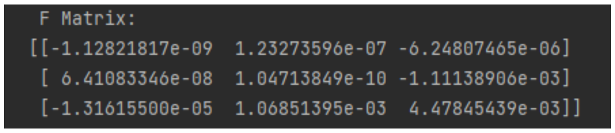
Algorithms in Computer Vision
HW 4

Submitted by:
Nir Sassy: 316491737

Part 1 – Sparse Reconstruction

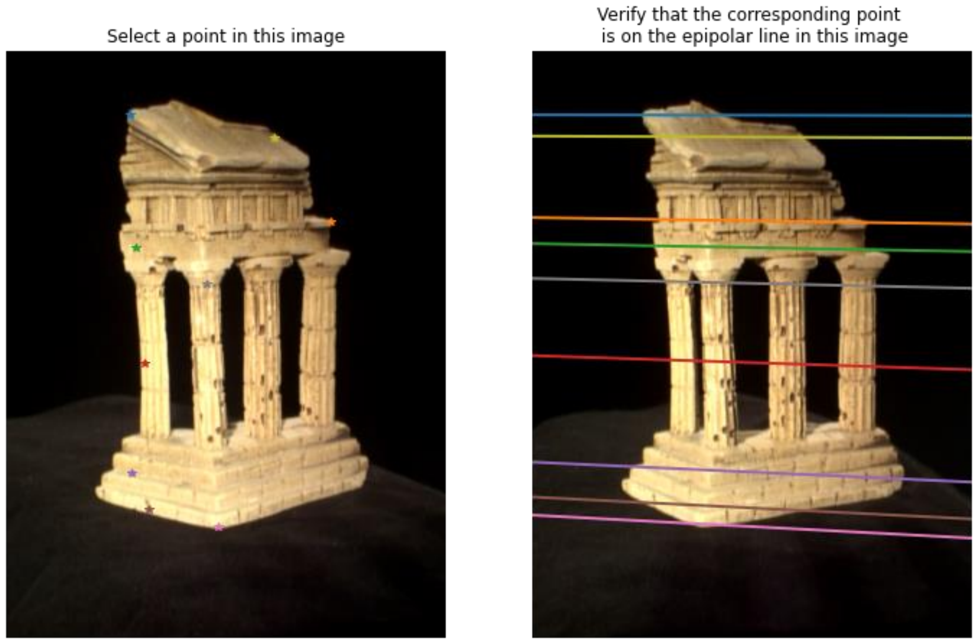
1.1 – Eight Point Algorithm

We implemented the eight-point algorithm by following the steps from the lecture, first we normalize the points to the range [0, 1] using ‘pmax’, then we constructed the matrix A and applied SVD on it, then we took the last column of the V vector , rearranged it to 3x3 to be the F matrix and then enforced the F matrix to be of rank 2 by taking the SVD of it, nullifying the smallest singular value in Σ and then multiplying the matrices again to get F of rank 2. Lastly, we refined (unnormalized) the F matrix by using the given ‘_refineF’ function. Our F matrix on the temple images:



```
 F Matrix:
[[-1.12821817e-09  1.23273596e-07 -6.24807465e-06]
 [ 6.41083346e-08  1.04713849e-10 -1.11138906e-03]
 [-1.31615500e-05  1.06851395e-03  4.47845439e-03]]
```

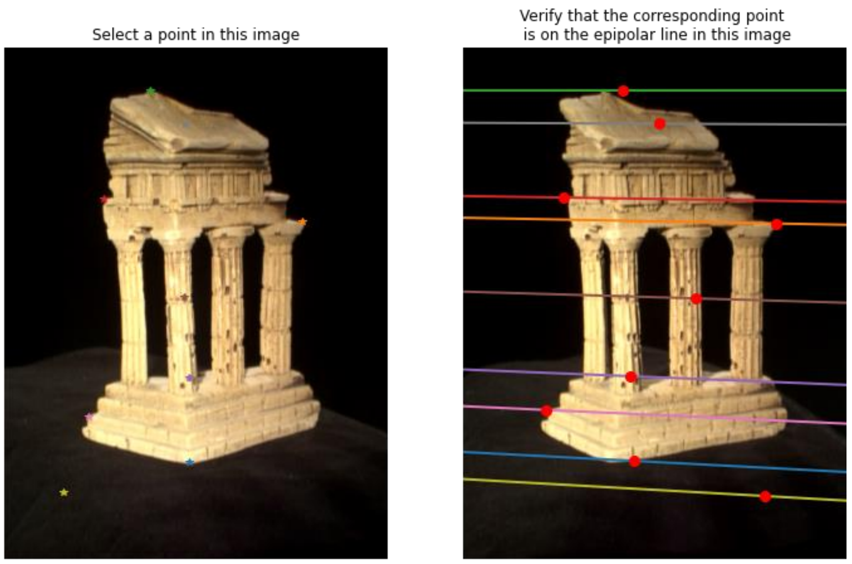
We used the ‘displayEpipolarF’ function to visualize our fundamental matrix correctness:


Select a point in this image
Verify that the corresponding point
is on the epipolar line in this image

We can see that the points selected on the left image are transformed to their corresponding epipolar lines correctly (as all the lines cross the points selected).

1.2 – Epipolar Correspondences

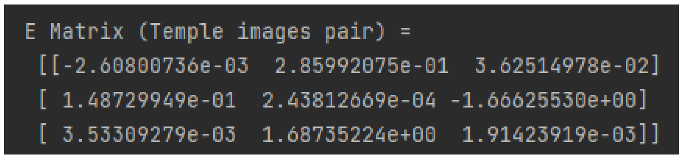
We implemented the function ‘epipolar_correspondences’ by moving along the epipolar line and using the Manhattan distance metric for comparing the patches along the line in the two images. Here are the results, using the ‘epipolarmatchGUI’ function:


Select a point in this image
Verify that the corresponding point is on the epipolar line in this image

Overall, the results are good as we can see that for the most part the points are matched correctly. Errors are mainly where the patch surrounding the point is very similar to other patches across the line (e.g., the yellow point and the gray point), and our Manhattan distance metric fails to pick the correct patch.

1.3 – Essential Matrix

We implemented the 'essential_matrix' based on the fundamental matrix and the intrinsic parameters of the cameras, by simply multiplying them in the correct order ($K_2^T \cdot F \cdot K_1$). Our essential matrix for the temple images is:



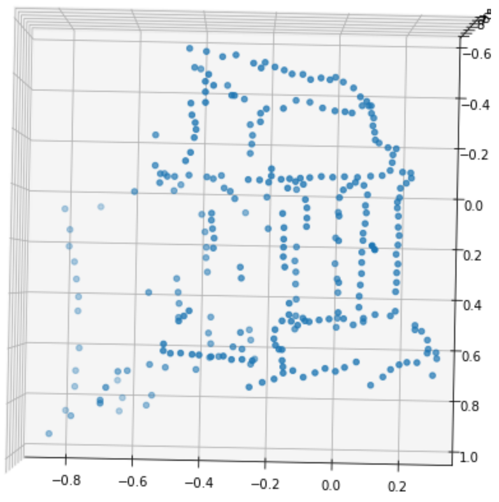
```
E Matrix (Temple images pair) =
 [[-2.60800736e-03  2.85992075e-01  3.62514978e-02]
 [ 1.48729949e-01  2.43812669e-04 -1.66625530e+00]
 [ 3.53309279e-03  1.68735224e+00  1.91423919e-03]]
```

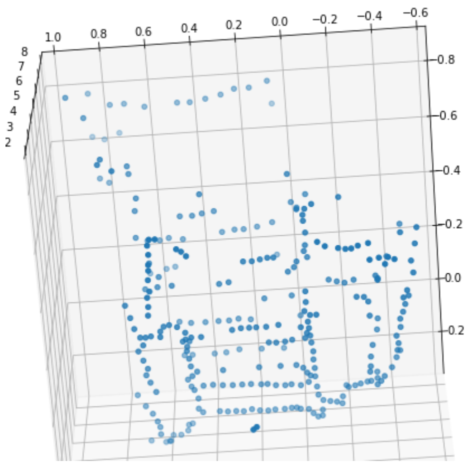
1.4 - Triangulation

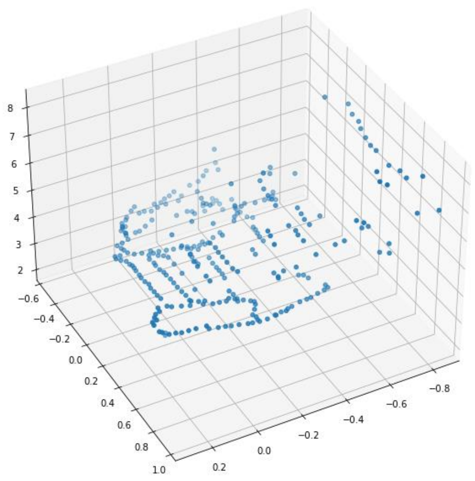
We implemented the triangulation function by constructing the A matrix as showed in the lecture (using the cross product between p and MP where p is the 2d point, M is the camera matrix and P is the 3d point in space) and solving with SVD as usual. We implemented the 'ChooseM' function that receives the 4 options for the M2 camera using the helper function 'camera2'. We used a for loop such that in each iteration, we triangulated the 3D point in space using the first camera matrix (P) and the second camera matrix (P'), both multiplied by their corresponding intrinsic parameters (which also inputted to the function). We then calculated the number of negative Z values for both P and P' and chose the M2 camera for which the sum of negative Z values for both cameras are minimal meaning that they have the maximum 3D points in front of them. We also calculated the reprojection error by reprojecting the 3D points to 2D and calculating the mean Euclidean error between the reprojected points and the original given points. Additionally, we implemented a function that checks the number of points with positive Z values after triangulated with the chosen M2 matrix.

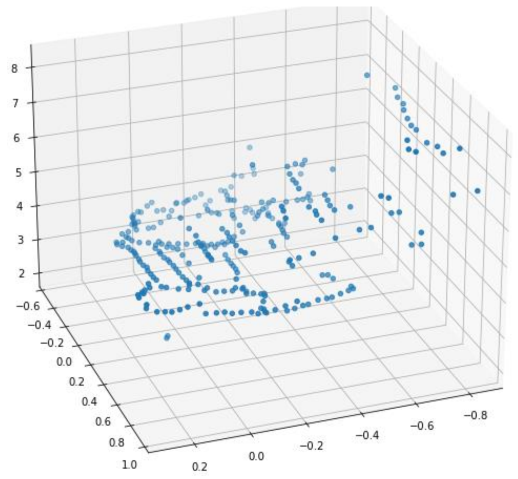
1.5 – Putting it all together:

We wrote a script that reads the images that loads the two images and every other relevant data, calculates the fundamental matrix and essential matrices, setting the M1 Matrix to $[I|0]$ and choosing the correct M2 matrix based on the logic explain in 1.4. We then triangulated the 3D points in space and received the following results:

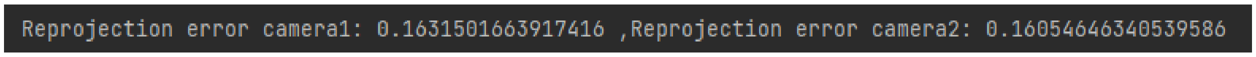








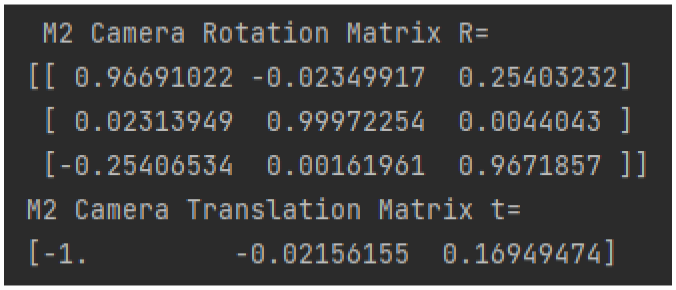
The reprojection errors for camera 1 and 2, calculated in the 'ChooseM' function, are:



```
Reprojection error camera1: 0.1631501663917416 ,Reprojection error camera2: 0.16054646340539586
```

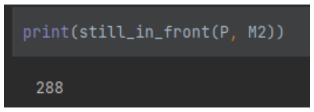
Which is less than 2 pixels.

For camera 2 matrix we got the following rotation and translation matrices:



```
 M2 Camera Rotation Matrix R=
[[ 0.96691022 -0.02349917  0.25403232]
 [ 0.02313949  0.99972254  0.0044043 ]
 [-0.25406534  0.00161961  0.9671857 ]]
M2 Camera Translation Matrix t=
[-1.         -0.02156155  0.16949474]
```

And finally, we checked the number of positive Z values of the triangulated 3D points (using to chosen M2 camera) and received:



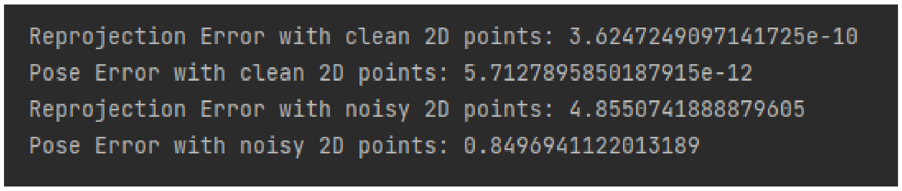
```
print(still_in_front(P, M2))

 288
```

where 288 is the total number of points loaded from the data, meaning that all triangulated points have a positive Z value.

Part 2 – Pose Estimation

2.1 – Estimating M

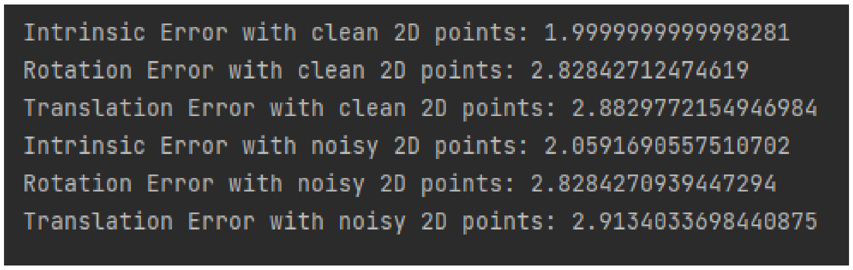
We implemented the function 'estimate_pose' that receives set of 2D and 3D points, and estimating the camera matrix M that connects them. We constructed the A matrix based on the equations seen in the lecture and solved with SVD, took the last column of the V matrix, and reshaped it to shape 3x4. After running the given script for this section, we received the following results:



```
Reprojection Error with clean 2D points: 3.6247249097141725e-10
Pose Error with clean 2D points: 5.7127895850187915e-12
Reprojection Error with noisy 2D points: 4.8550741888879605
Pose Error with noisy 2D points: 0.8496941122013189
```

2.2 – Intrinsic/Extrinsic Parameters

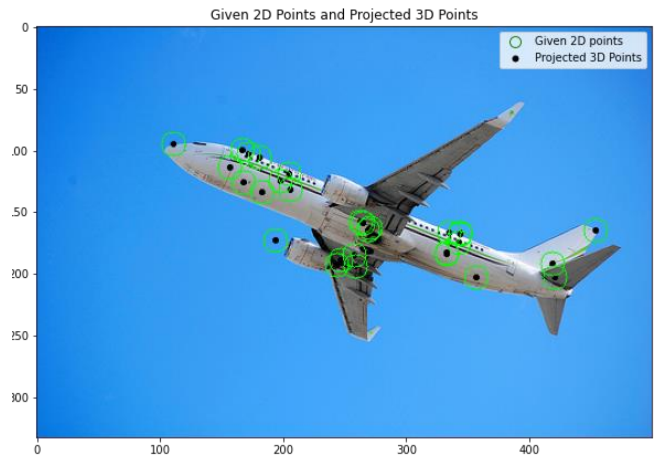
We implemented the function 'estimate_params' that receives the camera matrix M and estimates the camera intrinsic parameters K, and the extrinsic parameters R (Rotation matrix) and t (translation matrix). We followed the instructions directly; We found the null space of M which is the camera center coordinates, found K and R matrices by applying QR decomposition on the N matrix (Where $M = [N|-Nc]$) using scipy's function, finally we calculated the t matrix by applying $t = -R \cdot c$. After running the given script, we received the following results:



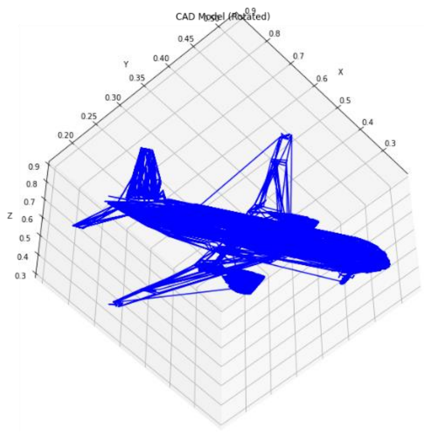
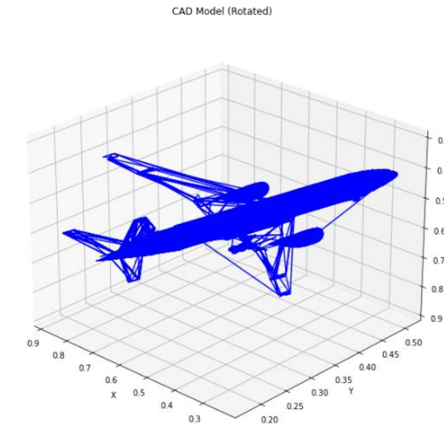
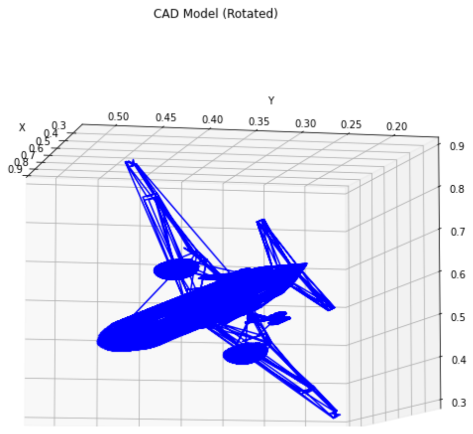
```
Intrinsic Error with clean 2D points: 1.9999999999998281
Rotation Error with clean 2D points: 2.82842712474619
Translation Error with clean 2D points: 2.8829772154946984
Intrinsic Error with noisy 2D points: 2.0591690557510702
Rotation Error with noisy 2D points: 2.8284270939447294
Translation Error with noisy 2D points: 2.9134033698440875
```

2.3 – Projecting CAD models to 2D:

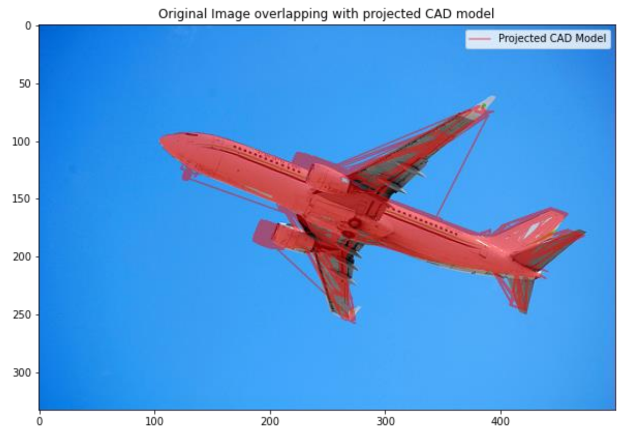
From the given data 'pnp.npz', we loaded $x, X, image, CAD$. We estimated the M matrix using our 'estimate_pose' function with x and X points which are the 2D and 3D points. Then, we calculated K, R, t using the 'estimate_params' function and the M matrix calculated before. We showed the projected 3D to 2D points (using $p = MP$) with the actual 2D points x and got the following results:



Then we loaded the CAD vertices and rotated them using the R matrix calculated before. Here is the CAD model rotated:

Lastly, we projected the CAD vertices to 2D using the M matrix calculated before, and plotted the projected CAD model on the plane image:



And we can see the alignment of the projected CAD model to 2D with the actual plane in the image, meaning our 2D projection worked pretty well.

Thanks!