The Implementation of the Forward Pass of a Convolutional Neural Network for Object Recognition

Isabella Salvi
Alexandria Eicher
Kritika Senthil
Osheen Arya

CMPEN 454, Fall 2020
Department of Electrical Engineering and Computer Science
Pennsylvania State University
08 November 2020

Summary

This project allows us to look at implementing forward and inverse camera projections. The goal is to perform triangulation from two cameras in order to reconstruct from pairs of 2D points to matching 3D points. Using knowledge about the pinhole camera from lectures and materials provided in class, this project allows us to understand how 2D image coordinates and 3D world coordinates are related to each other and the various transformations that occur. Therefore, this provides context for our project. We essentially need to use sets of 3D joint locations from a human body which is then converted into 3D viewing rays. This is where triangulation comes into play, the viewing rays are used to recover the original 3D coordinates that were originally started with.

In terms of tasks that need to be performed, we started with reading the 3D join data and the camera parameters. After thoroughly parsing through the datasets, we started actually projecting the 3D points into the 2D pixel coordinates. After this we went into the triangulation aspect of the project in order to get it back into a set of 3D scene points.The last fews steps included measuring errors and computing the epipolar lines between the two views provided to us. From this project, we expect to create an algorithm that allows us to achieve accurate quantitative results in terms of statistics from the distance data. We also expect to be able to visualize each step of the data processing. Our deliverables include a fully functional operation program and a written report.

Outline of Procedural Approach

In order to implement forward and inverse camera projection and to perform triangulation from the two cameras correctly, our team needed to design a model that features subroutines controlled by a main routine. The subroutines within our model are functions that perform projection of 3D world coordinates into 2D pixels, reconstruction of 3D coordinates from those same 2D pixels, error measurement between the original 3D coordinates and the 3D coordinates

that we projected, and computation of epipolar lines between the two different views for each frame number. Figure 1 shows how our main function calls each subroutine. When the main function is run, the videos, camera parameters, and joint data is loaded into the function. The main function iterates through each frame in the joint data. Once 3D world coordinates are parsed from the joint data, the main function calls the subroutine *project3DTo2D* with each set of camera parameters on the set of 3D world coordinates. Returned back to main is a set of 2D pixels that are then plotted onto an image of the frame for each camera. The main function then calls the *reconstruct3DFrom2D* function with the set of 2D pixel values, and receives a set of 3D coordinates. Using the set of 3D coordinates that we reconstructed and the original set of 3D coordinates, main calls a measureError function to calculate the error difference between the two sets of 3D coordinates. After the main function reports the error information that it received, it calls *findEpipolarLines*. Main receives epipolar lines for each of the camera views and plots those lines on the frame image for each of the videos.

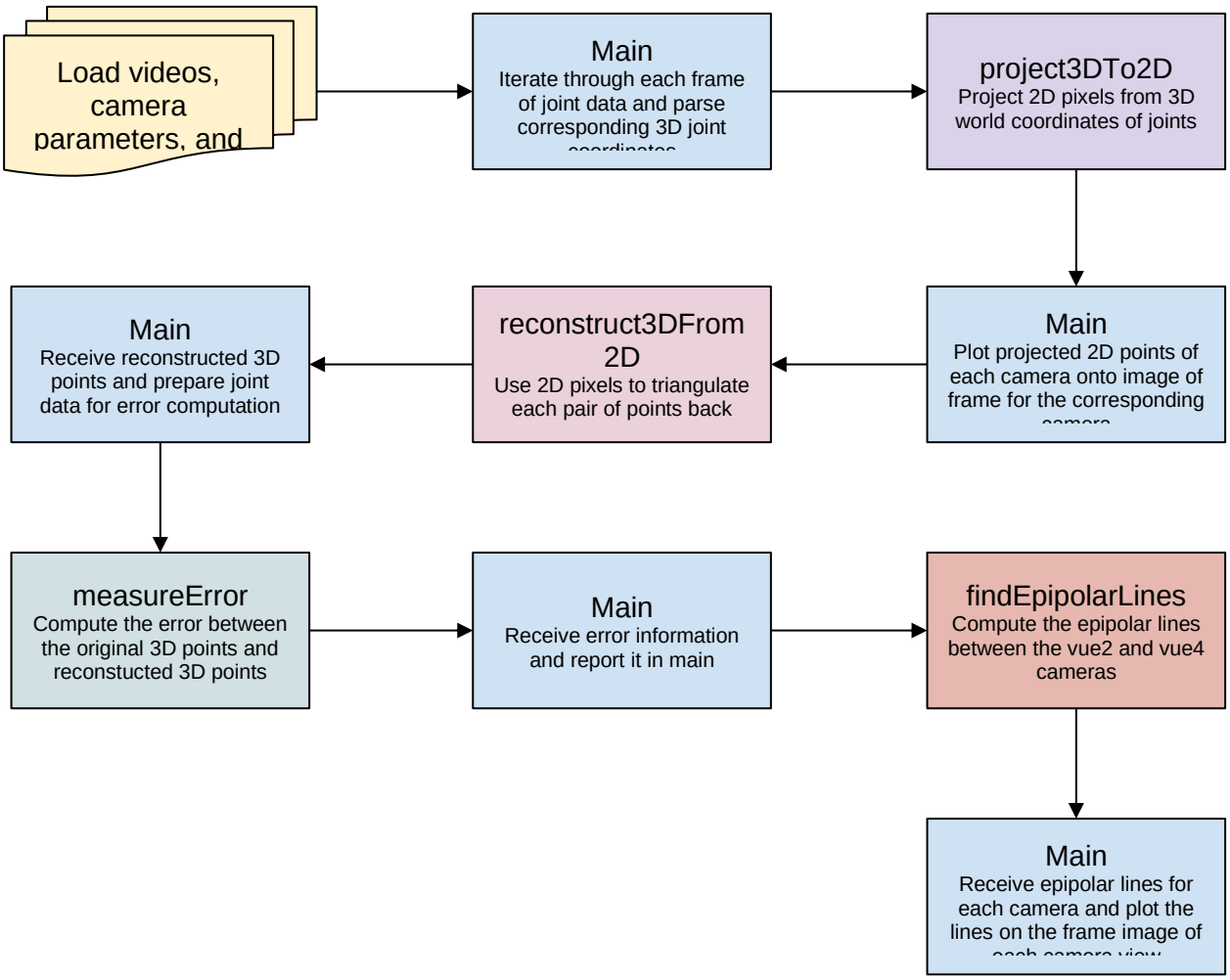


Figure 1: Flow chart showing the flow of control and subroutine structure of Matlab code.

Detailed in this section of the report are more information on each of our subroutines. The next subsection presents an interpretation of the 3D joint data. In the second subsection, we discuss our method for parsing camera parameters. Explained in the third subsection is our team's method for projecting 3D points into 2D pixels. On the other hand, the fourth subsection details how we took those 2D pixels and triangulated them back into 3D coordinates. Additionally, the fifth subsection presents the error calculation between the given 3D coordinates and the 3D coordinates that our team calculated. Our final subsection shows how we compute epipolar lines for each camera view and plot them on each frame image for each camera.

Reading the 3D Joint Data

The first step in this project was reading the 3D joint data provided to us. In order to understand, we extracted the x, y and z locations for the joints in a specific mocap frame. There are 12 joints for each location. Below are the screenshots of the values of each of the joints for every location.

x location:

1x12 double

	1	2	3	4	5	6	7	8	9	10	11	12
1	1.8312e...	1.8610e...	1.6514e...	1.5724e...	1.6024e...	1.4257e...	1.7544e...	1.7198e...	1.8881e...	1.6515e...	1.5016e...	1.5605e...

y location:

1x12 double

	1	2	3	4	5	6	7	8	9	10	11	12
1	1.6299e...	1.6115e...	1.7063e...	1.3033e...	1.2738e...	1.4122e...	1.5302e...	1.6410e...	1.5446e...	1.3967e...	1.4383e...	1.3026e...

z location:

1x12 double

	1	2	3	4	5	6	7	8	9	10	11	12
1	1.3225e...	1.0279e...	882.5990	1.3384e...	1.0403e...	867.0510	849.6600	435.1120	63.6316	836.6420	465.2640	47.9149

Reading Camera Parameters

For tackling section 2.2, we first loaded all the data into matlab in order to see all the parameters for vue2 and vue4. In order to choose the camera parameters that were needed, we looked at all the data and mapped it to the parameters based on the information provided in Lectures 13 and 14. First we figured out the fields in regards to the pinhole camera model parameters for vue2:

- Internal Parameters: Consists of Kmat

vue2.Kmat

	1	2	3
1	1.5578e...	0	976.0397
2	0	1.5578e...	562.8225
3	0	0	1

- External Parameters: Consists of entries in Pmat

vue2.Pmat

	1	2	3	4
1	-0.7593	-0.6491	0.0468	137.7154
2	-0.1381	0.0904	-0.9863	805.5227
3	0.6360	-0.7553	-0.1583	7.3365e...

- Internal Parameters that combine to form Kmat: Consists of focal length and prinpoint

vue2.prinpoint

	1	2
1	976.0397	562.8225

vue2.foclen

	1
1	1.5578e...

- External Parameters that combine to form Pmat: Consists of Rmat and position

vue2.Rmat

	1	2	3
1	-0.7593	-0.6491	0.0468
2	-0.1381	0.0904	-0.9863
3	0.6360	-0.7553	-0.1583

vue2.position

	1	2	3
1	-4.4501e...	5.5579e...	1.9491e...

- Location of Camera: Consists of the last column in Pmat - indicates T values

4
137.7154
805.5227
7.3365e...

- Verification of Pmat: If you look at the Pmat values, it is the combination of the Rmat values and the position values given to us by the multiplication of these values using transposition of the R matrix as well.

Then we went on to figure out the fields in regards to the pinhole camera model parameters for vue4:

- Internal Parameters: Consists of Kmat

vue4.Kmat

	1	2	3
1	1.5185e...	0	975.0173
2	0	1.5185e...	554.8964
3	0	0	1

- External Parameters: Consists of entries in Pmat

vue4.Pmat

	1	2	3	4
1	-0.8164	0.5767	0.0300	388.7687
2	0.1040	0.1979	-0.9747	295.0609
3	-0.5680	-0.7927	-0.2215	7.2830e...

- Internal Parameters that combine to form Kmat: Consists of focal length and pinpoint

vue4.foclen

	1
1	1.5185e...

vue4.prinpoint

	1	2
1	975.0173	554.8964

- External Parameters that combine to form Pmat: Consists of Rmat and position

vue4.Rmat

	1	2	3
1	-0.8164	0.5767	0.0300
2	0.1040	0.1979	-0.9747
3	-0.5680	-0.7927	-0.2215

vue4.position

	1	2	3
1	4.4236e...	5.4904e...	1.8890e...

- Location of Camera: Consists of the last column in Pmat - indicates T value

4
388.7687
295.0609
7.2830e...

- Verification of Pmat: If you look at the Pmat values, it is the combination of the Rmat values and the position values given to us by the multiplication of these values using transposition of the R matrix as well.

Projecting 3D Points into 2D Pixel Locations

To project 3D points into 2D pixel locations, we created a subprocess that is called by the main function. The function *project3DTo2D* takes a camera structure composed of parameters and a 3 x 12 array of world coordinates. The first dimension of the array corresponds to the x-coordinate, while the second and third dimensions of the array correspond to the y and z-coordinates. Moreover, the array represents x, y, and z-coordinates for 12 joints on a particular frame. In the functions, the first thing that we do is extract the K and P matrices from the camera parameters. Then, we create a 4 x 1 array by adding a fourth dimension of ones onto the array of world coordinates that was given as input. Using the pinhole camera model, we then multiply the newly created matrix of world coordinates by the K and P matrices to get a new set of points. Finally, each of the 12 x and y-coordinates in the new set of points are divided by the z-coordinate of that joint to get the final 2D pixel locations. The array of 2D coordinates is returned to the main function after adding on a third dimension of ones to the array.

An additional aspect to this part of the project is that we can validate if we are getting the correct 2D points from the world coordinates. To validate the pixel locations, we can plot the x-y pair onto an image of that particular frame. After the 2D pixel values are returned from the *project3DTo2D* function, the main function plots the 2D points onto the frame image for each camera (vue2 and vue4). In other words, there will be two figures that are shown for each frame

iteration: one showing the 2D joint locations from the perspective of the vue2 camera, and one showing the 2D joint locations from the perspective of the vue4 camera.

Triangulation Back into a Set of 3D Scene Points

The purpose for the subroutine *reconstruct3DFrom2D* in the program is to take the 2D points that we calculated in the previous function and triangulate them back into 3D coordinates. To do this, we relied mainly on lecture 15 for the class. In lecture 15, we learned the entire process on how to triangulate a 2D pixel into a 3D point. The equation from lecture 15,

$P_w = R^T K^{-1} P_u - R^T T$,

was our primary guide in developing the code for this subroutine. We needed to use this equation twice: once for the vue2 camera and corresponding pixel coordinates and another time for the vue4 camera and corresponding pixel coordinates. The first thing we did when writing this function was initialize all of the variables from equation 1. We also needed to initialize the variables for the vue2 and vue4 cameras separately because the two structs contain different information. We derived R^T and K^{-1} by taking R matrices and P matrices from each camera structure and applying a transpose and an inverse transformation on the matrices, respectively. P_u is the 2D point that we are transforming back into a 3D coordinate. The final variable, T, was calculated by multiplying the R matrix for each camera by a 3 x 1 matrix of the camera coordinates.

After all variables are initialized, the program loops through each of the twelve 2D points in each input array. For the sake of clarity here and in the actual function, let Ci be equal to $R^T T$ and let Vi be equal to $R^T K^{-1} P_u$. Once we compute C and V for each camera using the variables that we initialized at the beginning, we compute the unit vector for V1 and V2. Then, we create a third vector, V3, created from the cross product of the first two unit vectors divided by the normalization of that cross product. From here, we will use three unknown variables, a, b, and d, which will be the distances from C1 to P1, C2 to P2, and P1 to P2, respectively. We can compute these unknowns by creating a 3 x 3 matrix, Ax = b. Each unit vector makes up a column of A, while each x, y, and z value of C1 is subtracted from the corresponding value of C2 to create each row of b. After solving for a, b, and d, we can use them to compute the two points that we need, and then the midpoint of those two points. The first point is calculated by multiplying a by the first unit vector and adding that to C1. Likewise, the second point is calculated by multiplying b by the second unit vector and adding that value to C2. The final point that we need is calculated by adding the previous two points together and dividing it by 2. This process is done for each of the twelve 2D points in each input array to compute the output array.

Measure Error Between Triangulated and Original 3D Points

The purpose of functions *measureError* and *computeEuclidean* in the program is to compute the Euclidean (L^2) distance between all joint pairs. This is a per joint, per frame L^2 distance between the original 3D joints and the reconstructed 3D joints which provides us with the analysis of the distance between the joint pairs. Since we are projecting 3D points into 2D (forward projection) and then back projecting those into viewing rays (backward projection) to do

triangulation back into 3D, the reconstructed point locations are very similar to the original locations. However, the computed locations are not exactly the same as the original locations due to small inaccuracies.

To measure how close our reconstructed 3D points come to the original set of 3D points, we can compute the Euclidean distance between them using *computeEuclidean.* This function takes one original point and the reconstructed point as input and then uses the equation sqrt((X-X')^2 + (Y-Y')^2 + (Z-Z')^2) to calculate the distance between the points. Then the function *measureError* uses *computeEuclidean* to measure the distance and error between all the original and reconstructed points and then finds the average error (avgError) using the sum of the euclidean distances.

Computing Epipolar Lines Between the Two Views

The purpose of findEpipolarLines is to use the 2D coordinates we found in 2.3 to calculate and overlay epipolar lines on the frames of vue2 and vue4. As learned in lecture, in order to find the epipolar lines the 8-point algorithm must be implemented. The second approach was used when creating the method for this function - using a viewing array to get the 2D points. To get the variables that go into the A matrix of the 8-point Algorithm, Hartley pre-condition must be done on the x and y coordinates of the inputted camera pixel coordinates. After performing Hartley pre-conditioning, the A matrix must be created using the variables modified via Hartley Preconditioning. Given the fact that there are 12 points, the A matrix is expected to be of size 12x9:

```
A Matrix
    0.1276    0.9281   -0.5823    0.3834    2.7892   -1.7498   -0.2191   -1.5940    1.0000
   -0.2585    0.6700   -0.7606   -0.3465    0.8981   -1.0195    0.3399   -0.8810    1.0000
   -0.7985    0.3771   -0.7236   -0.6196    0.2927   -0.5615    1.1034   -0.5212    1.0000
   -0.0919   -1.1448    0.6848    0.2244    2.7949   -1.6718   -0.1343   -1.6718    1.0000
    0.2920   -0.7857    0.7298   -0.3945    1.0614   -0.9859    0.4001   -1.0765    1.0000
    0.7209   -0.4495    0.6231   -0.6154    0.3838   -0.5320    1.1569   -0.7214    1.0000
    0.0703    0.0150   -0.2331    0.0441    0.0094   -0.1461   -0.3018   -0.0646    1.0000
    0.2135   -0.5596   -0.4601   -0.5130    1.3447    1.1055   -0.4641    1.2163    1.0000
    0.5132   -1.3979   -0.5775   -1.9829    5.4015    2.2315   -0.8886    2.4206    1.0000
   -0.0760   -0.0403    0.2994    0.0268    0.0142   -0.1055   -0.2539   -0.1346    1.0000
   -0.1060    0.3981    0.4222   -0.2732    1.0262    1.0884   -0.2510    0.9429    1.0000
   -0.2818    1.2050    0.5778   -1.1443    4.8937    2.3467   -0.4876    2.0853    1.0000
```

From there, the eigenvector associated with the smallest eigenvalue of A^T*A must be found and set to be matrix F. From there, F must be recomputed in order to be of rank 2. This is done by

finding the SVD of F, setting D(3,3) to 0, and recomputing F by multiplying U*D*V'.

```
F Matrix
   -0.0000    0.0000   -0.0013
    0.0000    0.0000    0.0039
   -0.0003   -0.0141    3.1902
```

Once F is found, L (the epipolar line) must be computed using the equation: F * [x1; y1; ones(size(x1))]; and ([x2; y2; ones(size(x2))]'*F)'; - where x1 and y1 are all the x values and y values of the stored camera pixel coordinates. In order to check the results of the computations, the lines must be overlaid onto the frames of both vues. This is done in the main function of the program using the output of findEpipolarLines and conditional statements. For results, see section 7.0.

Experimental Observations

When running the code for finding out the values for x, y, z locations for each joint, we expected the code to run just as we thought. The program ran itself smoothly and accurately gave us what we needed to start our project. There were no intermediate results, we immediately got our final result by running it once. The expected results are provided in the screenshots provided in section 2.0.

The function *project2DTo3D* is expected to produce 2D joint coordinates given specific camera parameters and 3D world coordinates. For each frame, the *project2DTo3D* function is called twice with the same x, y, and z-coordinates obtained from the mocap dataset: first with the vue2 camera parameters and then with the vue4 camera parameters. It can be expected that the 2D coordinates that this function produces may not be exactly correct to where the joints are in the frame image because we are ignoring the nonlinear distortion parameter in the radial field of the camera structures, but the points should be very close. To verify that the points we calculated are what we expect, we show two figures. The figures that are produced when *project3DTo2D* is run are shown in Figure 2. One thing to note is that the *project3DTo2D* function is called from the main function. Therefore, a figure will not be shown for every frame because not every frame has a confidence value for all ones. We ignore the frames where the confidence values for each joint do not add up to be 12.

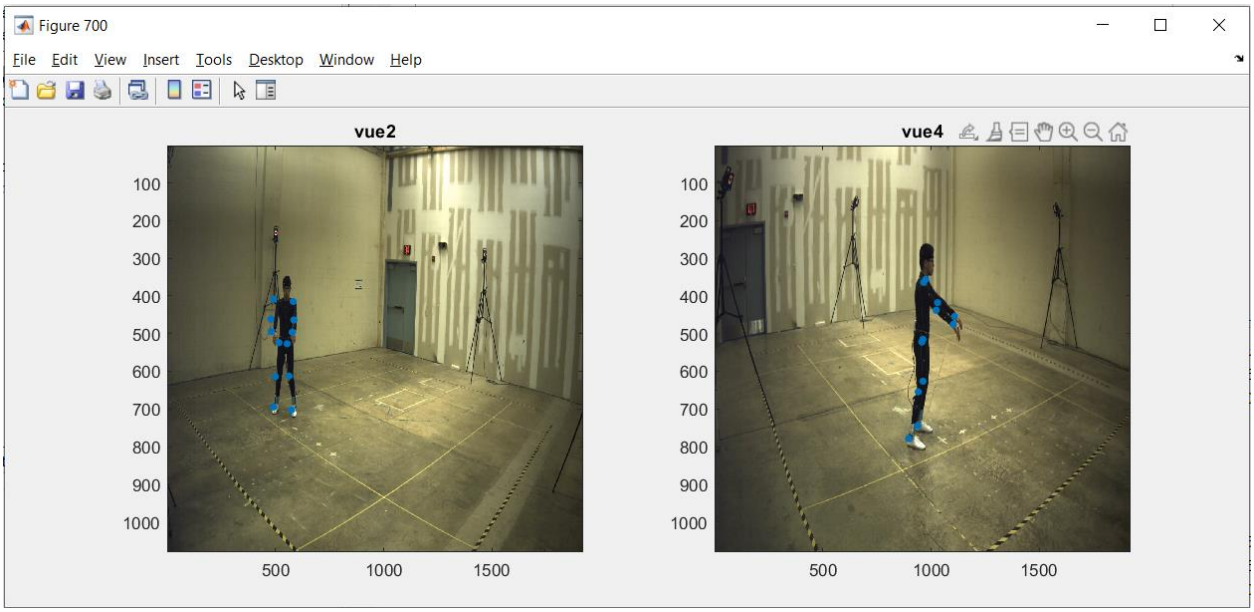


Figure 2: Figure that is shown when project3DTo2D is run using vue2 and vue4 camera parameters.

After viewing an example of the 2D coordinates that were produced when *project3DTo2D* is run, we can conclude that this function is performing as expected. Additionally, you can see in the figure that the points are slightly off from the joints, but not too far from where they should be.

When we run the function *reconstruct3DFrom2D*, the output is what it should be. The goal of this function is to take the 2D points from the *project3DTo2D* function and convert them back into 3D coordinates. Therefore, we should expect that the outputs of this function would be equal to the initial world coordinates. The next function tests that the reconstructed points are close enough to the original points by computing the error between them. Before we wrote the function to compute the error, we just manually checked that the points we were producing were almost exactly the same as the world coordinates for the same frame.

When we run *measureError* and *computeEuclidean* the output is as expected. The goal of the function is to quantitatively measure how close the reconstructed 3D points come to the original set of 3D points. The *computerEuclidean* function is successful in computing the L^2 between two points. In *measureError* this function is used correctly to calculate the *euclideanDistances* of all the original and reconstructed points. Lastly, just as suggested in the Project Description Document, we evaluate the average L^2 from the sum of *euclideanDistances* in order to summarize the reconstruction accuracy.

Running findEpipolarLines, the lines are displayed as expected in the project description. The goal for this function is to display epipolar lines that go through each of the 12 joints in both of the camera views. In order to do this, epipolar geometry must be utilized. There was a little difficulty getting the A matrix to work initially because there are 12 points as opposed to the 8 points used in the 8-Point Algorithm lecture sample code. Currently, as the code for the project

stands, A is able to compute a 12x9 matrix. At first, plotting was done in the findEpipolarLines function and the lines did not match the joints because both aspects (joints, lines) were referring to different frames because the joints were being plotted in main. With that said, it was important that the lines were plotted in the main function and not hard coded into the findEpipolarLines function in order to ensure proper line placement. Once those issues were resolved, the code seemed to run as expected - following the same workflow as the 8-point algorithm presented in class.

Quantitative Results

After computing the Euclidean distances for all the frames for a given joint we calculated the mean, standard deviation, minimum, median, and maximum of the L^2 distance. These are shown for each joint in a 13x5 matrix below. Finally we calculated these five metrics again for all joints and the results are displayed in the last row of the matrix.

	Mean	Standard deviation	Minimum	Median	Maximum
Right shoulder	1.0e-11 * 0.1141	1.0e-11 * 0.0538	1.0e-11 * 0	1.0e-11 * 0.1073	1.0e-11 * 0.4350
Right elbow	1.0e-11 * 0.1141	1.0e-11 * 0.0540	1.0e-11 * 0	1.0e-11 * 0.1066	1.0e-11 * 0.4374
Right wrist	1.0e-11 * 0.1148	1.0e-11 * 0.0547	1.0e-11 * 0	1.0e-11 * 0.1048	1.0e-11 * 0.4588
Left shoulder	1.0e-11 * 0.1137	1.0e-11 * 0.0558	1.0e-11 * 0	1.0e-11 * 0.1073	1.0e-11 * 0.4553
Left elbow	1.0e-11 * 0.1164	1.0e-11 * 0.0564	1.0e-11 * 0	1.0e-11 * 0.1073	1.0e-11 * 0.4802
Left wrist	1.0e-11 * 0.1169	1.0e-11 * 0.0565	1.0e-11 * 0	1.0e-11 * 0.1079	1.0e-11 * 0.4786
Right hip	1.0e-11 * 0.1160	1.0e-11 * 0.0539	1.0e-11 * 0	1.0e-11 * 0.1079	1.0e-11 * 0.4428
Right knee	1.0e-11 * 0.1163	1.0e-11 * 0.0535	1.0e-11 * 0	1.0e-11 * 0.1074	1.0e-11 * 0.4633
Right ankle	1.0e-11 * 0.1186	1.0e-11 * 0.0538	1.0e-11 * 0	1.0e-11 * 0.1100	1.0e-11 * 0.4616

Left hip	1.0e-11 * 0.1177	1.0e-11 * 0.0554	1.0e-11 * 0	1.0e-11 * 0.1092	1.0e-11 * 0.5486
Left knee	1.0e-11 * 0.1184	1.0e-11 * 0.0555	1.0e-11 * 0	1.0e-11 * 0.1092	1.0e-11 * 0.4576
Left ankle	1.0e-11 * 0.1207	1.0e-11 * 0.0559	1.0e-11 * 0.0007	1.0e-11 * 0.1122	1.0e-11 * 0.5545
All joints	1.0e-11 * 0.1167	1.0e-11 * 0.0550	1.0e-11 * 0	1.0e-11 * 0.1079	1.0e-11* 0.5545

By looping through each frame value, we summed the errors for the frame for all joints. We then plotted these points and their respective frames to see how error changes as time goes on. From the plot of the total error we are able to see two noticeable peaks.

Qualitative Results

To visualize the results at each step in the program, we displayed tables and figures where applicable. Because there are 21614 different frames to iterate through, we needed to choose a couple of frames to test. As suggested in the project description, we found the frame that had the maximum error between the original 3D world coordinates and our reconstructed 3D world coordinates. To do this, we iterated through all of the frames and captured the frames with the maximum and minimum error. As a result, frame 22347 had the minimum error of 8.3764e-13 between the 3D coordinates. Frame 16655 had the maximum error of 2.5282e-12 between the 3D coordinates. This section uses frames 22347 and 16655 to demonstrate the steps of our forward and inverse camera projection. The first step of the program is to use the mocap joint data for a given frame and read the 3D coordinates. Table 2a and 2b show the 3D coordinates that we use for future steps in the project.

Table 2a-b: Tables showing the 3D point data for each joint of frame 22347 and 16655.

3×12 table

	Right Shoulder	Right Elbow	Right Wrist	Left Shoulder	Left Elbow	Left Wrist	Right Hip	Right Knee	Right Ankle	Left Hip	Left Knee	Left Ankle
x-coord	1138.7	1190.5	1334.1	863.12	607.25	596.83	752.3	950.4	678.14	680.97	953.18	1111.3
y-coord	1238.2	1302.3	1467.1	1469.8	1353	1400.5	1090.7	1284.2	1279.3	1244	1433	1571.7
z-coord	969.64	706.51	528.34	1088.9	1002.8	722.19	655.07	362.87	71.943	656.8	444.66	47.324

	Right Shoulder	Right Elbow	Right Wrist	Left Shoulder	Left Elbow	Left Wrist	Right Hip	Right Knee	Right Ankle	Left Hip	Left Knee	Left Ankle
x-coord	-872.13	-936.11	-941.59	-599.56	-678.78	-850.18	-756.5	-1132.7	-1007.4	-674.82	-719.32	-718.06
y-coord	-1602.2	-1716.7	-1870.8	-1880.4	-1923.5	-1926.3	-1585	-1714.8	-1685.1	-1725.1	-1703	-1589.7
z-coord	1391.6	1115.4	1333.6	1417.3	1127.1	1336.4	925.17	922.13	556.21	877.28	480.21	39.539

After we collect the 3D joint data for a particular frame, we use the data to project 2D coordinates. The function project3DTo2D takes in the 3D coordinates and projects 2D points for the vue2 and vue4 cameras. To test that we correctly projected the points, we plotted skeletons using the 2D points we calculated. Shown in Figure 3a and 3b are plots showing the frame images for vue2 and vue4 with plotted 2D points along with line segments to create a skeleton of the body. Figure 3a shows the skeletons for frame 22347, which is the frame with the minimum error. Furthermore, Figure 3b shows the skeletons for frame 16655, which is the frame with the maximum error. Also given is Table 3a and 3b, which show the 2D points that we calculated for cameras vue2 and vue4.

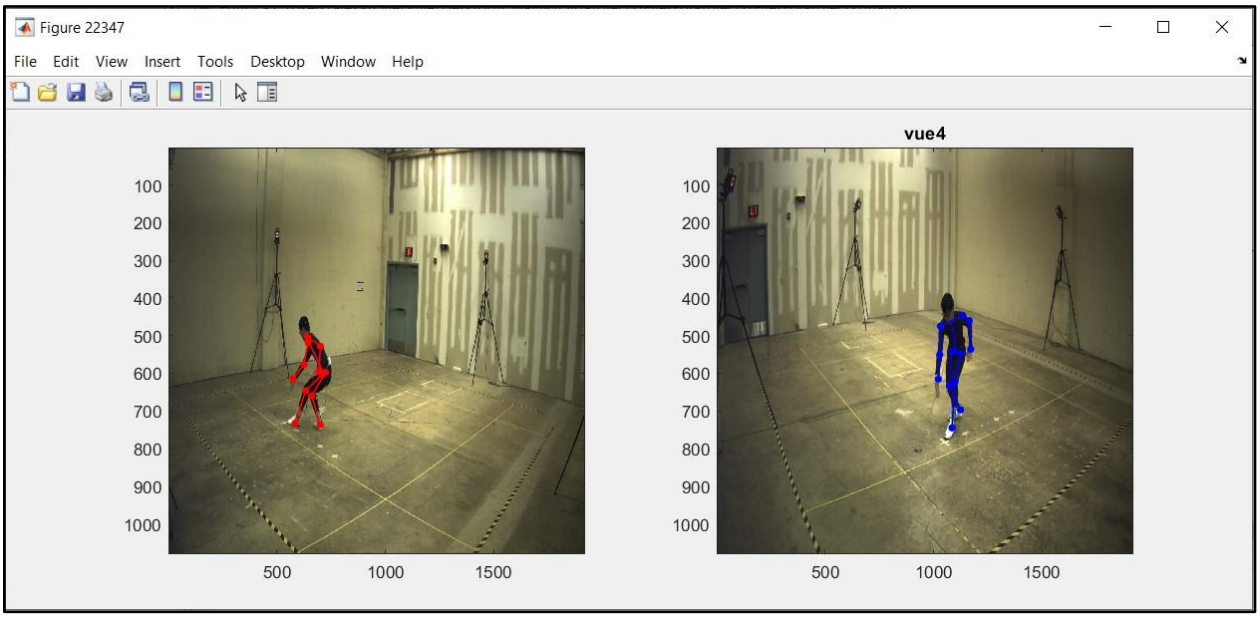


Figure 3a: Figure of 2D frame image with corresponding points and skeleton plotted for frame 22347.

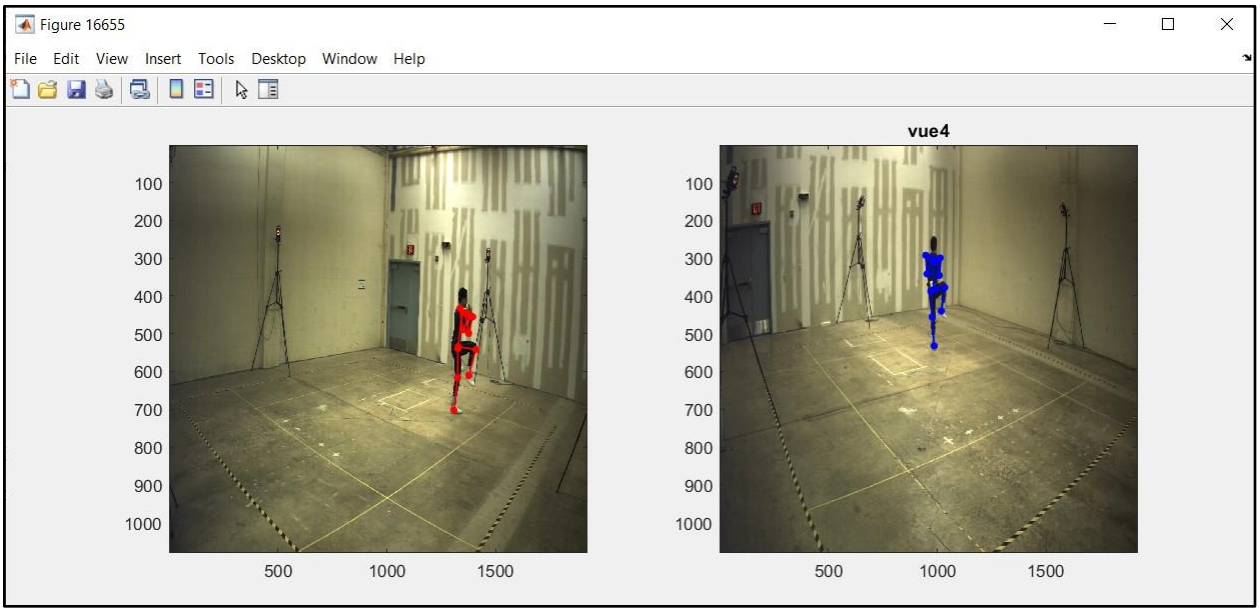


Figure 3b: Figure of 2D frame image with corresponding points and skeleton plotted for frame 16655.

Table 3a-b: Tables showing our calculated 2D points for each joint for frame 22347 and frame 16655.

points2DVue2 =

3×12 table

	Right Shoulder	Right Elbow	Right Wrist	Left Shoulder	Left Elbow	Left Wrist	Right Hip	Right Knee	Right Ankle	Left Hip	Left Knee	Left Ankle
x-coord	644.21	624.7	574.46	640.87	701.1	692.59	724.81	660.47	699.77	708.3	632.97	583.89
y-coord	518.99	576.62	614.67	502.71	528.27	595.53	597.68	660.26	736.39	603.61	646.52	732.83
z-coord	1	1	1	1	1	1	1	1	1	1	1	1

points2DVue4 =

3×12 table

	Right Shoulder	Right Elbow	Right Wrist	Left Shoulder	Left Elbow	Left Wrist	Right Hip	Right Knee	Right Ankle	Left Hip	Left Knee	Left Ankle
x-coord	1031.5	1028	1021.7	1134.1	1164.2	1170.6	1084.9	1073	1123.8	1125.1	1099.2	1084.6
y-coord	474.87	551.49	615.42	446.16	460.3	536.24	542.06	634.06	696.86	547.47	622.25	744.99
z-coord	1	1	1	1	1	1	1	1	1	1	1	1

points2DVue2 =

3×12 table

	Right Shoulder	Right Elbow	Right Wrist	Left Shoulder	Left Elbow	Left Wrist	Right Hip	Right Knee	Right Ankle	Left Hip	Left Knee	Left Ankle
x-coord	1357.9	1375.3	1393.8	1335.3	1348.3	1382.2	1327.6	1408.1	1375.8	1325.5	1324.9	1307.2
y-coord	444.3	499.31	455.14	432.96	489.26	452.51	534.11	542.26	611.01	539.19	616.23	702.12
z-coord	1	1	1	1	1	1	1	1	1	1	1	1

points2DVue4 =

3×12 table

	Right Shoulder	Right Elbow	Right Wrist	Left Shoulder	Left Elbow	Left Wrist	Right Hip	Right Knee	Right Ankle	Left Hip	Left Knee	Left Ankle
x-coord	1013	1008.4	995.03	946.78	952.52	977.05	995.83	1033.9	1017.9	970.07	976.44	985.11
y-coord	299.68	345.85	306.55	292.47	341.86	305.59	381.81	377.52	440.15	388	455.52	532.37
z-coord	1	1	1	1	1	1	1	1	1	1	1	1

Using the 2D points from the previous step, we call the function reconstruct3DFrom2D to transform the 2D points for each camera into one set of 3D points. Tables 4a and 4b show the 3D coordinates that we constructed for each frame.

Table 4a-b: Tables showing the reconstructed 3D point data for frames 22347 and 16655.

recovered3Dpoints =

3×12 table

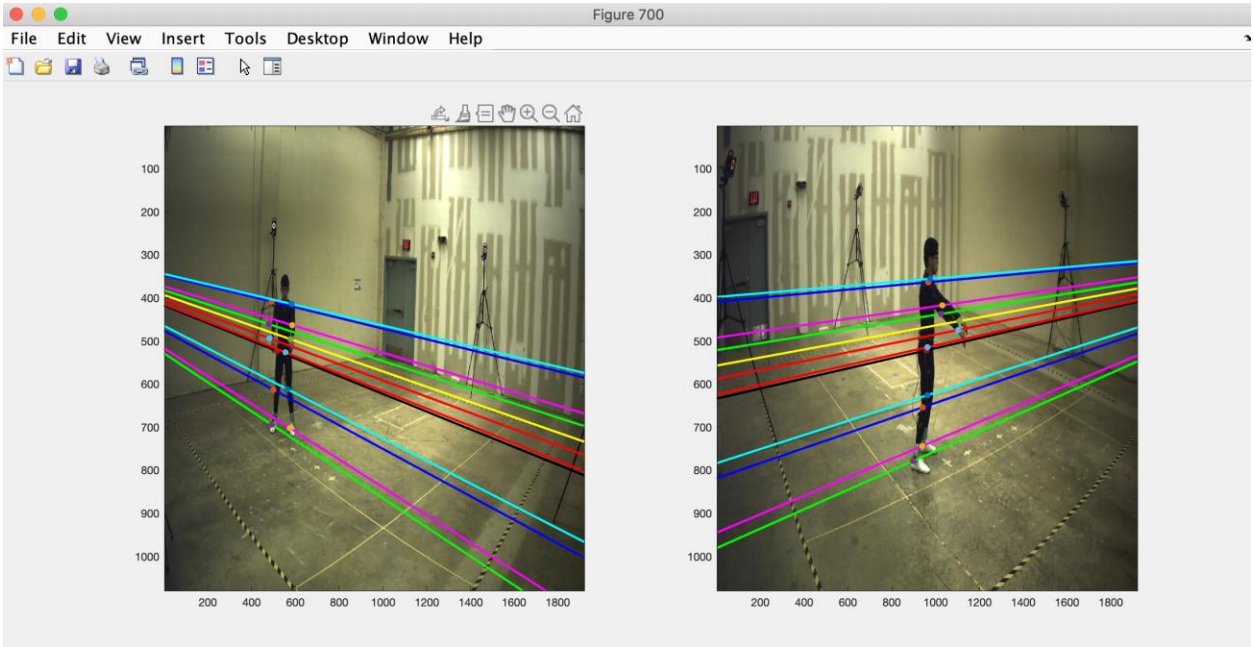
	Right Shoulder	Right Elbow	Right Wrist	Left Shoulder	Left Elbow	Left Wrist	Right Hip	Right Knee	Right Ankle	Left Hip	Left Knee	Left Ankle
x-coord	1138.7	1190.5	1334.1	863.12	607.25	596.83	752.3	950.4	678.14	680.97	953.18	1111.3
y-coord	1238.2	1302.2	1467.1	1469.8	1353	1400.5	1090.7	1284.2	1279.3	1244	1433	1571.7
z-coord	969.64	706.51	528.34	1088.9	1002.8	722.19	655.07	362.87	71.943	656.81	444.66	47.324

recovered3Dpoints =

3×12 table

	Right Shoulder	Right Elbow	Right Wrist	Left Shoulder	Left Elbow	Left Wrist	Right Hip	Right Knee	Right Ankle	Left Hip	Left Knee	Left Ankle
x-coord	-872.14	-936.11	-941.59	-599.56	-678.78	-850.18	-756.5	-1132.7	-1007.4	-674.82	-719.32	-718.06
y-coord	-1602.2	-1716.7	-1870.8	-1880.4	-1923.5	-1926.4	-1585	-1714.8	-1685.1	-1725.1	-1703	-1589.7
z-coord	1391.6	1115.4	1333.6	1417.3	1127.1	1336.4	925.17	922.13	556.21	877.28	480.21	39.539

Two-view Epipolar Lines Visualization



The epipolar lines for this project were computed using the 2D pixel coordinates retrieved from the function projecting 3D to 2D. Having the x and y's of those coordinates, Hartley preconditioning has to be done to to calculate the elements in the A matrix. Hartley preconditioning requires taking the mean of the x and y pixel coordinates separately, finding the standard deviation of the x and y pixels divided by 2, creating a T array, and then performing (x1-mux)/stdxy for both

x and y. This has to be done for both camera angles. Once Hartley preconditioning is done, the A array must be formulated using the following equations for each element:

$$\begin{bmatrix} x_1x'_1 & x_1y'_1 & x_1 & y_1x'_1 & y_1y'_1 & y_1 & x'_1 & y'_1 & 1 \\ \vdots & & & & & & & & \\ x_8x'_8 & x_8y'_8 & x_8 & y_8x'_8 & y_8y'_8 & y_8 & x'_8 & y'_8 & 1 \end{bmatrix}$$

The number of rows in A correlates to the number of points specified in the image. In the case of this project, A was a 12x9 matrix. After having the A matrix, eigenvectors and eigenvalues must be found using the equation A^T*A. The fundamental matrix, F, is then created using the values of the eigenvector associated with the smallest eigenvalue. F must be made singular by performing svd (singular value decomposition - UfDfVf^T) on F, making D'f the smallest singular value of F by setting it to 0, and recomputing F by performing the calculation for UD'V^T. Finally, F must be denormalized by performing T2' * F * T1 (T2 and T1 were found while performing Hartley Preconditioning). To get the components of the lines that go through each point, the equation is F * [x1; y1; ones(size(x1))] for camera 1 and ([x2; y2; ones(size(x2))]'*F)' for camera 2. The results of each computation are printed below.

```
Epipolar Line
  Columns 1 through 11

    0.0003    0.0005    0.0007    0.0003    0.0005    0.0007    0.0008    0.0012    0.0016    0.0008    0.0012
    0.0066    0.0066    0.0067    0.0071    0.0071    0.0071    0.0069    0.0069    0.0069    0.0070    0.0072
   -2.7192   -3.4561   -3.9211   -2.8235   -3.5197   -3.9778   -4.3519   -5.6163   -6.7555   -4.4037   -5.6165

  Column 12

    0.0016
    0.0073
   -6.8952

Epipolar Line
  Columns 1 through 11

    0.0012    0.0015    0.0017    0.0011    0.0014    0.0016    0.0020    0.0026    0.0032    0.0019    0.0025
   -0.0096   -0.0093   -0.0089   -0.0096   -0.0093   -0.0089   -0.0095   -0.0095   -0.0096   -0.0095   -0.0094
    3.3485    3.5617    3.6037    3.3054    3.4742    3.5153    3.9800    4.5215    5.0675    3.9450    4.3816

  Column 12

    0.0030
   -0.0094
    4.8771
```

Contributions

This table provides the tasks that each person in our group did and by what date it was done.

Name	Task	Date

Kritika Senthil	1. Kritika started by writing up the summary and context section. 2. Did sections 2.1 and 2.2 as given in the project description document 3. Put in all the screenshots for section 5.0	1. 10/9 2. 10/13 3. 11/8
Alexandria Eicher	1. Began experimenting with sample code in the project document in preparation to write the function described in 2.3 2. Set up Github repository for the project and uploaded initial code to it 3. Wrote initial code for projecting 3D points into 2D pixels 4. Finished the code that plots 2D joint coordinates onto a frame. The points are not correct yet. 5. Finished function 2.3 by retrieving correct 2D joint coordinates and plotting them on the frame. 6. Reconstructed function 2.3 based on the new supplemental requirements 7. Created flowchart and wrote opening statement for Outline of Procedural Approach 8. Finished writing Matlab code for function 2.4 and filled in those parts of the document. 9. Rearranged images in section 5.0 and wrote that part of the report.	1.10/8 2. 10/9 3. 10/13 4.10/21 5.10/23 6. 10/27 7.10/29 8. 11/5 9. 11/8

Isabella Salvi	1. Recognized use for 8 Point Algorithm - began experimenting in MatLab with lecture sample code 2. Worked on modifying 8 Point Algorithm to work for Section 2.6 (epipolar line function) 3. Plotted epipolar lines onto video frames in the main function 4. Wrote about 2.6 in the Outline of Procedural Approach 5. Wrote and retrieved screenshot for the Two View Epipolar Lines section of the written report 6. Wrote Experimental Observations for finding epipolar lines	
Osheen Arya	1. Began looking into euclidean distance for function 2.5 and the Quantitative evaluation section. 2. Worked on using the function provided for L^2 distance to compute distance between two points. 3. Finished code to calculate the distance between triangulated and original 3D points (function 2.5), began work on the Evaluation section. 4. Completed quantitative evaluation code (5 metrics and total error) 5. Wrote in section 2 (2.5), wrote in section 3 (2.5) 6. Completed section 4.0. Created 13x5 table of metrics and total	1. 10/13 2. 10/19 3. 10/21 4. 11/2 5. 11/8

	error plot. Wrote about results and added screenshots of code.	6. 11/8