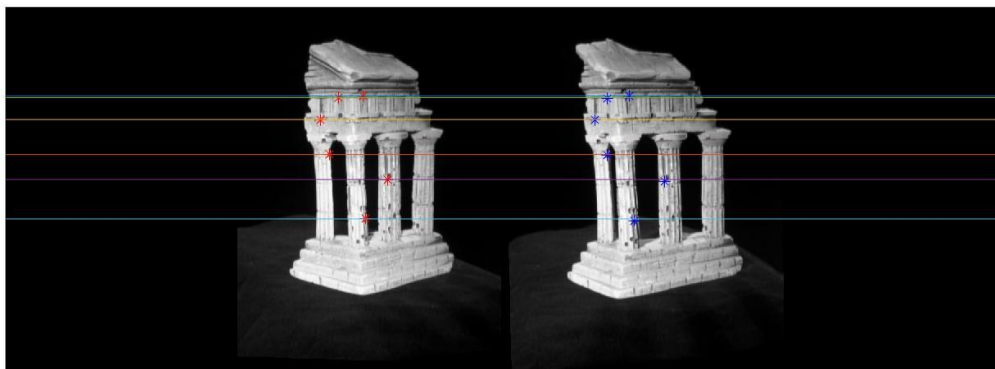
### 2.3.1 Image rectification

Write a program that computes rectification matrices.

```
function [M1, M2, K1n, K2n, R1n, R2n, t1n, t2n] = rectify_pair (K1, K2, R1, R2, t1, t2)
```

This function takes left and right camera parameters ( $K$ ,  $R$ ,  $t$ ) and returns left and right rectification matrices ( $M1$ ,  $M2$ ) and updated camera parameters. You can test your function using the provided script testRectify.m.

**#Result:**



### 2.3.2 Dense window matching to find per pixel density

Write a program that creates a disparity map from a pair of rectified images (im1 and im2).

```
function dispM = get_disparity(im1, im2, maxDisp, windowSize)
```

maxDisp is the maximum disparity and windowSize is the window size. The output dispM has the same dimension as im1 and im2. Since im1 and im2 are rectified, computing correspondences is reduced to a 1-D search problem

**#Result:**

**The code is in matlab file**

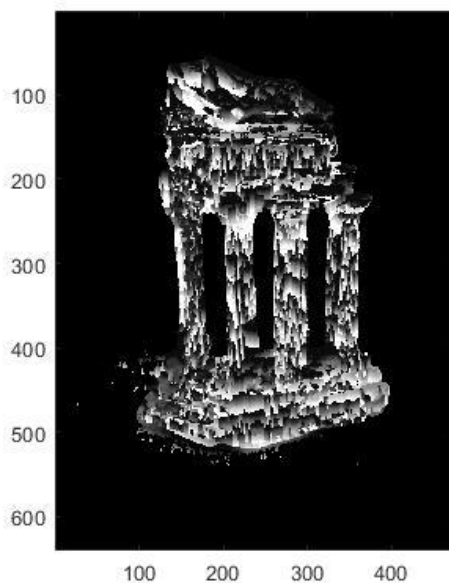
### 2.3.3 Depth map

Write a function that creates a depthmap from a disparity map (dispM).

```
function depthM = get_depth(dispM,K1,K2,R1,R2,t1,t2)
```

**#Result:**

Disparity



## Depth

