CMPEN 454 Project 2

Team Members: Brian Nguyen, Kyle Bradley, Jordan Reed, Drice Bahajak

Summary

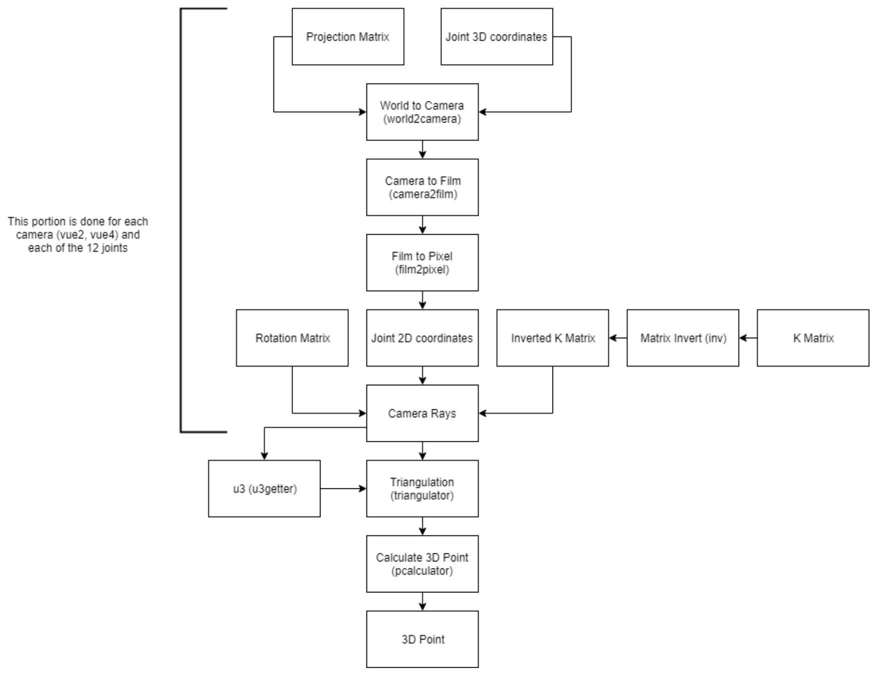
We believe that the goal of this project is to work towards having a clear understanding of camera projection. This will be done by performing numerous tasks within camera projection such as forward and inverse camera projection, triangulation, and all of the necessary matrix transformations for these tasks. To achieve this, we used 3D point data coming from a video to translate into 2D points, and then back to confirm that the projections are accurate. Our approach consisted of creating various functions in Matlab for the various steps of camera projection such as translating world coordinates to camera coordinates, camera coordinates to film coordinates, film coordinates to pixel coordinates and triangulation. From performing these tasks, we wished to accomplish accurate translation from 3D to 2D points as well as similar calculated 3D points to the originals.

Approach

Given the structure with fields: focalen, orientation, Pmat, Rmat, and Kmat, each one of those fields is related to the pinhole camera model parameters. The internal parameters are focal length, offset, and prinpoint. The external parameters are Rmat and the position vector. The internal parameters that combine to form Kmat are the focal lengths with the offset. Kmat is also known as the perspective projection matrix. The external parameters that combine to form Pmat are Rmat with another row multiplied with position vector with another column. The location of the camera can be defined as Pc=R*Pw. Therefore, if you perform the multiplication, you will find Pc = [137.7154, 805.5227, 7.3365e+03]' which is the fourth column of Pmat. If we wanted to check, we could find Pw by R'*Pc=Pw to find that Pw = [-4.4501e+03, 5.5579e+03, 1.9491e+03]'. This is what we expect to get because this is the exact position vector we started with and combined with Rmat in order to find Pmat.

Below is a flowchart and explanation of our implementation of the forward and inverse camera projection of a human model and triangulation. The functions in parentheses refers to the functions we have created/used for the given step.

For this project, we were given the 3D point data per frame of 12 body joints coming from a video, the necessary parameters of two video cameras, such as focal length, positions, rotation matrices, pixel matrices and projection matrices, and an mp4 movie file to display our results.

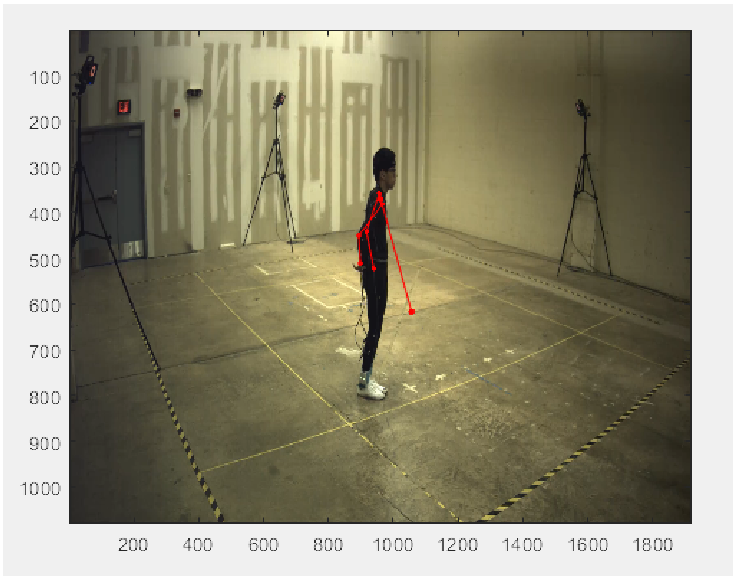

This portion is done for each camera (vue2, vue4) and each of the 12 joints
Projection Matrix
Joint 3D coordinates
World to Camera (world2camera)
Camera to Film (camera2film)
Film to Pixel (film2pixel)
Rotation Matrix
Joint 2D coordinates
Inverted K Matrix
Matrix Invert (inv)
K Matrix
Camera Rays
u3 (u3getter)
Triangulation (triangulator)
Calculate 3D Point (pcalculator)
3D Point

- 3D points to 2D points done for each camera (vue2 and vue4)
  - World Coordinates to Camera Coordinates (function world2camera)
    - By matrix multiplying our given projection matrix and the world coordinates for each joint, we were able to convert world coordinates to camera coordinates.
  - Camera Coordinates to Film Coordinates (function camera2film)
    - By using the perspective projection equations: x = f * X / Z, y = f * Y / Z, where f is the focal length of the camera and (X, Y, Z) are the camera coordinates, we were able to convert camera coordinates to film coordinates for each joint.
    - **NOTE:** Use of this function is not necessary because kmat was already provided to us.
  - FIlm Coordinates to Pixel Coordinates (function film2camera)
    - By matrix multiplying the provided K Matrix and the film coordinates for each joint, we were able to convert film coordinates to pixel coordinates.
- 2D points back to 3D points by using pixel coordinates from each camera
  - Calculating Camera Rays
    - Two camera rays were calculated, one for vue2 and one for vue4. By matrix multiplying our provided rotation matrix, inverted K matrix, and pixel coordinates, we were able to calculate the camera rays.
  - Calculating u3 (function u3getter)
    - The perpendicular line to the camera rays, u3, is calculated by taking the cross product of the two camera rays and then dividing it by the normalized version of the cross product of the camera rays.
  - Triangulation (function triangulator)
    - By taking the “u matrix”, where u1 and u2 are the two cross rays and u3 is the perpendicular line to them, inverting it, and then multiplying it with the difference of the position matrices for the two cameras, we were able to calculate a, b, d.
  - Calculating 3D points (function pcalculator)
    - After solving for a, b, d, we wish to calculate p1 and p2: p1 = (c1 + a * u1), p2 = c2 + (b * u2) where c1 and c2 are the two camera rays. From p1 and p2, we are able to calculate p, the 3D point, with the equation p = (p1 + p2) / 2
- Calculating metrics
  - Euclidean distance (function euclidean_distance)

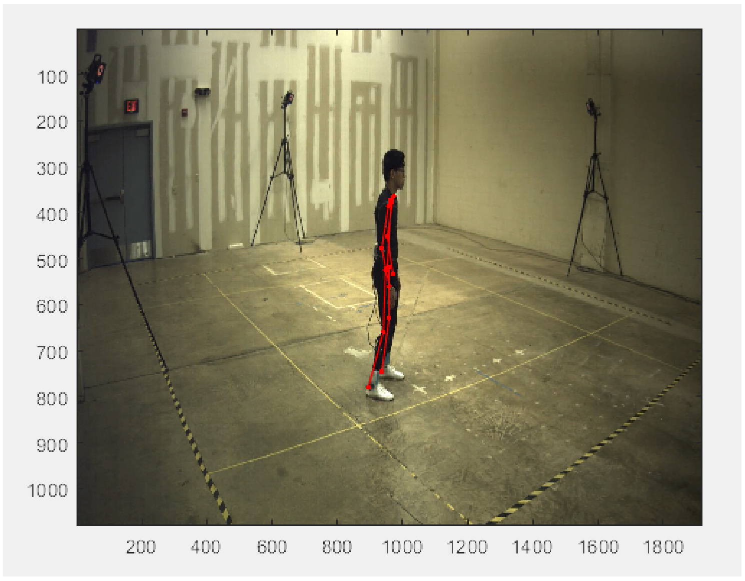
        - For each given frame and its associated points for each joint, we computed the euclidean distance between the point and our point we calculated after all of the above operations. The equation for the euclidean distance is as follows: x = sqrt(sum((point1(:) - point2(:)) .^ 2)).
    - Mean, SD, Min, Median, Max
        - After computing the euclidean distances for all of the frames for a given joint, we would compute the five required metrics for those calculations. These would be stored for each joint into a matrix to have it in a finalized 13x5 matrix. After all of the euclidean distances were calculated for all joints, we calculated the five metrics again across all joints to get the last row of the matrix.
    - Sums of error across the joints
        - By looping through each frame value, we summed the errors for the frame for all joints and stored these calculations into a matrix. We then plotted these points and their respected frames to see how error changes as time goes on.

Observations

Each joint has a binary “confidence” associated with it. Joints that are not defined in a frame have a confidence of zero. Before removing the frames with confidence zero, we noticed that the skeleton lines would not always match up with the joints correctly. Some frames worked well, and then others like frame 1 below did not work as well.

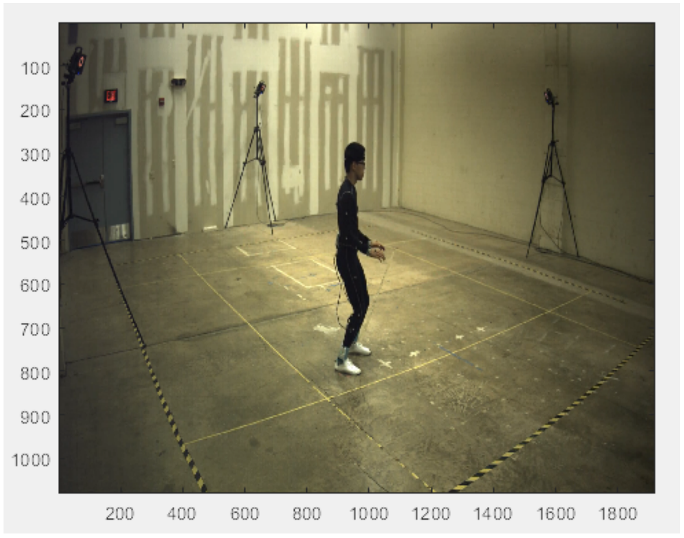


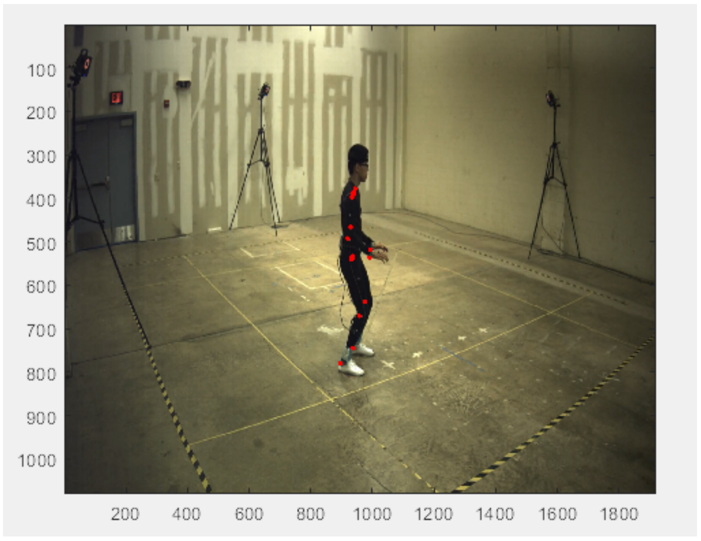
When confidence of 0 is taken into account for frame 1.

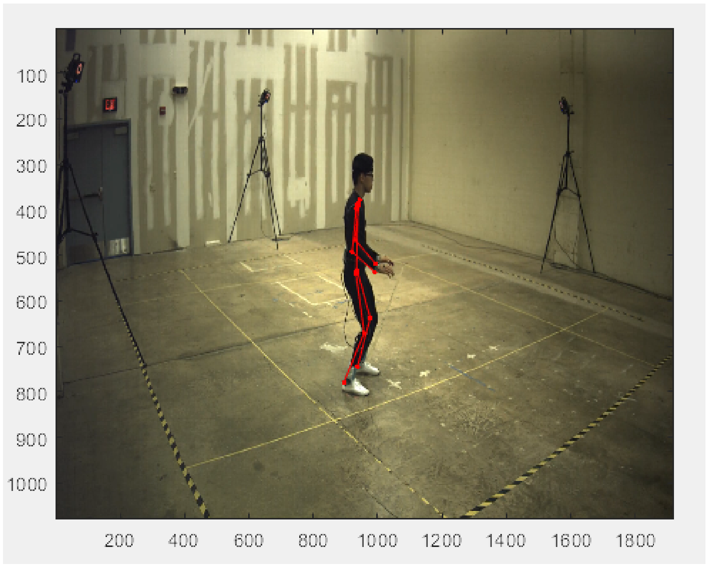


When confidence of 0 is removed for frame 1.

The first step of creating the skeleton was projecting the 3D video to a 2D picture. Once that was accomplished, adding the points to each of the 12 joints was the next step. In order to get the lines from each joint, we just plotted each line individually and then finding the average between the shoulders and hips to plot a point and then adding the spine from the midpoint of the hips to shoulders.





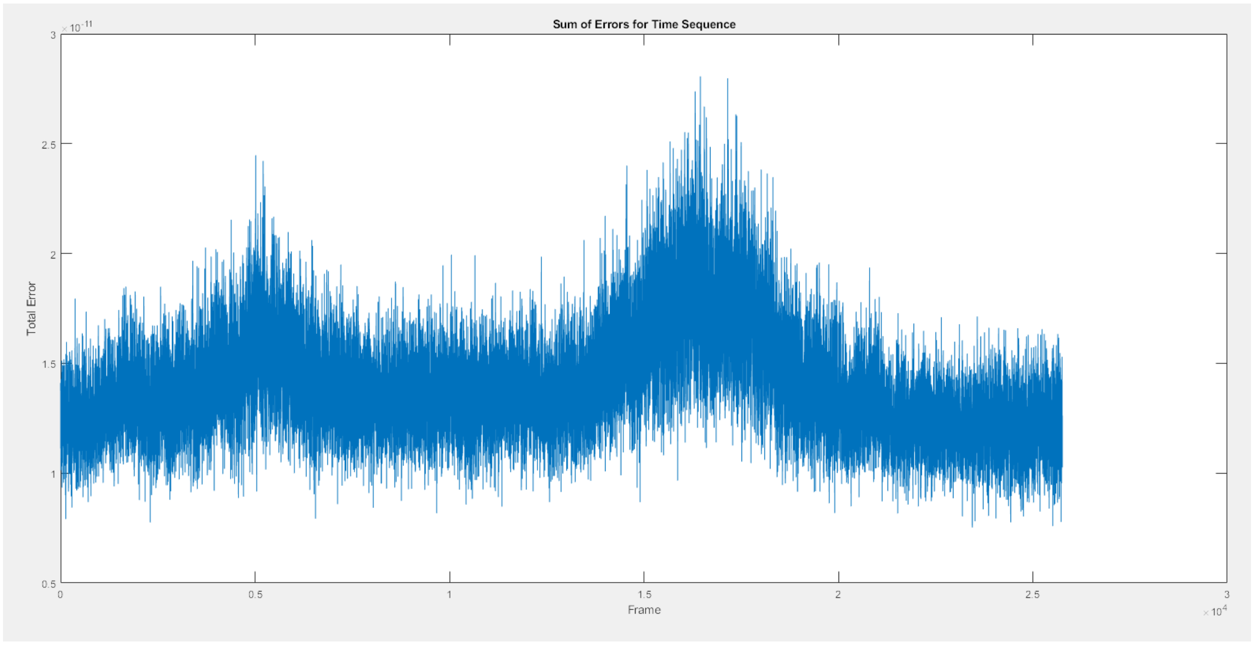


Overall, the program works just how we wanted and expected it to. The Euclidean distance (L^2) between joint pairs (per joint per frame) are extremely small, on the scale of 10^-12. That is very good because there will inevitably be some error. If the error was Zero, then that would not have been correct. That would be like find the error between two pictures that are exactly the same. Therefore, having an error that small is a good thing and don't know if it could get much better.

Results

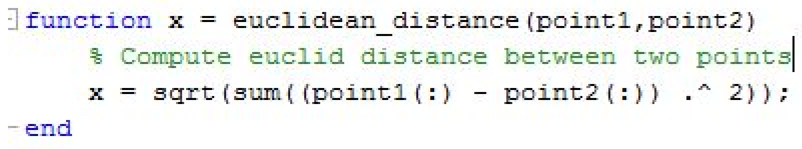
Quantitative

	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**

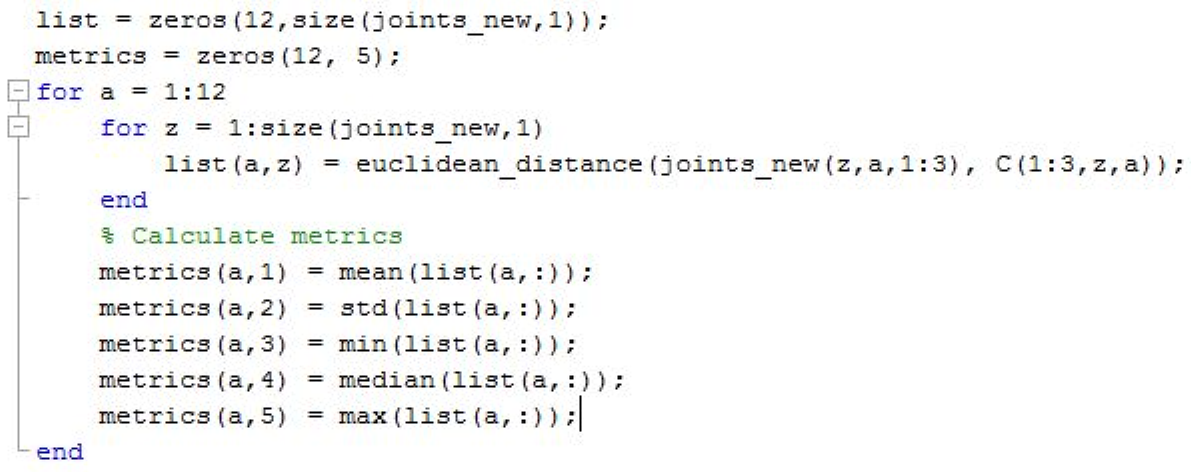


From the plot of of the total error for the whole time sequence, we can see that there are two noticeable peaks.

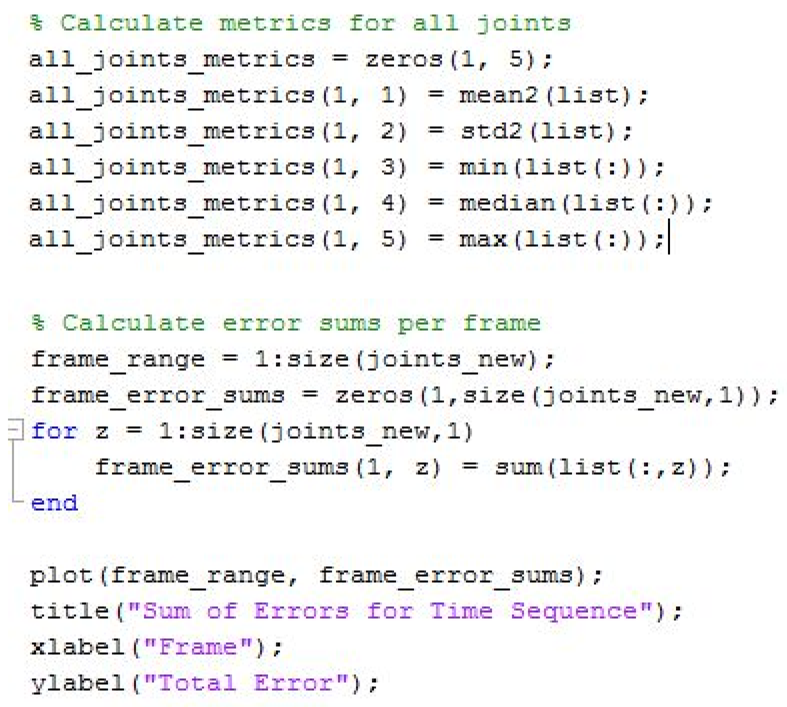
Below is the specific parts of our script that were used to calculate these metrics.



```
function x = euclidean_distance(point1,point2)
    % Compute euclid distance between two points
    x = sqrt(sum((point1(:) - point2(:)) .^ 2));
end
```



```
list = zeros(12,size(joints_new,1));
metrics = zeros(12, 5);
for a = 1:12
    for z = 1:size(joints_new,1)
        list(a,z) = euclidean_distance(joints_new(z,a,1:3), C(1:3,z,a));
    end
    % Calculate metrics
    metrics(a,1) = mean(list(a,:));
    metrics(a,2) = std(list(a,:));
    metrics(a,3) = min(list(a,:));
    metrics(a,4) = median(list(a,:));
    metrics(a,5) = max(list(a,:));
end
```



```
% Calculate metrics for all joints
all_joints_metrics = zeros(1, 5);
all_joints_metrics(1, 1) = mean2(list);
all_joints_metrics(1, 2) = std2(list);
all_joints_metrics(1, 3) = min(list(:));
all_joints_metrics(1, 4) = median(list(:));
all_joints_metrics(1, 5) = max(list(:));

% Calculate error sums per frame
frame_range = 1:size(joints_new);
frame_error_sums = zeros(1,size(joints_new,1));
for z = 1:size(joints_new,1)
    frame_error_sums(1, z) = sum(list(:,z));
end

plot(frame_range, frame_error_sums);
title("Sum of Errors for Time Sequence");
xlabel("Frame");
ylabel("Total Error");
```

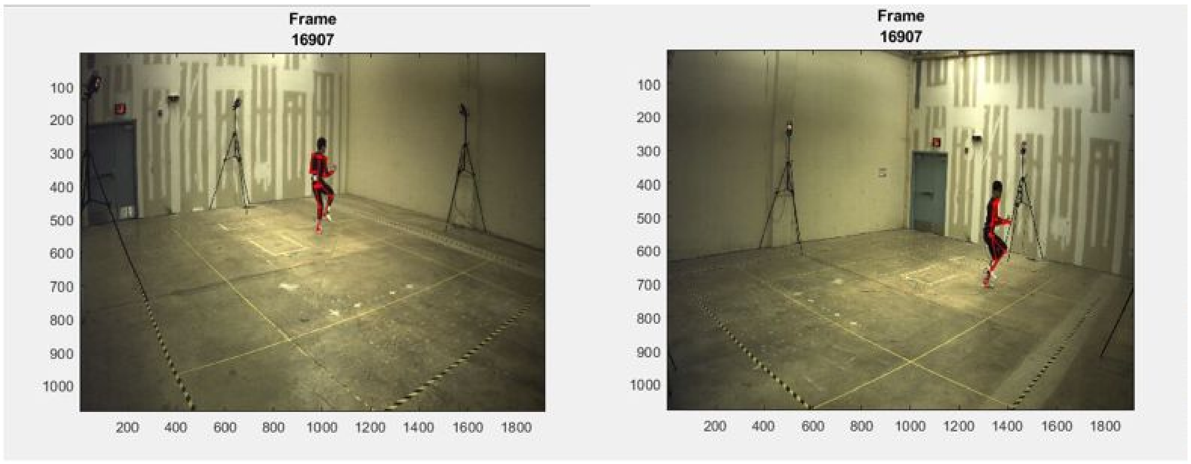
Qualitative

For our frames, we chose to use the frame of the minimum error for all joints, the maximum error for all joints, and then three random frames. The computed 3D values are the same to their origins up to 5 significant digits and over for each frame. Therefore, instead of displaying the computed 3D values for each of these examples, we share the computed error.

The indices for these frames are as follows:

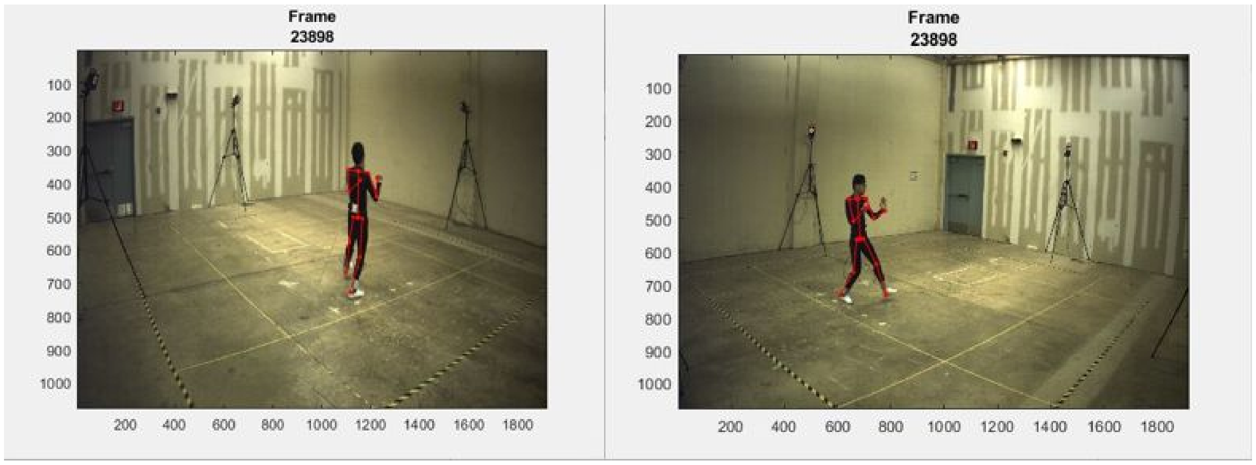
	Index	**Frame**
Maximum Error	16451	16907
Minimum Error	23442	23898
Random	765	1221
Random	2424	2880
Random	7431	7887

- Frame 16907:
  - World Point: (-701.69.1, -1921.3, 1022.0)
  - Rendered 2D Skeletons:



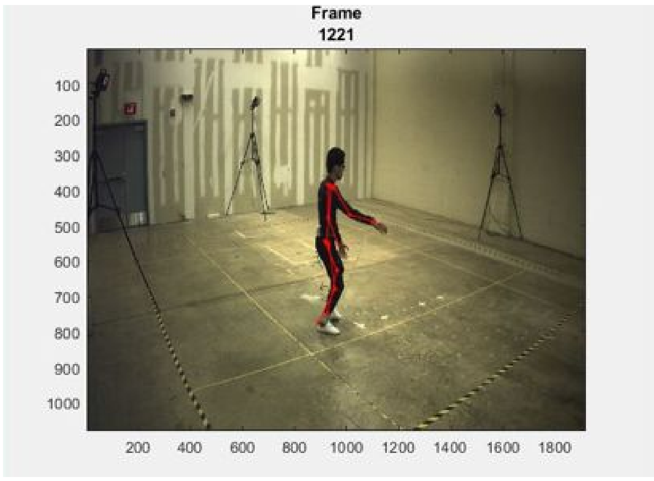
  - Computed Error: 2.1600 x10^-12

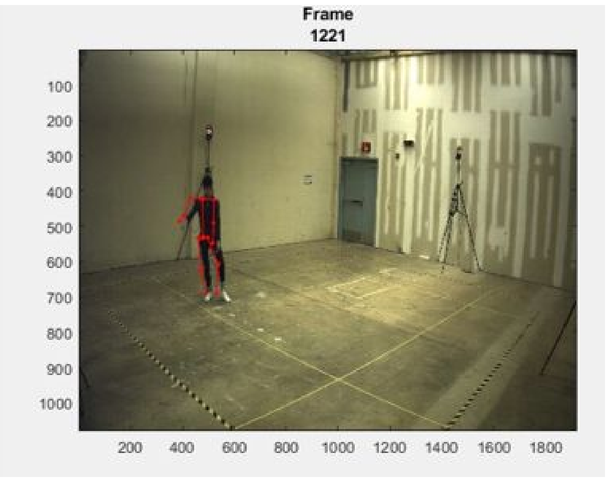
- Frame 23898:
  - World Point: (671.047, 1114.4, 1103.4)
  - Rendered 2D Skeletons:



  - Computed Error: 5.0842 x 10^-13

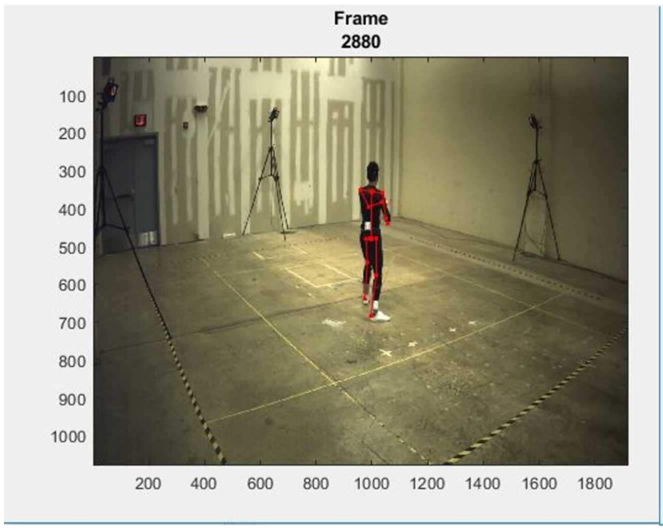
- Frame 1221:
  - World Point: (1743.4, 1484.8, 1042.1)
  - Rendered 2D Skeletons:

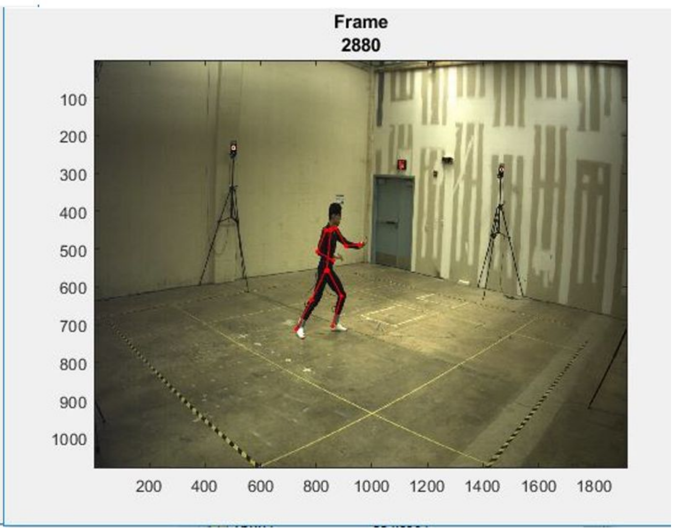




- Computed Error: 1.7465 x 10^-12

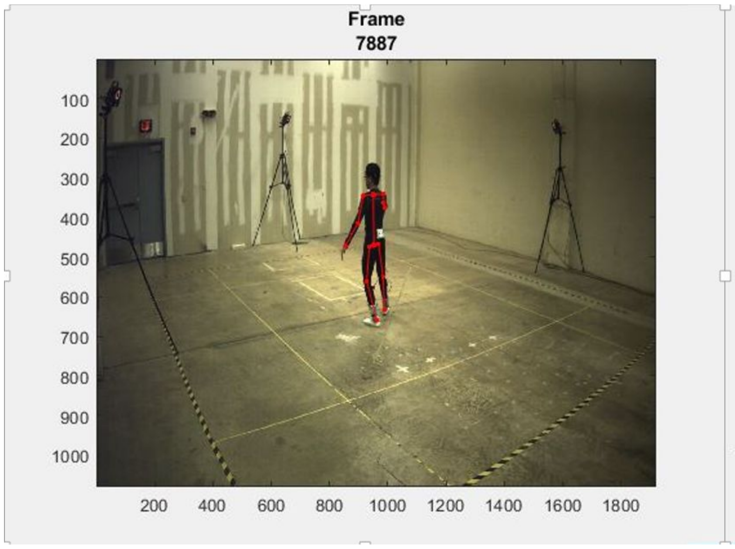
- Frame 2880:
  - World Point: (511.644, 150.421, 1094.3)
  - Rendered 2D Skeletons:

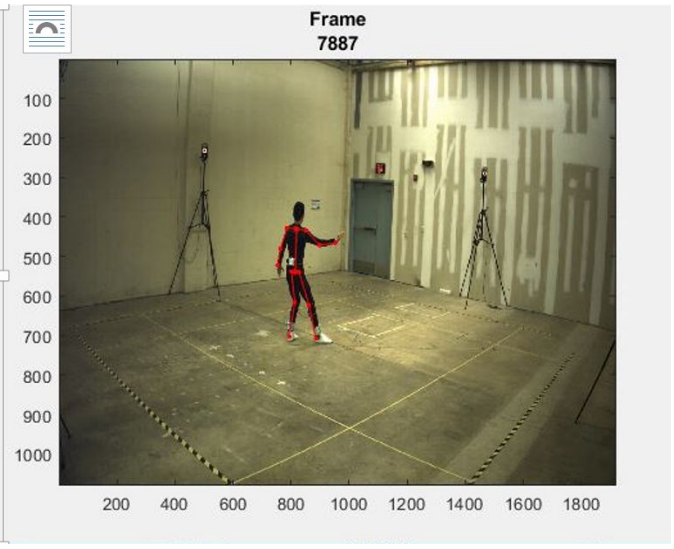




  - Computed Error: 6.6534 x 10^-13

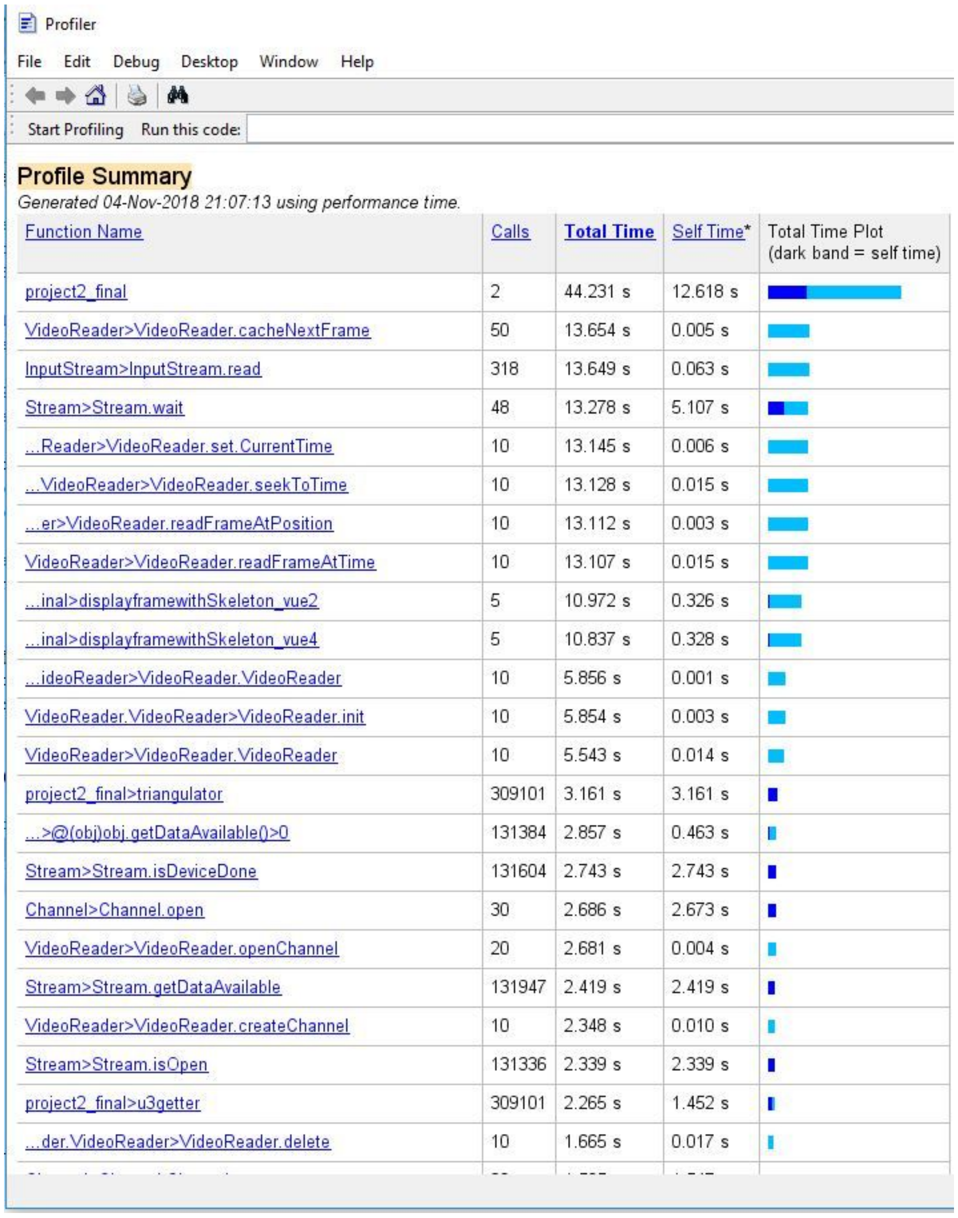
- Frame 7887:
    - World Point: (1216.3, 462.958, 1093.5)
    - Rendered 2D Skeletons:





    - Computed Error: 1.3074 x 10^-12

Efficiency


Profile Summary

Generated 04-Nov-2018 21:07:13 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
project2_final	2	44.231 s	12.618 s	
VideoReader>VideoReader.cacheNextFrame	50	13.654 s	0.005 s	
InputStream>InputStream.read	318	13.649 s	0.063 s	
Stream>Stream.wait	48	13.278 s	5.107 s	
...Reader>VideoReader.set.CurrentTime	10	13.145 s	0.006 s	
...VideoReader>VideoReader.seekToTime	10	13.128 s	0.015 s	
...er>VideoReader.readFrameAtPosition	10	13.112 s	0.003 s	
VideoReader>VideoReader.readFrameAtTime	10	13.107 s	0.015 s	
...inal>displayframewithSkeleton_vue2	5	10.972 s	0.326 s	
...inal>displayframewithSkeleton_vue4	5	10.837 s	0.328 s	
...ideoReader>VideoReader.VideoReader	10	5.856 s	0.001 s	
VideoReader.VideoReader>VideoReader.init	10	5.854 s	0.003 s	
VideoReader>VideoReader.VideoReader	10	5.543 s	0.014 s	
project2_final>triangulator	309101	3.161 s	3.161 s	
...>@(obj)obj.getDataAvailable()>0	131384	2.857 s	0.463 s	
Stream>Stream.isDeviceDone	131604	2.743 s	2.743 s	
Channel>Channel.open	30	2.686 s	2.673 s	
VideoReader>VideoReader.openChannel	20	2.681 s	0.004 s	
Stream>Stream.getDataAvailable	131947	2.419 s	2.419 s	
VideoReader>VideoReader.createChannel	10	2.348 s	0.010 s	
Stream>Stream.isOpen	131336	2.339 s	2.339 s	
project2_final>u3getter	309101	2.265 s	1.452 s	
...der.VideoReader>VideoReader.delete	10	1.665 s	0.017 s	

By running the Matlab profiler, we can see the time taken for different components above. Besides the triangulation, most of our functions run in very fast time, but that is simply due to it being run on each frame. It seems as though most of the time taken comes from the video being displayed. From these numbers, we are happy with the efficiency of our code, with a total time of about 12.6 seconds.

Contribution

	Tasks Done
Brian Nguyen	● Created code for computing all metrics data and visuals used for quantitative results section. Created flowchart. Wrote summary, approach, and quantitative results sections. Added documentation.
Kyle Bradley	● Worked on functionality of camera projection.
Jordan Reed	● Worked on functionality/mathematics of camera projection.
Drice Bahajak	● Created skeleton from 3d to 2d points. Wrote observations section. Answered the questions under the 2.2 section.