EE046746: Computer Vision
HW-4
Structure From Motion

Daniela Boguslavsky 315720003
Shahar Rashty 312465305

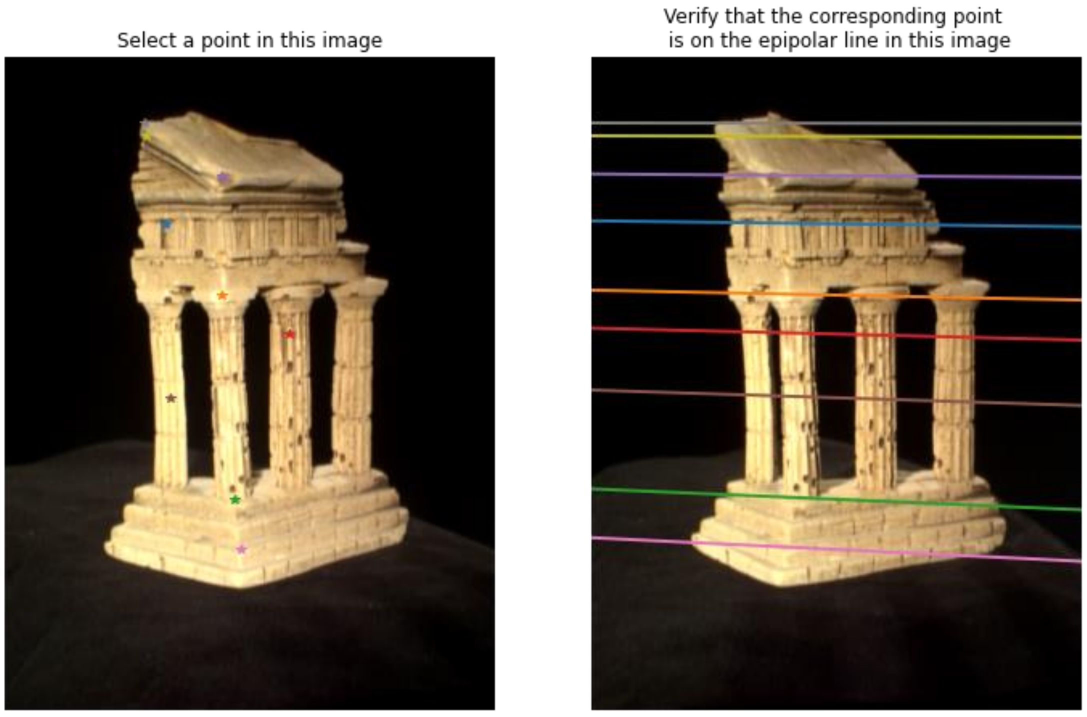
Due date: 6.7.23

Part 1 - Sparse Reconstruction

1.1 Eight Point Algorithm :

The algorithm is used to estimate the fundamental matrix between two images.
We implemented the function eight_point(pts1, pts2, pmax) that takes as input the corresponding points in image 1 (`pts1`) and image 2 (`pts2`), along with `pmax`, a scalar value computed as the maximum of the image height (`H1`) and width (`W1`).
It initializes a normalization matrix `T` that scales the points to be within the range [0, 1] based on `pmax`.
The points `pts1` and `pts2` are normalized by dividing them by `pmax` .
Constructs a matrix `A` as we saw in the lectures.
Using SVD on matrix `A`, the fundamental matrix `F` is estimated by selecting the last column of the V matrix.
Enforcre rank 2 constraint on `F` by performing SVD on `F` and setting the last singular value to zero. We named the resulting `F_rank2`.
The solution is refined by calling a helper function `refineF` with `F_rank2`, `pts1_norm`, and `pts2_norm`.
`F_refined` is un-normalized by multiplying it with the transposed normalization matrix `T` and returning the final fundamental matrix `F`.
We checked the correctness of the estimated F we used the provided function `displayEpipolarF`, which takes in F, and the two images.
To use the GUI we first needed to set a different backhand by :
matplotlib.use('Qt5Agg')

We chose random points and all of them were corresponded correctly to an epipolar line in the second image:


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

1.2 *Epipolar Correpondences*

We implemented the function `epipolar_correspondences(I1, I2, F, pts1)` that takes as input image1 (`I1`), image2 (`I2`), the fundamental matrix (`F`), and points in image 1 (`pts1`) and returns the corresponding points in image 2 (`pts2`).

It initializes an empty list `pts2` to store the corresponding points in image 2.

Generates a list of x-coordinates (`x_list`) for the search along the epipolar line. It covers a range of x-values within image 1, excluding a margin of 15 pixels on each side same as the windows size we chose.

For each point (`pt1`) in `pts1`, the code performs the following steps:

a. Converts `pt1` to homogeneous coordinates.

b. Computes the epipolar line (`epipolar_line`) corresponding to `pt1` by multiplying `F` with `pt1_hom`.

c. Generates a list of y-coordinates (`y_list`) for the epipolar line search using the equation of the line (ax + by + c = 0 => y = (-c - ax) / b). The search is performed for each x-coordinate in `x_list`.

d. Initializes the minimum difference (`min_diff`) as infinity and the corresponding point in image 2 (`min_pt2`) as None.

e. Iterates over each point (`pt2`) in the search range:

- Defines a window size for computing the similarity score between image patches, We tried a few and chose 15X15 as it gave us better results than smaller windows sizes.

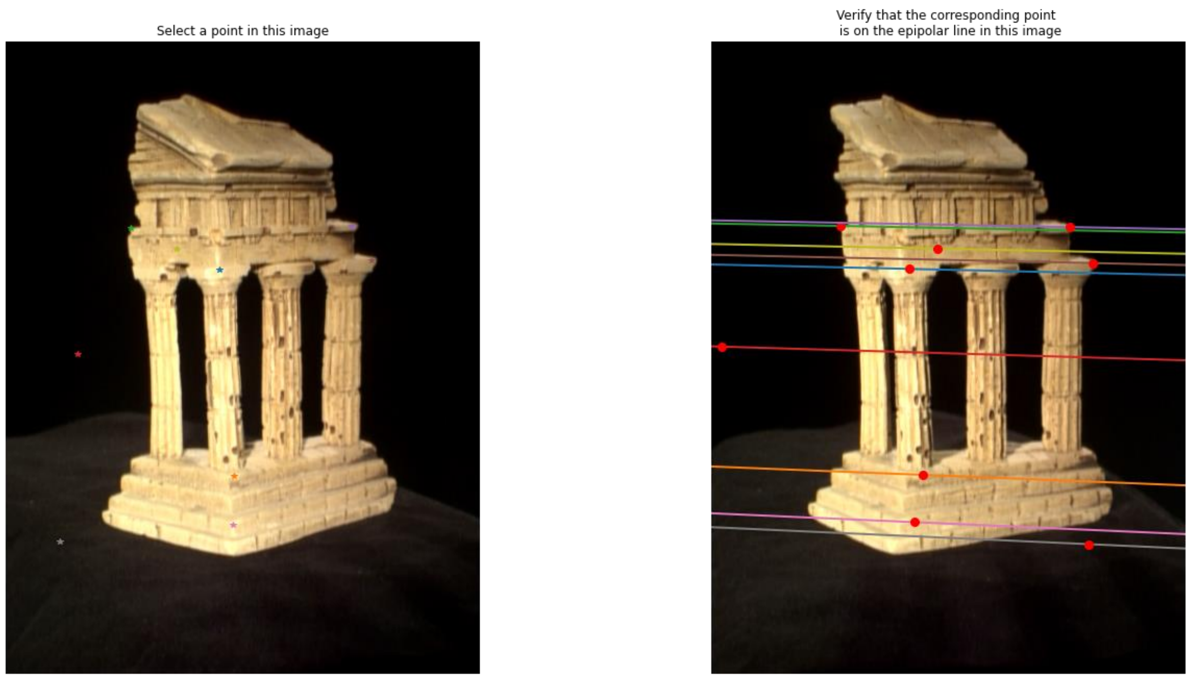
- Extracts image patches centered at `pt1` in image 1 and `pt2` in image 2.

- Computes the similarity score between the two patches using the Manhattan distance, We tried a few settings in order to define the similarity between the two windows,We chose Manhattan distance which gave us better results than L2(Euclidean distance) or L1 norm.

- Updates `min_diff` and `min_pt2` if the computed difference is smaller than the current minimum difference.

f. Appends the corresponding point `min_pt2` (x, y) to the `pts2` list.

WE checked our code using the provided epipolarMatchGUI:


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

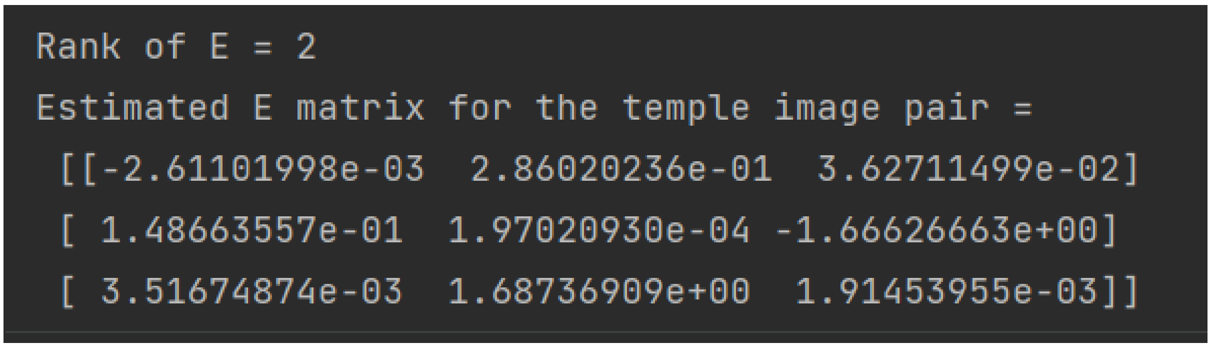
We can see that all the corresponding points are on the epipolar lines, But not all points were matched correctly , we can see that the matching algorithm works well for corners and sharp edges but consistently fails with points where there is no much change in the nerby, for example points on the black/brown background were not matched correctly as can be seen with the red and gray points.

1.3 - Essential Matrix

We implemented essential_matrix(F, K1, K2) That takes the Fundamental matrix computed between two images, `K1` and `K2` are the intrinsic camera matrices for the first and second image respectively and returns E , the essential matrix between the two images which is calculated as :

E = K2' * F * K1 as we saw in the lectures .

We loaded K1 and K2 from the given file and got :


```
Rank of E = 2
Estimated E matrix for the temple image pair =
 [[-2.61101998e-03  2.86020236e-01  3.62711499e-02]
 [ 1.48663557e-01  1.97020930e-04 -1.66626663e+00]
 [ 3.51674874e-03  1.68736909e+00  1.91453955e-03]]
```

1.4 – Triangulation

The function triangulate(M1, pts1, M2, pts2) takes as input camera projection matrix 1 (M1), points in image 1 (pts1), camera projection matrix 2 (M2), and points in image 2 (pts2) and estimate the 3D coordinates of points in space.

it initializes the number of points N as the number of rows in pts1.

If there are no points (N == 0), the function returns an empty 2-dimensional array.

Otherwise, the function initializes a matrix pts3d of size Nx4 to store the 3D points in homogeneous coordinates.

The code iterates over each point (indexed by i) in the range from 0 to N-1:

a.Extracts the x and y coordinates of the corresponding points in image 1 and image 2 (x1, y1, x2, y2).

b. Constructs matrix A using these coordinates to perform linear triangulation as we sat in the lectures.

c. Performs Singular Value Decomposition (SVD) on matrix A and extracts the last column of the V transpose matrix (Vt) as the solution for the 3D point in homogeneous coordinates.

d. Stores the homogeneous coordinates of the 3D point (P) in the pts3d matrix.

The code divides the homogeneous coordinates of the 3D points by the fourth coordinate to convert them to Euclidean coordinates.

Finally, the function returns the Euclidean coordinates of the 3D points as a NumPy array.

Next, we implemented the function **find_best_M2**(M2s, M1, pts1, pts2)

Which takes a set of 4 options for M2 (M2s 3X4X4 matrix) that was returned from the given camera2(E) function, M1 *camera projection matrix 1 (3x4 matrix) and sets of corresponding points* pts1, pts2 it returns the best M2 and print the reprojection errors using the given pts1 and pts2.

1. It initializes the minimum count of 3D points with negative z-values (neg_z_count_min) to a large value, and selects the first matrix in the list (M_selected) as the initial best M2.
2. The code then iterates over each M2 in the list of candidate matrices:

a. Triangulates the 3D points (P3d) using M1 and the current M2 by calling the triangulate function.

b. Counts the number of 3D points with negative z-values by checking if their z-coordinate is less than 0 , This will give us the number of points **in front of camera 1**! since camera1 and Real world coordinate are the same .
To check if the points are in front of camera2 as well we use M2 on the 3D points, to get the projection in camera 2 coordinates , we saved the positive z points as P3d_cam2 and then the final check will be based on np.*sum*(P3d[:, 2] < 0) + np.*sum*(P3d_cam2[2, :] < 0) to make sure we chose the M2 that gave us the most points in front of the two cameras!

c. If the current M2 has fewer 3D points with negative z-values than the previously selected M2, it updates M_selected, neg_z_count_min, and P3d_selected with the current M2, the count of negative z-values, and the corresponding 3D points.

3. The code creates a new matrix P3d_new of size Nx4 and sets its first three columns as the selected 3D points in homogeneous coordinates.
4. It projects the 3D points onto the image plane using M1 by multiplying M1 with the transpose of P3d_new and stores the result in projected.
5. The projected points are converted to non-homogeneous coordinates by dividing the first two rows of projected by the third row.
6. The code calculates the mean reprojection error by calculating the Euclidean distance for each projected points and pts1, and then taking the mean of these distances.
   We projected the projected 3D points onto images planes using M1

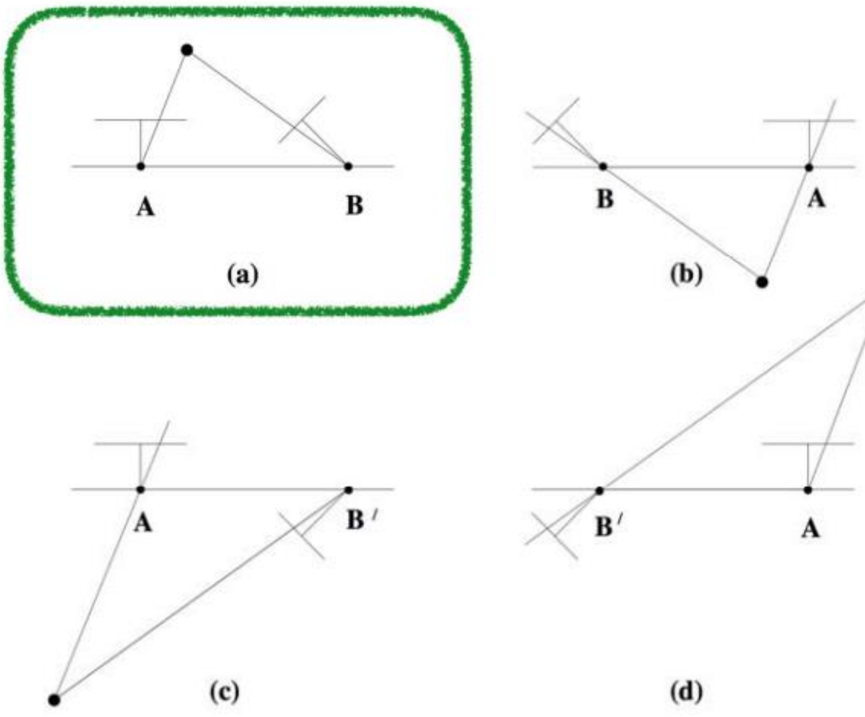
   And M2 and then calculated the mean distance from the projected points and pts1 or pts2 .
7. Finally, the function returns the selected M2 (M_selected).

This function helps us to find the best M2 out of the 4 options, the best one is the one with the most 3D points that are in front of the **two** cameras, meaning their Z value is larger than zero in both cameras coordinates systems.

Camera 1 coordinate system is the same as the real world coordinate system, since we chose R=I and t=0, for checking if the points are in front of camera 2, meaning they have Z>0 in camera 2 coordinate system we implement helper function :

check_points_in_front_of_camera(points_3d, M2)

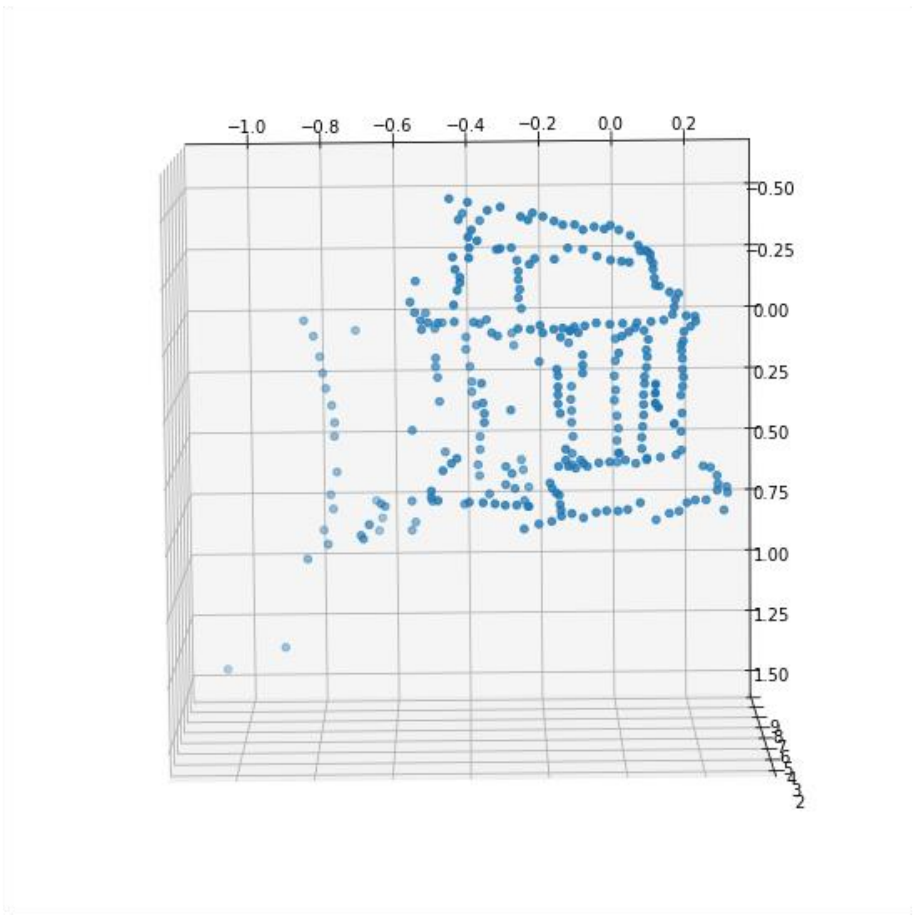
It Applies the the extrinsics of camera 2(chosen M2 matrix) to the recovered 3D points and checks that they are still in front of camera 1, M1 was chosen to be (I|0) so the real world coordinate and M1 coordinate system are the same one.

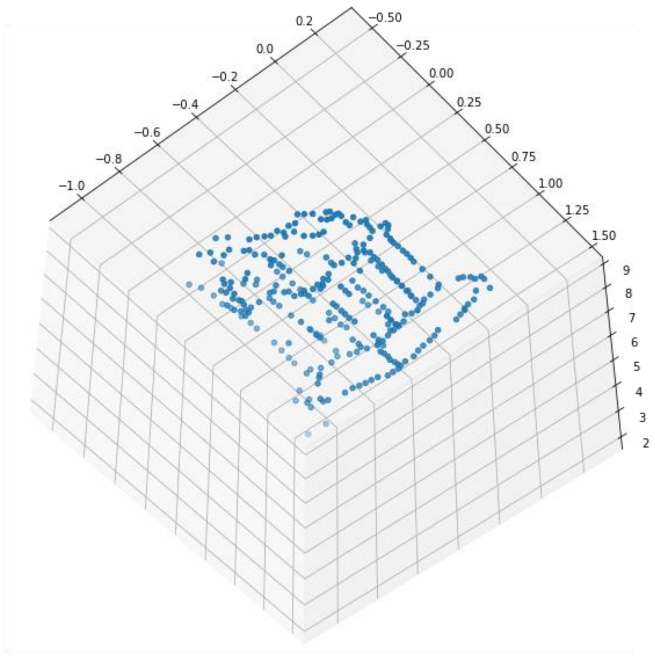


1.5 - Putting It All Together

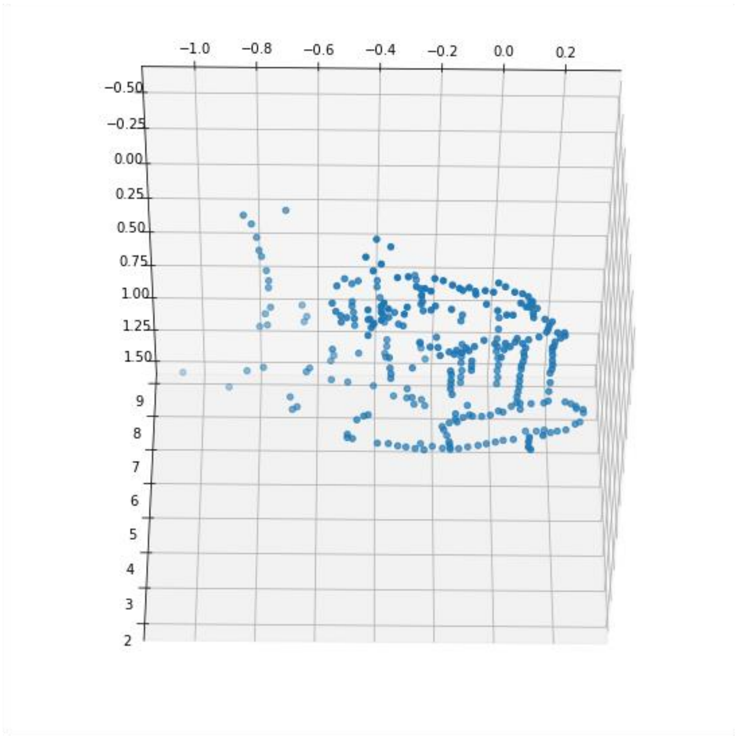
Loaded the two images and the point correspondences from `data/some_corresp.npz`.
* Ran eight point to compute the fundamental matrix `F`.
* Loaded the points in image 1 contained in `data/temple_coords.npz` and ran `epipolar_correspondences` on them to get the corresponding points in image2.
* Loaded `data/intrinsics.npz` and compute the essential matrix `E`.
* Computed the first camera projection matrix `M1` and use `camera2` to compute the four candidates for `M2`.
* Ran your `triangulate` function using the four sets of camera matrix candidates (after scaling with the intrinsics), the points from `data/temple_coords.npz`, and their computed correspondences.
*To figure out the correct `M2` and the corresponding 3D points we used the find_best_M2(M2s, M1, pts1, pts2) we implemented.
* Finally used matplotlib's scatter function to plot these point correspondences on screen.

Our final reconstruction results from 3 angles:





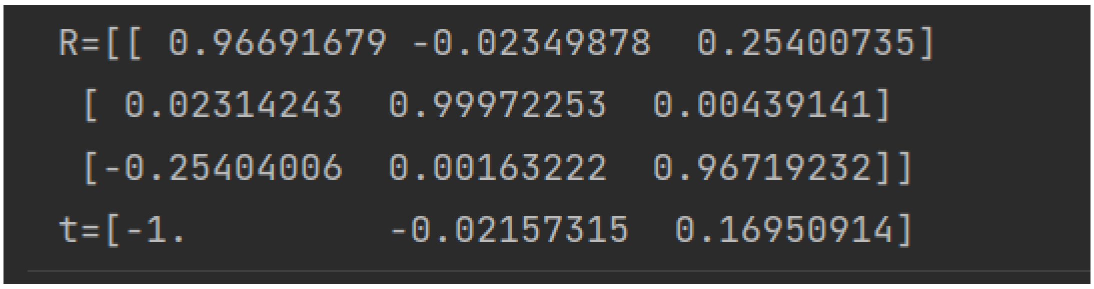
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
1.50
9
8
7
6
5
4
3
2



−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
1.50
9
8
7
6
5
4
3
2

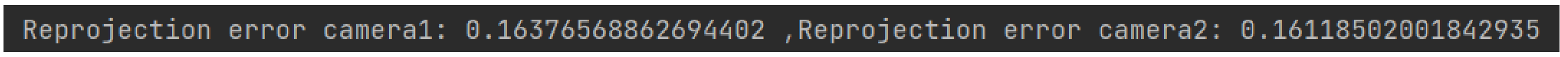
For the first camera we chose R1=Identity matrix and t1=[0,0,0]

For camera2,we got R2 and t2 :



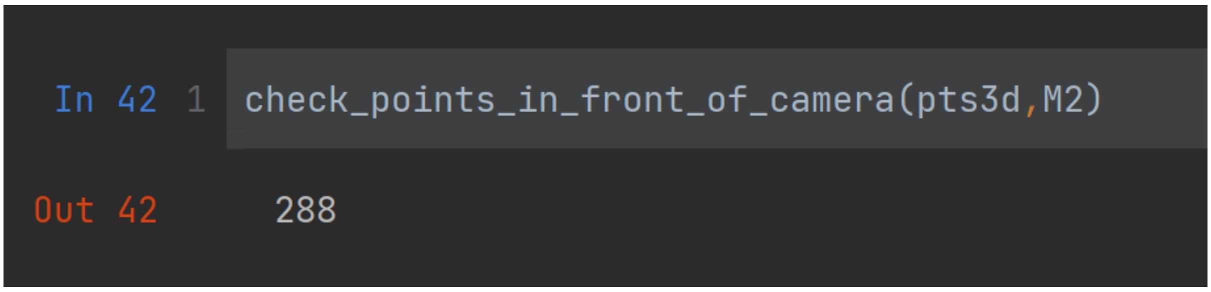
```
R=[[ 0.96691679 -0.02349878  0.25400735]
 [ 0.02314243  0.99972253  0.00439141]
 [-0.25404006  0.00163222  0.96719232]]
t=[-1.         -0.02157315  0.16950914]
```

The reprojection errors :



```
Reprojection error camera1: 0.16376568862694402 ,Reprojection error camera2: 0.16118502001842935
```

And we checked the number of points that are in front of the cameras with our helper function :



```
In 42   check_points_in_front_of_camera(pts3d,M2)

Out 42  288
```

In deed all points are In front of both cameras.

Part 2 - Pose Estimation

2.1 - Estimating M:

We implemented estimate_pose(p, P) that takes `p` is 2xN matrix denoting the (x, y) coordinates of the N points on the image plane and `P` is 3xN matrix denoting the (x,y,z) coordinates of the corresponding points in the 3D world.

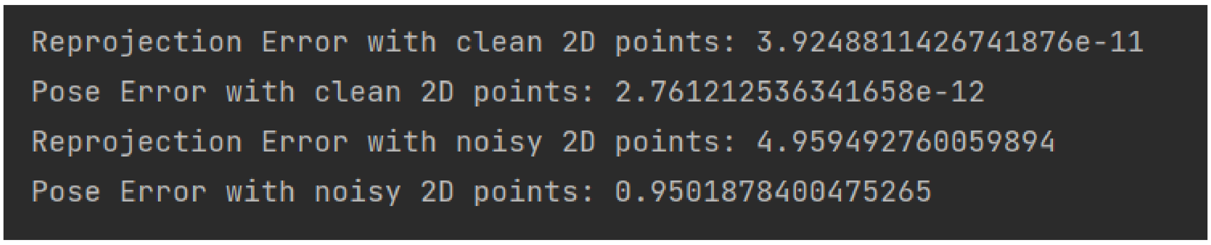
we first assert that the input matrices p and P have the correct dimensions. Then, we create an empty matrix A of size (2N, 12) to store the linear equations.

Next, we iterate over each point and fill in the corresponding rows of A with the appropriate values. The linear equations are constructed based on the DLT method, similar to homography estimation.

After constructing A, we perform Singular Value Decomposition (SVD) on A using np.linalg.svd. The camera matrix M is then extracted from the last column of the right singular matrix V and reshaped to a (3, 4) matrix.

Finally, we return the estimated camera matrix M

We ran the provided test script and got :



```
Reprojection Error with clean 2D points: 3.9248811426741876e-11
Pose Error with clean 2D points: 2.761212536341658e-12
Reprojection Error with noisy 2D points: 4.959492760059894
Pose Error with noisy 2D points: 0.9501878400475265
```

2.2 - Intrinsic/Extrinsic Parameters

Implemented estimate_params(M)

It takes *camera matrix (3x4 matrix) M , extract from it c= camera center using SVD and K and R using QR decomposition . K ,intrinsic matrix ,is Upper triangular matrix and rotation R is Orthonormal matrix.*

It ccomputes the translation t by multiplying the camera center with the rotation R.

We ran the provided test script :

```
Intrinsic Error with clean 2D points: 2.000000000000675
Rotation Error with clean 2D points: 2.8284271247461903
Translation Error with clean 2D points: 7.897289927533791
Intrinsic Error with noisy 2D points: 2.174641268730304
Rotation Error with noisy 2D points: 2.8284271128093366
Translation Error with noisy 2D points: 7.755849954919463
```

2.3 - Projecting CAD Models to 2D

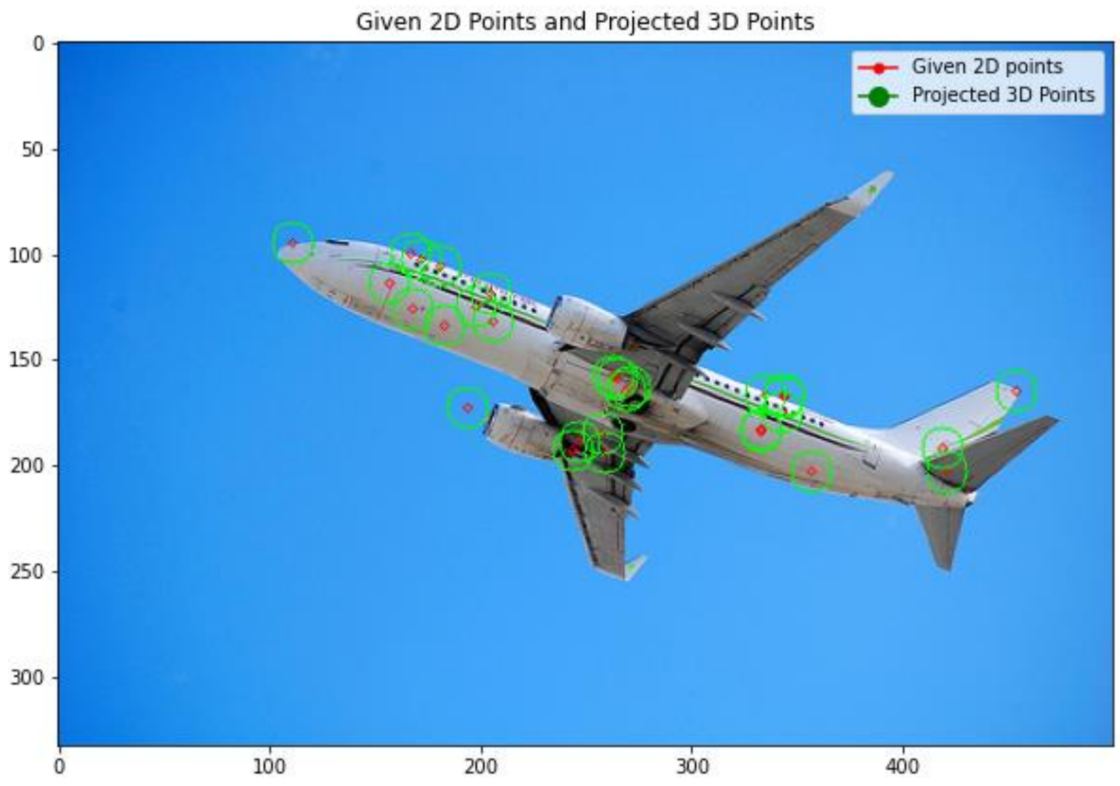
We loaded X,x,image,cad

Estimated M by X and x

Calculated K, R, t with estimate_params(M)

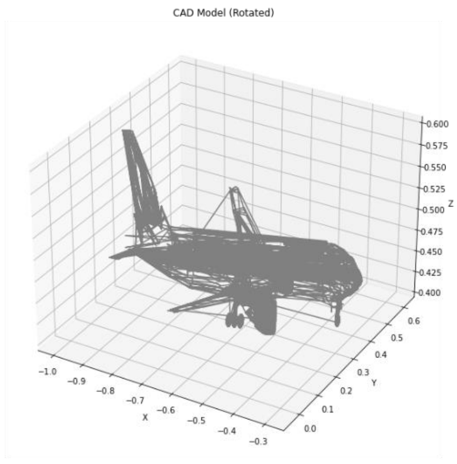
projected 3D points onto the image using camera matrix M

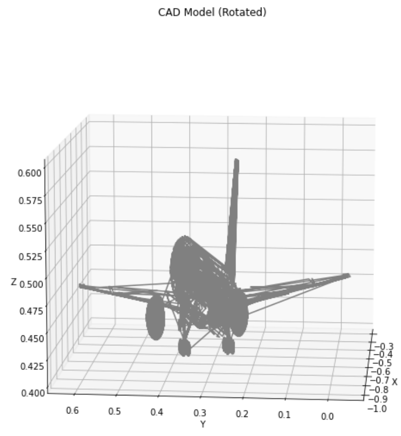
the draw the projected points and the 3D ones :

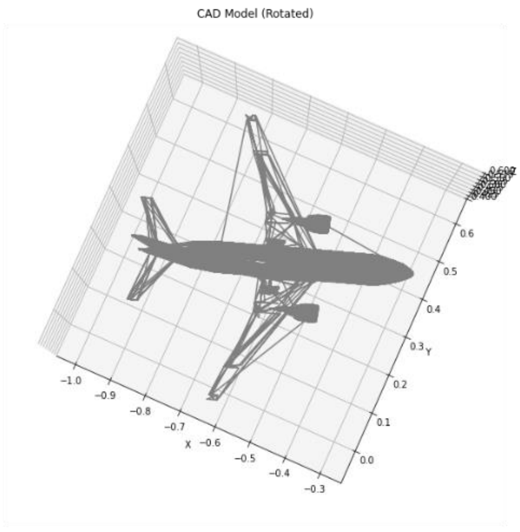


We Extracted the CAD model vertices and faces

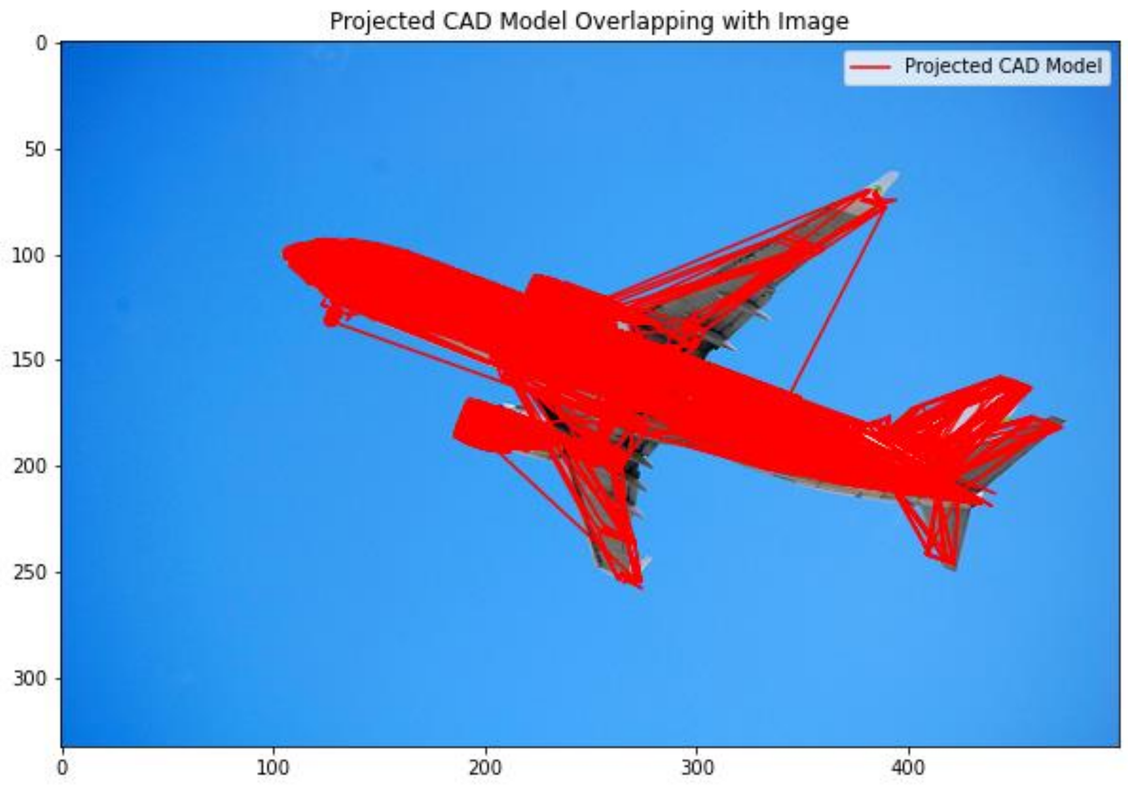
Rotate the CAD model vertices with R and got :







Finally, we projected the CAD model vertices onto the image using camera matrix M:


Projected CAD Model Overlapping with Image
Projected CAD Model