ELEC0140

Robotics for Electronic Engineering
Virtual Robot Report II

Group 17
Rachel Sava
Suizhong Qin
Kai Xu
Module Tutor: Dr Chow Yin Lai

University College London

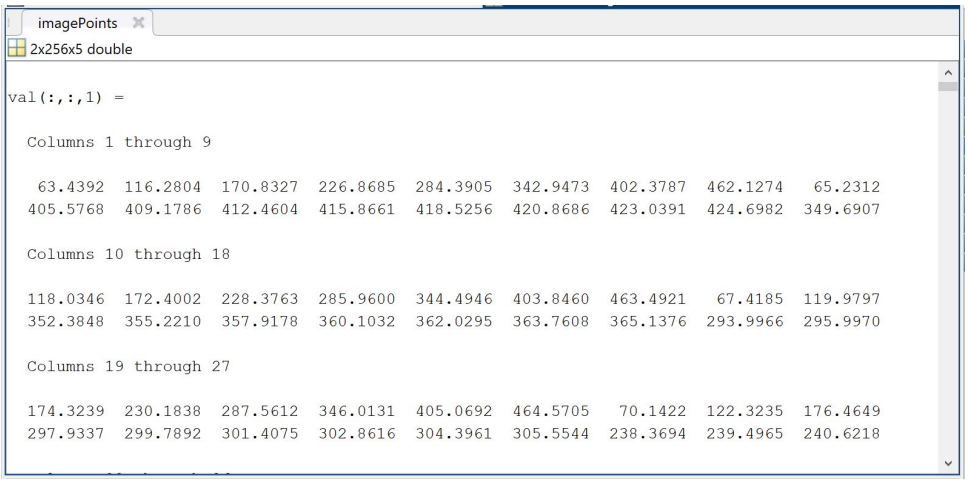
January 2021

Contents

1 Image Processing

1.1 Task 1: Camera Calibration Given Checkerboard Data

The first task of this Vision assignment focuses on calibrating a camera, given pre-processed checkerboard data downloaded from Zhengyou Zhang's Microsoft Research publication; select snapshots of the data are represented in figures 1 and 2. The data to be used during the calibration consists of 5 files containing the (x,y) pixel coordinates of corner points in 5 checkerboard orientations, and a file of the (X,Y) coordinates of the corner points on the physical checkerboard printout.

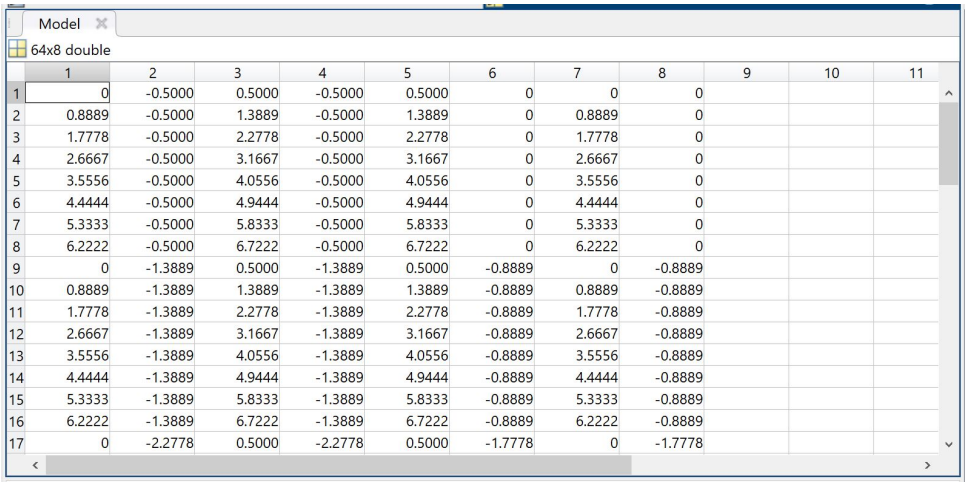


imagePoints

2x256x5 double

```
val(:,:,1) =

  Columns 1 through 9

   63.4392  116.2804  170.8327  226.8685  284.3905  342.9473  402.3787  462.1274   65.2312
  405.5768  409.1786  412.4604  415.8661  418.5256  420.8686  423.0391  424.6982  349.6907

  Columns 10 through 18

  118.0346  172.4002  228.3763  285.9600  344.4946  403.8460  463.4921   67.4185  119.9797
  352.3848  355.2210  357.9178  360.1032  362.0295  363.7608  365.1376  293.9966  295.9970

  Columns 19 through 27

  174.3239  230.1838  287.5612  346.0131  405.0692  464.5705   70.1422  122.3235  176.4649
  297.9337  299.7892  301.4075  302.8616  304.3961  305.5544  238.3694  239.4965  240.6218
```

Figure 1: Part of Layer 1 'imagePoint' matrix containing (x,y) values from each of 5 image datasets



Model

64x8 double

	1	2	3	4	5	6	7	8	9	10	11
1	0	-0.5000	0.5000	-0.5000	0.5000	0	0	0			
2	0.8889	-0.5000	1.3889	-0.5000	1.3889	0	0.8889	0			
3	1.7778	-0.5000	2.2778	-0.5000	2.2778	0	1.7778	0			
4	2.6667	-0.5000	3.1667	-0.5000	3.1667	0	2.6667	0			
5	3.5556	-0.5000	4.0556	-0.5000	4.0556	0	3.5556	0			
6	4.4444	-0.5000	4.9444	-0.5000	4.9444	0	4.4444	0			
7	5.3333	-0.5000	5.8333	-0.5000	5.8333	0	5.3333	0			
8	6.2222	-0.5000	6.7222	-0.5000	6.7222	0	6.2222	0			
9	0	-1.3889	0.5000	-1.3889	0.5000	-0.8889	0	-0.8889			
10	0.8889	-1.3889	1.3889	-1.3889	1.3889	-0.8889	0.8889	-0.8889			
11	1.7778	-1.3889	2.2778	-1.3889	2.2778	-0.8889	1.7778	-0.8889			
12	2.6667	-1.3889	3.1667	-1.3889	3.1667	-0.8889	2.6667	-0.8889			
13	3.5556	-1.3889	4.0556	-1.3889	4.0556	-0.8889	3.5556	-0.8889			
14	4.4444	-1.3889	4.9444	-1.3889	4.9444	-0.8889	4.4444	-0.8889			
15	5.3333	-1.3889	5.8333	-1.3889	5.8333	-0.8889	5.3333	-0.8889			
16	6.2222	-1.3889	6.7222	-1.3889	6.7222	-0.8889	6.2222	-0.8889			
17	0	-2.2778	0.5000	-2.2778	0.5000	-1.7778	0	-1.7778			

Figure 2: 64x8 double matrix containing (X,Y) coordinates of the checkerboard printout corner points.

Given this dataset, to extract the intrinsic and extrinsic parameters by the camera calibration and ultimately compared with Zhang's known results.

$$\begin{bmatrix} 0 & 0 & 0 & X_1 & Y_1 & 1 & -y_1X_1 & -y_1Y_1 & -y_1 \\ X_1 & Y_1 & 1 & 0 & 0 & 0 & -x_1X_1 & -x_1Y1 & -x_1 \\ 0 & 0 & 0 & X_2 & Y_2 & 1 & -y_2X_2 & -y_2Y_2 & -y_2 \\ X_2 & Y_2 & 1 & 0 & 0 & 0 & -x_2X_2 & -x_2Y2 & -x_2 \\ 0 & 0 & 0 & X_3 & Y_3 & 1 & -y_3X_3 & -y_3Y_3 & -y_3 \\ X_3 & Y_3 & 1 & 0 & 0 & 0 & -x_3X_3 & -x_3Y3 & -x_3 \\ 0 & 0 & 0 & X_4 & Y_4 & 1 & -y_4X_4 & -y_4Y_4 & -y_4 \\ X_4 & Y_4 & 1 & 0 & 0 & 0 & -x_4X_4 & -x_4Y4 & -x_4 \end{bmatrix} \times \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The intrinsic parameters of a camera are defined within the K matrix:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \sim \begin{bmatrix} \alpha & \gamma & x_0 \\ 0 & \beta & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = K \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix}$$

First, determining the H values for each image sequentially:

For each point i, in the linear-in-parameter form, the equation relating h values to the known parameters is:

$$\begin{bmatrix} 0 & 0 & 0 & X_i & Y_i & 1 & -y_iX_i & -y_iY_i & -y_i \\ X_i & Y_i & 1 & 0 & 0 & 0 & -x_iX_i & -x_iYi & -x_i \end{bmatrix} \times \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

To implement the camera calibration algorithm, there are 7 steps to follow. Since the step 1 to step 4 a) are provided in our code, we need to start our code from step 4 b), which is to build the matrix of phi:

```
phi = zeros(64*4*2,9,5); % create five 512x9 zeros matrices
% Step 4 b) Build the matrix of phi
for layer_num = 1:5
    for odd_row = 1:2:512
        i = (odd_row+1)/2;
        phi(odd_row,:,layer_num) = [0,0,0,X(i),Y(i),1,...
            -imagePoints(2,i,layer_num)*X(i),...
            -imagePoints(2,i,layer_num)*Y(i),...
            -imagePoints(2,i,layer_num)];

        phi(odd_row+1,:,layer_num)=[X(i),Y(i),1,0,0,0,...
            -imagePoints(1,i,layer_num)*X(i),...
            -imagePoints(1,i,layer_num)*Y(i),...
            -imagePoints(1,i,layer_num)];
```

```
    end
end
```

Firstly, create five $512 \times 9 \times 5$ zeros matrix. Follow the phi matrix pattern, use a for loop to print odd rows and event rows separately. Later, from the Step 4 c), we need to calculate the h vector using the Singular Value Decomposition(SVD) method.

```
% Step 4 c) Calculate the h vector using SVD
h = zeros(9,5);
H = zeros(3,3,5);
for layer_num = 1:5
    [U,S,V] = svd(phi(:,:,layer_num));
    % obtain h vector, 9 is the last column
    h(:,layer_num) = V(:,9);
    % obtain the camera matrix
    H(:,:,layer_num)=[h(1,layer_num),h(2,layer_num),h(3,layer_num);
        h(4,layer_num),h(5,layer_num),h(6,layer_num);
        h(7,layer_num),h(8,layer_num),h(9,layer_num)];
end
```

Create h vector and camera(H) zeros matrix firstly. Then, use the for loop and svd matlab built-in function to obtain the h vector and camera matrix. For Step 4 d), since the h vector is already obtained, v_{11}^T, v_{22}^T, and v_{12}^T can be calculated by the vector in Figure 3:

$$v_{ij}^T = \begin{bmatrix} h_{1i}h_{1j} & \begin{pmatrix} h_{2i}h_{1j} \\ +h_{1i}h_{2j} \end{pmatrix} & h_{2i}h_{2j} & \begin{pmatrix} h_{3i}h_{1j} \\ +h_{1i}h_{3j} \end{pmatrix} & \begin{pmatrix} h_{3i}h_{2j} \\ +h_{2i}h_{3j} \end{pmatrix} & h_{3i}h_{3j} \end{bmatrix}$$

Figure 3: Vector for v_{ij}^T

```
% Step 4 d) use the known h values to calculate vT11 vT22 vT12
vT11 = zeros(1,6,5);
vT22 = zeros(1,6,5);
vT12 = zeros(1,6,5);

for layer_num = 1:5
    vT11(:,:,layer_num) = vTij(1,1,layer_num,H);
    vT22(:,:,layer_num) = vTij(2,2,layer_num,H);
    vT12(:,:,layer_num) = vTij(1,2,layer_num,H);
end
```

Create v_{11}^T, v_{22}^T, and v_{12}^T with zero matrices as well. Use a for loop and declare a new function vTij in this part. The implementation of the vTij function is shown below:

```
% declare a new function vTij(i,j,layer_num,H)
function v = vTij(i,j,layer_num,H)
```

```
    v = transpose([H(1,i,layer_num)*H(1,j,layer_num);
    H(1,i,layer_num)*H(2,j,layer_num)+...
    H(2,i,layer_num)*H(1,j,layer_num);
    H(2,i,layer_num)*H(2,j,layer_num);
    H(1,i,layer_num)*H(3,j,layer_num)+...
    H(3,i,layer_num)*H(1,j,layer_num);
    H(2,i,layer_num)*H(3,j,layer_num)+...
    H(3,i,layer_num)*H(2,j,layer_num);
    H(3,i,layer_num)*H(3,j,layer_num)]);
end
```

Step 5 is to build a v matrix function, the equation is shown in Figure 4:

$$\underbrace{\begin{matrix}\text{From first image} \rightarrow \\ \\ \\ \text{From last image} \rightarrow \end{matrix}\begin{bmatrix} v_{12}^T \\ (v_{11}^T - v_{22}^T) \\ \vdots \\ v_{12}^T \\ (v_{11}^T - v_{22}^T) \end{bmatrix}}_{v} \begin{bmatrix} b_{11} \\ b_{12} \\ b_{22} \\ b_{13} \\ b_{23} \\ b_{33} \end{bmatrix} = 0$$

Figure 4: Equation for v Matrix

```
% Step 5 Build the v matrix
v = zeros(10,6);
for layer_num = 1:5
    v(2*layer_num-1,:) = vT12(:,:,layer_num);
    v(2*layer_num,:) = vT11(:,:,layer_num)-vT22(:,:,layer_num);
end
```

The code is shown above is solving the v matrix equation. Create a zeros matrix of v, then using a for loop to obtain each element.

Step 6 is to calculate the b vector using the SVD method. The code is shown below. Assign these elements in the b vector to variable b11 to b33, hence it would be easier for further intrinsic calculation.

```
% Step 6 Calculate the b vector using SVD method
[U,S,V] = svd(v);
b = V(:,6);
b11 = b(1); %make elements clear for further calculation
b12 = b(2);
b13 = b(4);
b21 = b(2);
b22 = b(3);
b23 = b(5);
b31 = b(4);
b32 = b(5);
b33 = b(6);
```

Step 7 is to recover the intrinsic parameter. By the equation provided in the workshop note, we can easily obtain intrinsic parameters, the code is shown below:

```
% Step 7 Recover the intrinsic parameter
y0 = (b12*b13-b11*b23)/(b11*b22-b12^2);
lambda = b33-(b13^2+y0*(b12*b13-b11*b23))/b11;
alpha = sqrt(lambda/b11);
beta = sqrt((lambda*b11)/(b11*b22-b12^2));
gamma = -(b12*alpha^2*beta)/lambda;
x0 = (gamma*y0)/alpha-(b13*alpha^2)/lambda;
K = [alpha,gamma,x0 ; 0,beta,y0 ; 0,0,1];
display(K)
```

The final step is to recover the extrinsic parameters when the checker board is roughly at the same height as the object. The equation for each parameter is provided as well in the workshop note,

```
% Step 8 recover the extrinsic parameters
trans_matrix = zeros(3,4,5); %create 5 3x4 zeros matrices
for layer_num = 1:5
    sigma = 1/norm(inv(K)*H(:,1,layer_num));
    r1 = sigma*inv(K)*H(:,1,layer_num);
    r2 = sigma*inv(K)*H(:,2,layer_num);
    r3 = cross(r1,r2);
    PCWorg = sigma*inv(K)*H(:,3,layer_num);
    trans_matrix(:,1,layer_num) = r1;
    trans_matrix(:,2,layer_num) = r2;
    trans_matrix(:,3,layer_num) = r3;
    trans_matrix(:,4,layer_num) = PCWorg;
    %print results
    disp("Rotation and translation of the No."...
    + layer_num +'image:')
    disp(trans_matrix(:,:,layer_num))
end
```

Use a for loop to calculate six images' extrinsic parameters, and print the rotation and translation matrix for each of them.

The final result of Task 1 is shown in Figure 5.

```
K =

  871.3641    0.2163  300.7358
         0  871.0714  220.9412
         0         0    1.0000

Rotation and translation of the N0.1 image:
    0.9903   -0.0266    0.1358   -3.7843
    0.0162    0.9945    0.0927    3.4239
   -0.1380   -0.0899    0.9852   13.6769

Rotation and translation of the N0.2 image:
    0.9962   -0.0042    0.0877   -3.6639
    0.0187    0.9801   -0.2013    3.5381
   -0.0856    0.2025    0.9764   14.1112

Rotation and translation of the N0.3 image:
    0.9145   -0.0355    0.4033   -2.8868
   -0.0009    0.9964    0.0953    3.5311
   -0.4046   -0.0884    0.9112   15.1090

Rotation and translation of the N0.4 image:
    0.9863   -0.0180   -0.1640   -3.3558
    0.0308    0.9956    0.0840    3.4188
    0.1620   -0.0882    0.9826   13.3021

Rotation and translation of the N0.5 image:
    0.9679   -0.1968   -0.1559   -4.0161
    0.1886    0.9798   -0.0694    2.9622
    0.1664    0.0379    0.9854   15.2806
```

Figure 5: Own Code Results

```
% alphax, skew, alphay, x0, y0, k1, k2 %

832.5 0.204494 832.53 303.959 206.585 -0.228601 0.190353

% Rotation and translation of first image %

0.992759 -0.026319 0.117201 -3.84019
0.0139247 0.994339 0.105341 3.65164
-0.11931 -0.102947 0.987505 12.791

% Rotation and translation of second image %

0.997397 -0.00482564 0.0719419 -3.71693
0.0175608 0.983971 -0.17746 3.76928
-0.0699324 0.178262 0.981495 13.1974

% Rotation and translation of third image %

0.915213 -0.0356648 0.401389 -2.94409
-0.00807547 0.994252 0.106756 3.77653
-0.402889 -0.100946 0.909665 14.2456

% Rotation and translation of fourth image %

0.986617 -0.0175461 -0.16211 -3.40697
0.0337573 0.994634 0.0977953 3.6362
0.159524 -0.101959 0.981915 12.4551

% Rotation and translation of fifth image %

0.967585 -0.196899 -0.158144 -4.07238
0.191542 0.980281 -0.0485827 3.21033
0.164592 0.0167167 0.98622 14.3441
```

Figure 6: Zhang's Results

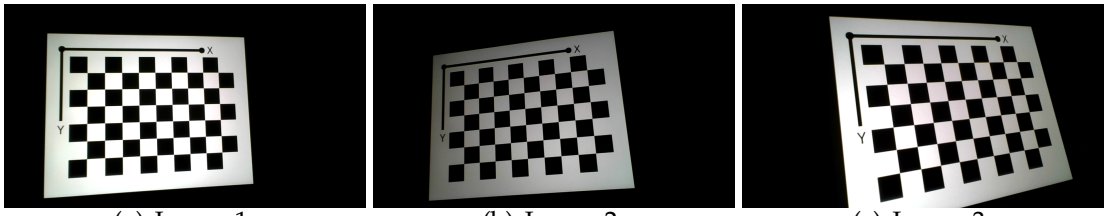
Conclusion to task 1: Assessing the results of the novel calibration performed during this task vs. Zhang's provided results, minor differences appear. The highest point of similarity can be found in the extrinsic parameter calculations via the rotation and translation matrices of each image, where result values from both methodologies were identical to the second decimal place for all images tested. Greater variability is observed in the intrinsic parameter matrix. Since Zhang's algorithm contains the image distortion. We are not required to consider the distortion in this project.

As both the current task and Zhang's methods employed the same initial inputs - in the form of the (X,Y) checkerboard coordinates and (x,y) image points as pre-extracted datasets - this variation necessarily arises during the multi-step matrix manipulation. This may occur through different rules dictating the rounding of values in sequential operations (ex. the process of SVD), or the variation may arise from Zhang employing a slightly altered processes of calculation to the one followed here. Ultimately, the high degree of similarity between the results demonstrates the efficacy of the Task 1 program at achieving camera calibration when using Zhang's 5 image dataset.

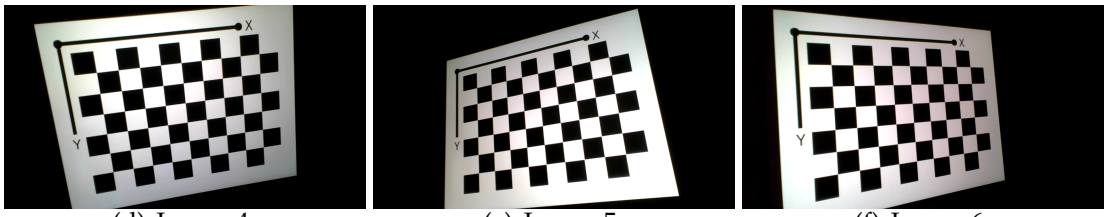
1.2 Task 2: Camera Calibration Experiment

Task 2 employs MATLAB's intrinsic, built-in tools and an adapted instance of the calibration performed during task 1 to obtain the intrinsic and extrinsic parameters of a live camera.

The first step in calibrating the external webcam is to obtain images with variations in orientation, depth of field, and rotation of a printed checkerboard. This process was automated through the provided Code 1 script using the MATLAB Support Package for USB Webcams. Figure 7 shows 6 different images taken by the webcam.



(a) Image 1 (b) Image 2 (c) Image 3

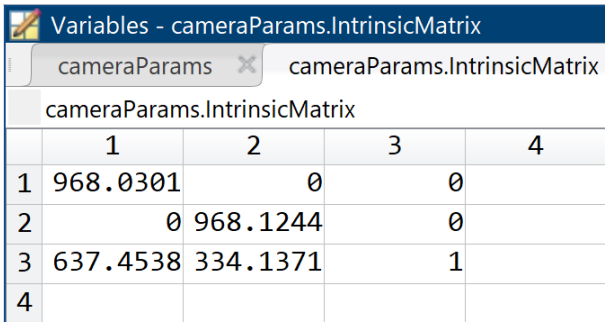


(d) Image 4 (e) Image 5 (f) Image 6

Figure 7: Images by Webcam

Following the generation of test images, the next step involved the inbuilt MATLAB functions detectCheckerboardPoints() to automatically detect the corners in the images, and generateCheckerboardPoints() to generate the physical checkerboard corners based on the given square sizes.

Once the corner data values are extracted and generated, the final inbuilt function - estimateCameraParameters()- calculates the FocalLength, principalPoint, radial distortion, tangential distortion, skew, intrinsic matrix and set of rotation and translation datasets. Figure 8 is an example of intrinsic matrix generated by the bulit-in tool.



Variables - cameraParams.IntrinsicMatrix

cameraParams | cameraParams.IntrinsicMatrix

cameraParams.IntrinsicMatrix

	1	2	3	4
1	968.0301	0	0	
2	0	968.1244	0	
3	637.4538	334.1371	1	
4				

Figure 8: Intrinsic Matrix generated by MATLAB tool from webcam images.

The accuracy of calibration is visualised through the reprojection error bar graph, and the extrinsic parameters are visualised as coordinate positions of each perceived checkerboard in Figure 9 and 10. Figure 9 shows the overall mean error is 0.18 (pixels), which means our webcam is well calibrated.

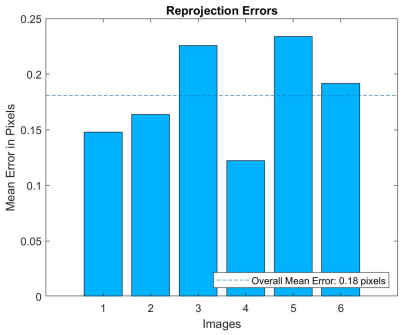


Figure 9: Reprojection Errors

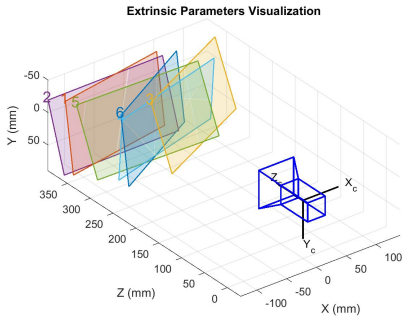


Figure 10: Extrinsic Parameters Visualisation

The next step was to adapt the calibration program from Task 1 to process the webcam images as input. To facilitate this step, the same imagePoints values as were generated by MATLAB - corresponding to the four corner locations of each square in each image - were reused as input.

The generated parameter worldPoints - a matrix of 54[x,y] corner coordinates for the squares - was additionally reused as the checkerboard coordinates input based on the different size and number of squares to the initial Task 1 image set.

To adapt the calibration code to employ these newly generated parameters, we noted that the generated data required structural manipulation to replicate the matrix format of the initial source data. The first step was thus restructuring these matrices: the worldPoints matrix (54x2) was extracted into X(1x54) and Y(1x54) matrices. Similarly, the extracted imagePoints(54x2x6) was converted by rearranging the row and column through the MATLAB permute function.

Further adjustments to the task 1 code involved adapting the generation of phi matrices and subsequent calculations of H and B to accommodate the additional (sixth) image in the task 2 method. Furthermore, the difference in quantity of corner point coordinate values (64x8) vs (54x2) changed the phi zeros to (54*2,9,6), an alteration which was then carried through each subsequent calculation.

The subsequent program - calculating the b vector using SVD, recovering the intrinsic parameters, and ultimately the extrinsic parameters for an image at the same height as the object - remained unchanged other than the aforementioned adjustment in parameter quantities.

```
K =

  968.4662    0.6113  637.5191
         0  968.5667  333.2958
         0         0    1.0000
```

Figure 11: Results of Intrinsic Matrix Generated by Our Code

After running the MATLAB task 2 file, the intrinsic matrix generated by our code is shown in Figure 11, compared with the intrinsic matrix generated by MATLAB built-in tool(Figure 8), we suggest the relationship between these matrices are transpose. Here is the K matrix for built-in tool calculation:

$$K_{Built-in} = \begin{bmatrix} 968.0301 & 0 & 0 \\ 0 & 968.1244 & 0 \\ 637.4538 & 334.1371 & 1 \end{bmatrix}$$

$$K_{Built-in}{}^{T} = \begin{bmatrix} 968.0301 & 0 & 637.4538 \\ 0 & 968.1244 & 334.1371 \\ 0 & 0 & 1 \end{bmatrix}$$

MATLAB and Task 2 Calibration Derived Parameters		
Parameter	MATLAB	Calibration Code
α	968.0301	968.4662
β	968.1244	968.5667
γ	0	0.6113
x_0	637.4538	637.5191
y_0	334.1371	333.2958

By observation, our results are very closed to the transpose of results calculated by MATLAB built-in tool. Due the different calculation configuration between MATLAB built-in tool and our code, this may be the reason why we have to transpose the matrix. Similarly, we need to transpose the translation and rotation matrices for later comparison.

Figure 12 shows the results of rotation and translation of our own code. The first three columns are the rotation matrix, the forth(last) column is the translation vector. The MATLAB built-in tool calculates six rotation and translation matrices separately, and we have to transpose these six matrices for a clear comparison.

```
Rotation and translation of the N0.1 image:
    0.9881    0.0339   -0.1499 -125.7896
   -0.0171    0.9929    0.1196  -32.9126
    0.1530   -0.1158    0.9816  357.1011

Rotation and translation of the N0.2 image:
    0.9842    0.0729    0.1613 -127.4540
   -0.1074    0.9711    0.2143  -18.8733
   -0.1408   -0.2282    0.9636  392.8900

Rotation and translation of the N0.3 image:
    0.9161    0.1606   -0.3681  -38.8093
   -0.0655    0.9650    0.2547  -35.0591
    0.3956   -0.2087    0.8945  283.0984

Rotation and translation of the N0.4 image:
    0.9818    0.1013   -0.1606  -89.1457
   -0.1307    0.9740   -0.1845  -26.9575
    0.1378    0.2022    0.9695  279.8561

Rotation and translation of the N0.5 image:
    0.9389    0.1016    0.3291 -101.2628
   -0.1874    0.9516    0.2429  -14.9796
   -0.2887   -0.2900    0.9125  364.7808

Rotation and translation of the N0.6 image:
    0.8854    0.0761   -0.4586  -92.1748
   -0.0412    0.9954    0.0853  -34.9211
    0.4631   -0.0565    0.8845  278.9360
```

Figure 12: Results of Rotation and Translation Matrix Generated by Our Code

$$T_1 = \begin{bmatrix} 0.9881 & 0.0348 & -0.1497 & -125.8121 \\ -0.0170 & 0.9928 & 0.1186 & -33.2219 \\ 0.1527 & -0.1146 & 0.9816 & 356.5550 \end{bmatrix}$$

$$T_2 = \begin{bmatrix} 0.9843 & 0.0735 & 0.1608 & -127.4544 \\ -0.1073 & 0.9712 & 0.2127 & -19.2048 \\ -0.1405 & -0.2266 & 0.9638 & 392.2776 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.9156 & 0.1604 & -0.3687 & -38.7429 \\ -0.0661 & 0.9645 & 0.2557 & -35.2670 \\ 0.3966 & -0.2097 & 0.8937 & 282.5235 \end{bmatrix}$$

$$T_4 = \begin{bmatrix} 0.9819 & 0.1018 & -0.1598 & -89.1598 \\ -0.1310 & 0.9741 & -0.1843 & -27.1997 \\ 0.1369 & 0.2019 & 0.9698 & 279.5425 \end{bmatrix}$$

$$T_5 = \begin{bmatrix} 0.9393 & 0.1020 & 0.3275 & -101.2827 \\ -0.1872 & 0.9525 & 0.2404 & -15.3044 \\ -0.2874 & -0.2871 & 0.9138 & 364.2018 \end{bmatrix}$$

$$T_6 = \begin{bmatrix} 0.8855 & 0.0769 & -0.4582 & -92.2015 \\ -0.0417 & 0.9954 & 0.0863 & -35.1513 \\ 0.4627 & -0.0573 & 0.8847 & 278.6218 \end{bmatrix}$$

From T_1 to T_6, these transformation matrices are calculated by MATLAB built-in tool. To obtain these matrices, transpose rotation matrices firstly, then adding the translation vector to the forth column. It is clear to observe that results from our code is very closed to results of MATLAB's calculation, which means our web camera calibration is successful.

Conclusion to task2: Assessing the results of the two methods (MATLAB derivation vs. the calibration code from our code) minor differences appear in the results of calculation. Note that when comparing two results, we have to transpose the MATLAB's calculation, this may due to MATLAB built-in tool's different calculation configuration. As both methods employed the same initial inputs - in the form of the (X,Y) checkerboard coordinates and (x,y) image points - this variation necessarily arises during the multi-step matrix manipulation. This may occur through different rules dictating the rounding of values in sequential operations (ex. the process of SVD), or the variation may arise from MATLAB's estimateCameraParameters() function employing a slightly altered processes of calculation. Ultimately, the high degree of similarity between methods demonstrates the efficacy of the Task 2 program at achieving camera calibration when using novel webcam images.

1.3 Task 3: Robot Calibration Experiment

This task will calibrate the virtual robot and find the relationship between the robot frame and the world coordinate frame. The camera parameters have been saved from Task 2. These parameters can also be used in Task 3 as they are almost constants. In order to find a new set of extrinsic parameters, we have to take an image where the checkerboard is well-aligned with the camera (straight up and right in front of the webcam). Use the Code 3 given by the workshop handout. We can take one picture of the checkerboard, the image is shown in Figure 13.

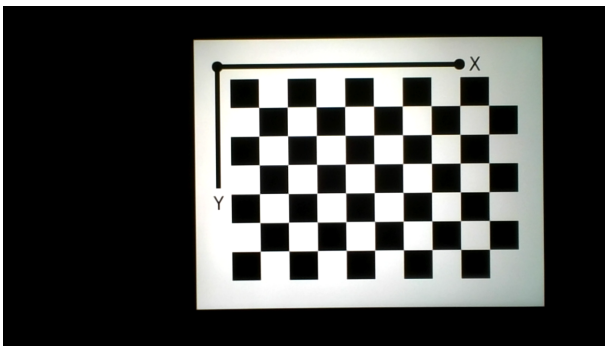


Figure 13: The Picture of the Checkerboard

After taking the picture, we start to calculate the extrinsic parameters. Copy and paste the start code from Code 4 in the workshop handout. Before writing our code, upload the LabCameraParams.mat file to the Task 3 folder, we can obtain a graph (Figure 14) of points in the image coordinates mapped into the points in the world coordinate.

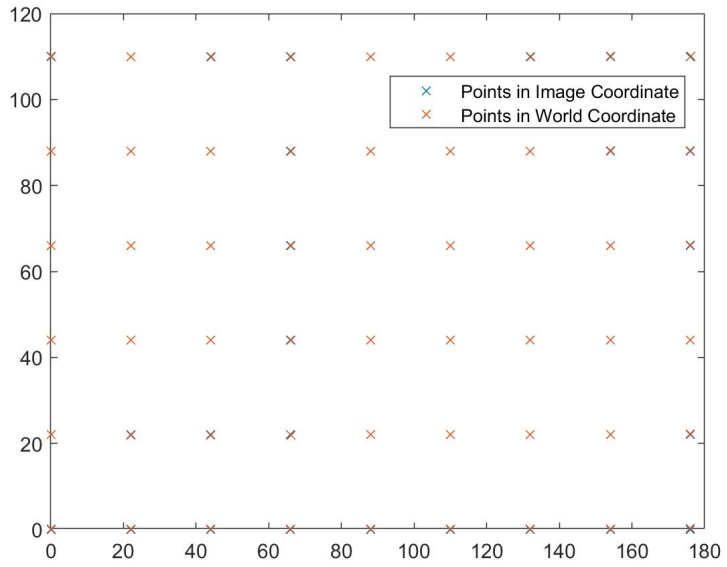


Figure 14: Points in the Image Coordinates Mapped into the Points in the World Coordinate

Open our virtual robot .mlapp file, move to function ForwardButtonPushed(app.event), change the values of CameraIntrinsics based on results from Task 2:

```
CameraIntrinsics = [9.680301351561671e+02,0,0;
                    0,9.681244230787817e+02,0;
                    6.374537532574340e+02,3.341370702892024e+02,1];
```

From the code given by the workshop handout, when I run it, the rotation matrix and translation vector of world with respect to camera is calculated, change the old version in the virtual robot file with these new values. Hence:

```
RCameraWorld = [0.999952630910421,-0.009665852476411,...
-0.001143342132380;
                0.009676606842265,0.999905082543870,...
                0.009807608421833;
                0.001048434713101,-0.009818207516653,...
                0.999951250604655];

PCameraWorld = [-33.311967791009540,-42.571184101341650,...
3.485812262572603e+02];
```

Later, run the virtual robot app, and activate the Calib Tool. Jog the robot's tip to touch four corner points of the virtual checkerboard. Record the Cartesian positions of each corner point. The relationship between the world frame and robot frames is shown in Figure 15:

$$\begin{bmatrix} X_{ri} \\ Y_{ri} \\ Z_{ri} \\ 1 \end{bmatrix} = \begin{bmatrix} \rho_{11} & \rho_{12} & \rho_{13} & t_x \\ \rho_{21} & \rho_{22} & \rho_{23} & t_y \\ \rho_{31} & \rho_{32} & \rho_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ 0 \\ 1 \end{bmatrix}$$

Figure 15: Relationship between the world and robot frames

To solve the unknown parameters, we will use a method proposed by Cashbaugh et al. Then we can obtain four matrices equations:

$$\begin{bmatrix} \rho_{11} \\ \rho_{12} \\ t_x \end{bmatrix} = \begin{bmatrix} \sum X_i X_i & \sum Y_i X_i & \sum X_i \\ \sum X_i Y_i & \sum Y_i Y_i & \sum Y_i \\ \sum X_i & \sum Y_i & n \end{bmatrix}^{-1} \begin{bmatrix} \sum X_{ri} X_i \\ \sum X_{ri} Y_i \\ \sum X_{ri} \end{bmatrix}$$

$$\begin{bmatrix} \rho_{21} \\ \rho_{22} \\ t_y \end{bmatrix} = \begin{bmatrix} \sum X_i X_i & \sum Y_i X_i & \sum X_i \\ \sum X_i Y_i & \sum Y_i Y_i & \sum Y_i \\ \sum X_i & \sum Y_i & n \end{bmatrix}^{-1} \begin{bmatrix} \sum Y_{ri} X_i \\ \sum Y_{ri} Y_i \\ \sum Y_{ri} \end{bmatrix}$$

$$\begin{bmatrix} \rho_{31} \\ \rho_{32} \\ t_z \end{bmatrix} = \begin{bmatrix} \sum X_i X_i & \sum Y_i X_i & \sum X_i \\ \sum X_i Y_i & \sum Y_i Y_i & \sum Y_i \\ \sum X_i & \sum Y_i & n \end{bmatrix}^{-1} \begin{bmatrix} \sum Z_{ri} X_i \\ \sum Z_{ri} Y_i \\ \sum Z_{ri} \end{bmatrix}$$

$$\begin{bmatrix} \rho_{13} \\ \rho_{23} \\ \rho_{33} \end{bmatrix} = \begin{bmatrix} \rho_{11} \\ \rho_{21} \\ \rho_{31} \end{bmatrix} \times \begin{bmatrix} \rho_{12} \\ \rho_{22} \\ \rho_{32} \end{bmatrix}$$

The four corner points (wrt. robot frame) we chose are:

```
% 4 corner points wrt. robot frame
Xr = [-78.3873,-56.3907,-78.7713,-56.7747];
Yr = [177.619,177.235,155.623,155.239];
Zr = [-20.0013,-19.9636,-20.0404,-20.0027];
```

The corresponding location wrt. world frame by checking the Graph in Figure 14:

```
% 4 corner points wrt. world frame
X = [22.0006,44.0047,21.9882,44.0049];
Y = [21.9567,21.9661,43.9583,43.9598];
```

By following Cashbaugh's method, implement these equations by MATLAB code:

```
% start of Task 3
% List of joint angles and Cartesian positions of tool tip
Cartersian = [Xr;Yr;Zr];
theta_1 = [113.8130,107.6495,116.8470,110.0887];
theta_2 = [43.5326,45.2994,47.6261,49.3891];
theta_3 = [-98.8290,102.8517,-108.3607,-112.5781];
JointAngels = [theta_1;theta_2;theta_3];

coefficient = [sum(X.*X), sum(Y.*X), sum(X);
    sum(X.*Y), sum(Y.*Y), sum(Y);
    sum(X), sum(Y), 4];

mat_1 = (inv(coefficient)*[sum(Xr.*X); sum(Xr.*Y); sum(Xr)]);
mat_2 = (inv(coefficient)*[sum(Yr.*X); sum(Yr.*Y); sum(Yr)]);
mat_3 = (inv(coefficient)*[sum(Zr.*X); sum(Zr.*Y); sum(Zr)]);

r_13 = mat_1(1)*mat_1(2);
r_23 = mat_2(1)*mat_2(2);
r_33 = mat_3(1)*mat_3(2);

trans_matrix = [mat_1(1), mat_1(2), r_13, mat_1(3);
    mat_2(1), mat_2(2), r_23, mat_2(3);
    mat_3(1), mat_3(2), r_33, mat_3(3);
    0, 0, 0, 1];
Rrob = trans_matrix(1:3,1:3);
trob = trans_matrix(1:3,4);

disp('TransformationMatrix:')
disp(trans_matrix)
disp(Rrob)
disp(trob)
```

Run the programme, the final results are shown below. Hence, we have obtained rotation matrix and translation vector:

$$\text{Transformation Matrix} = \begin{bmatrix} 0.9994 & -0.0172 & -0.0172 & -99.9938 \\ -0.0172 & -0.9999 & 0.0172 & 199.9545 \\ 0.0017 & -0.0018 & -0.0000 & -20.0000 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}$$

$$Rrob = \begin{bmatrix} 0.9994 & -0.0172 & -0.0172 \\ -0.0172 & -0.9999 & 0.0172 \\ 0.0017 & -0.0018 & -0.0000 \end{bmatrix}$$

$$trob = \begin{bmatrix} -99.9938 \\ 199.9545 \\ -20.0000 \end{bmatrix}$$

1.4 Task 4: Image Processing Experiments

Before doing Task 4, prepare a camera view image by pressing the snap camera button in MATLAB virtual robot app. Follow the steps in the workshop handout, the processed image is shown in Figure 16.

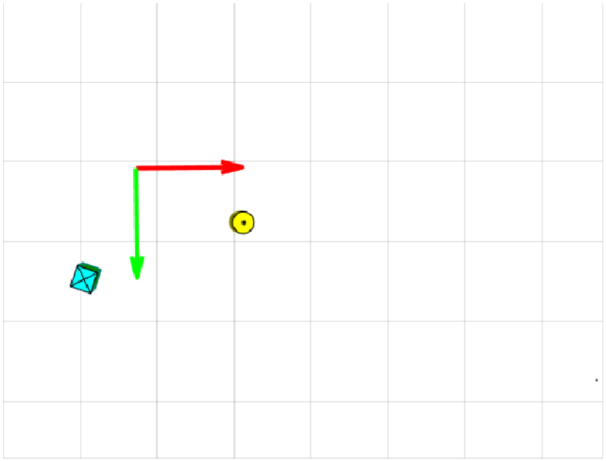


Figure 16: Image of the Camera View

```
    % import image and convert into grey
I = imread('task_4.bmp');
figure, imshow(I),title("Original Image")
IRed = double(I(:,:,1));
IGreen = double(I(:,:,2));
IBlue = double(I(:,:,3));
IGrey = (IRed+IGreen+IBlue)/3;
I = uint8(IGrey);
figure, imshow(I),title("Grey")
```

The code above is to convert the original image to the grey scale:

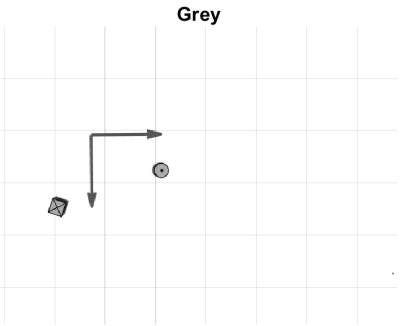


Figure 17: Grey Scale

```
    % Thresholding
I = double(I);
```

```
[m,n] = size(I);
INegative = ones(m, n) * 255  - I;
INegative = uint8(INegative);
I = double(INegative);
[m, n] = size(I);
IThres = zeros(m, n);
for i = 1:m
    for j = 1:n
        if I(i,j) > 180
            IThres(i,j) = 255;
        else
            IThres(i,j) = 0;
        end
    end
end
IThres = uint8(IThres);
figure, imshow(IThres),title('Threshold')
```

Figure 18 shows the thresholding image.

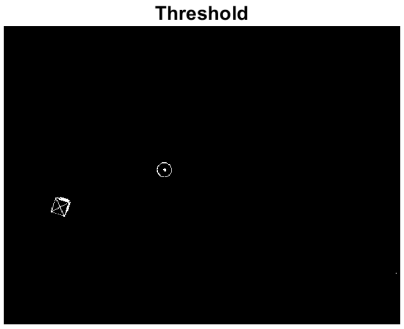


Figure 18: Thresholding

When the shape is the cube:

- Position of the cube with respect to the robot frame is:$(151.74, 100.90, -20.26)$
- Joint angles: $\theta_1 = 146.38$, $\theta_2 = 45.98$, $\theta_3 = -104.65$

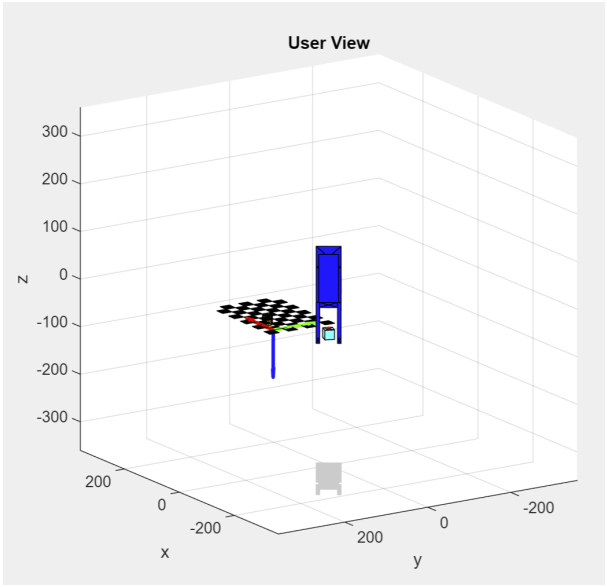


Figure 19: Robot at the Position to Grip the Cube

When the shape is the cylinder:

- Position of the cylinder with respect to the robot frame is: $(-0.8950, 148.26, -19.92)$
- Joint angles: $\theta_1 = 90.35$, $\theta_2 = 52.44$, $\theta_3 = -120.18$

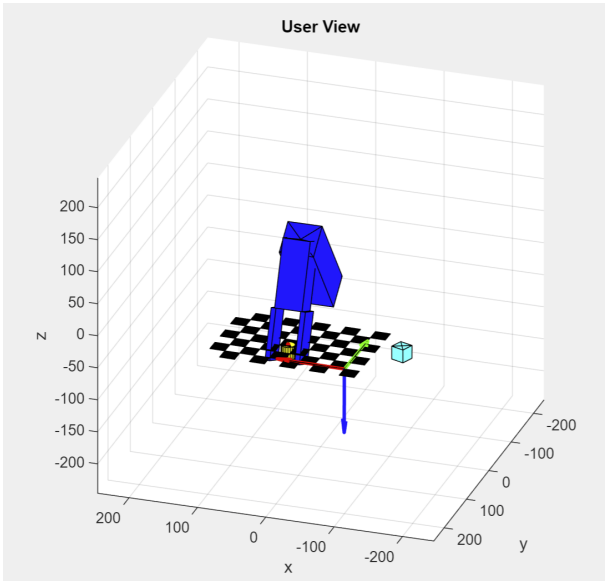


Figure 20: Robot at the Position to Grip the Cylinder

2 Trajectory Planning

The Trajectory Planning task of this Vision assignment focuses on implementing a smooth path from the initial position to the target location or position.

To accomplish this, a series of new buttons were implemented in the UI view. Primarily, built upon the previously established 'Forward' method, the 'Fwd Trajectory' button accomplishes the same joint-angle based target motion but with a smooth trajectory made up of micro-segmented movements. The 'Inv Trajectory button' does the same, executing smooth cartesian-coordinate based motion to the specified target location.

2.1 Forward Trajectory MATLAB Implementation

To implement this, first the initial theta1, theta2 and theta3 positions were obtained from the global variables declared in the FwdButton (Figure 21).

```
% obtain intial theta 1-3 from global variables in FwdButton
global theta1_0;
global theta2_0;
global theta3_0;
```

Figure 21: Establishing initial theta values via importing them as global variables from the FwdButton.

The theta values inputted by the user are then stored in variables.

To establish the desired time profile of the motion, the cubic polynomial is employed, as it ensures an authentic motion that might be viewed in real world robots where the start and ending velocities are zero.

There are four parameters - a0, a1, a2, a3 - which satify four constraints:

```
u(0) = u0;
u(tf) = uf;
u(0) = 0;
u(tf) = 0;
```

This is completed by solving four simultaneous equations:

$$u(0) = a_0 + a_1 0 + a_2 0^2 + a_3 0^3 \tag{2.1}$$

$$u(t_f) = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 \tag{2.2}$$

$$u(0) = a_1 + 2a_2 0 + 3a_3 0^2 \tag{2.3}$$

$$u(t_f) = a_1 + 2a_2 t_f + 3a_3 t_f^2 \tag{2.4}$$

The implementation of these cubic polynomials is as follows:

```
% calculate cubic polynomial constant terms
% for theta 1:
a1_0 = theta1_0;
a1_1 = 0;
a1_2 = 3/tf^2*(theta1_f - theta1_0);
a1_3 = -2/tf^3*(theta1_f - theta1_0);

% for theta 2:
a2_0 = theta2_0;
a2_1 = 0;
a2_2 = 3/tf^2*(theta2_f - theta2_0);
a2_3 = -2/tf^3*(theta2_f - theta2_0);

% for theta 3:
a3_0 = theta3_0;
a3_1 = 0;
a3_2 = 3/tf^2*(theta3_f - theta3_0);
a3_3 = -2/tf^3*(theta3_f - theta3_0);
```

Figure 22: Constant polynomial term derivations.

The next stage is the implementation of a recursive loop. For each 0.1 second interval between 0 and 2 seconds, the joint space schemes based on the cubic polynomials are calculated and updated. These updated values are then displayed in the Theta output fields.

Finally, at each instance of the loop the ForwardButtonPushed command is executed - as the theta values in the display have previously been updated, the ForwardButtonPushed method automatically takes these new values as input. Upon calling the command, there is a 0.001 second pause to visually show the trajectory to the user.

2.2 Inverse Trajectory MATLAB Implementation

The method implemented for the Inverse Trajectory button is precisely as described above, altering only that XYZ coordinates are first taken as the global variables and the InverseButtonPushed is called instead.

The first step is once again to establish the global X, Y, and Z coordinates from the Inverse button. The inputted XYZ values are then extracted to serve as the target endpoint of the robot end effector.

The cubic polynomial constant terms were calculated using the same method as previously described in the forward trajectory section. Furthermore, the same recursive loop was employed at intervals of 0.1 seconds from 0-2 seconds. The cartesian space schemes based on the cubic polynomials then yielded X, Y, and Z values which were displayed in the Edit fields. (Figure 23).

```
for t = 0 : 0.1 : tf
    % cartesian space schemes based on cubic polynomial
    xCubic = ax_0+ax_1*t+ax_2*t^2+ax_3*t^3;
    yCubic = ay_0+ay_1*t+ay_2*t^2+ay_3*t^3;
    zCubic = az_0+az_1*t+az_2*t^2+az_3*t^3;

    % display the trajectory planning for each point
    app.CartesianXEditField.Value = xCubic;
    app.CartesianYEditField.Value = yCubic;
    app.CartesianZEditField.Value = zCubic;

    % call InvButton to run
    InverseButtonPushed(app,event)

    % pause and continue the for loop every 0.001s to show the trajectory
    pause(0.001)
end
```

Figure 23: MATLAB for loop implementation of gradual inverse trajectory motion.

As above, the InverseButtonPushed function was called at each interval, using the updated edit field values as input. The same 0.001 seconds pause was used at the end of each cycle.

CONCLUSION: The result of the recursive loops through either joint angle positions or cartesian space yield a gradual movement from the intiial position to the inputted target.

3 Pick and Place Demonstration

The pick and place demonstration took place on December 17th, 2020. This demonstrated the smooth motion resulting from the trajectory planning (described in the previous section), as well as the coordinated movement of a Cube and Cylinder upon selection of the 'Grab Cube' or 'Grab Cylinder' button.

The 'Approach Object' button allows the robot to approach an object in the default, pre-established coordinates, or user inputted X and Y (See Figure 24). When pressed, the arm moves in a smooth, obstacle-free path to the target location.

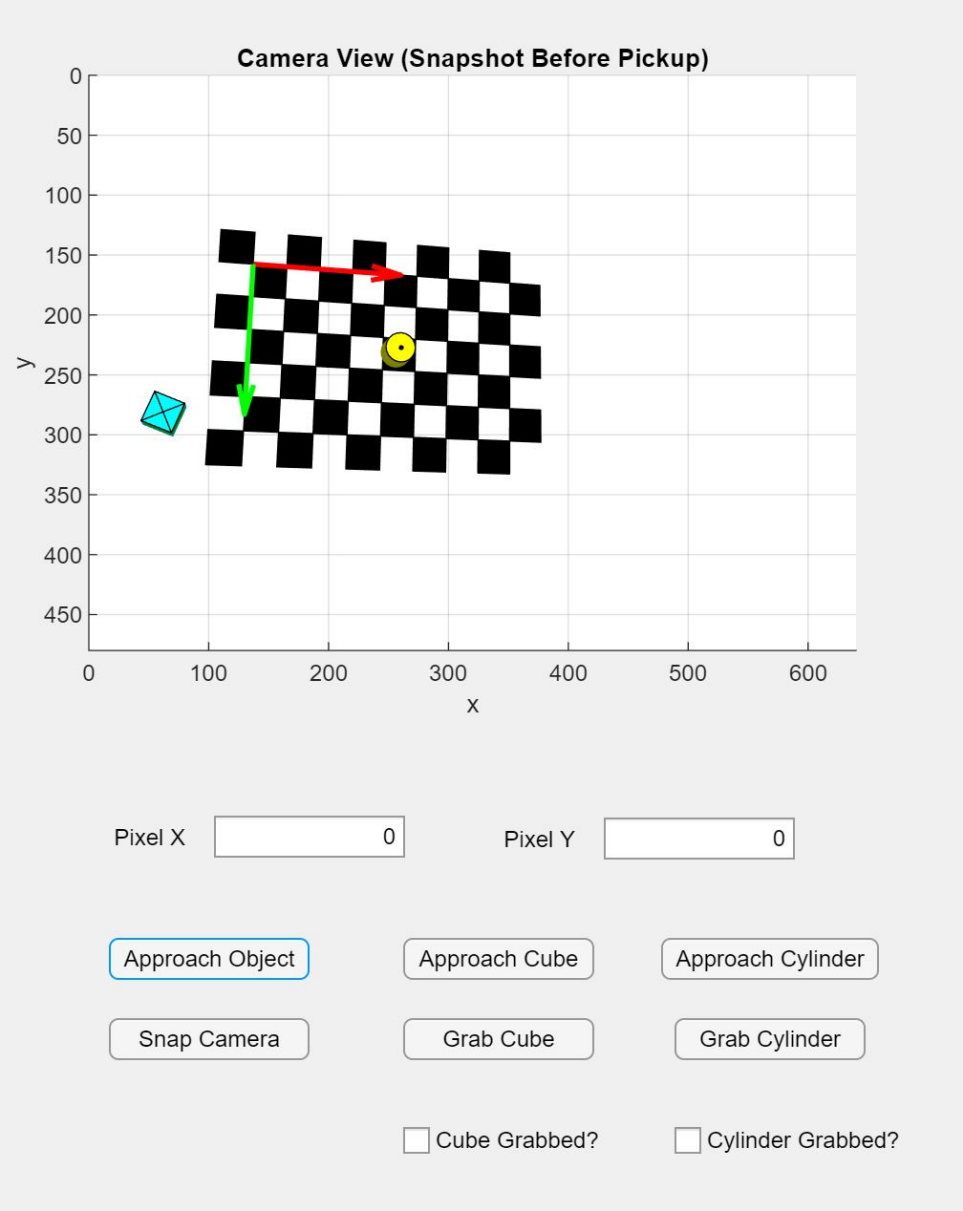


Figure 24: Approach buttons implemented in MATLAB robot UI

First, the initial XYZ position of the robot were taken as global variables from the InvButton. Next, the inputted final XYZ values were stored.

Employing the process previously described in the trajectory buttons, the cubic polynomial constant terms were calculated for X, Y and Z.

As previously described, the robot arm then moves above(50mm) the object in 2 seconds, holds position for 2 seconds, and approaches the target in another 2 seconds.

3.1 Robot-to-World Transformations

To establish the basis for external manipulations using the robot, the following calculations were performed to establish the transformation betweent the robot and

world frames.

Transformation between robot and world frames, where the world is the small frame in the user view and the robot is the real cartesian XYZ values:

$$thetaXRobotWorld = 180.1$$

$$Rx = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos180.1 & -sin180.1 \\ 0 & sin180.1 & cos180.1 \end{bmatrix}$$

$$thetaYRobotWorld = 0.1$$

$$Ry = \begin{bmatrix} cos0.1 & 0 & sin0.1 \\ 0 & 1 & 0 \\ -sin0.1 & 0 & cos0.1 \end{bmatrix}$$

$$thetaZRobotWorld = 1.0$$

$$Rz = \begin{bmatrix} cos1 & -sin1 & 0 \\ sin1 & cos1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For the Rotation only, the following matrix was derived (4 d.p):

$$R_{world}^{Robot} = R_x \times R_y \times R_z \approx \begin{bmatrix} 0.9998 & -0.0175 & 0.001 \\ -0.0175 & -0.9998 & 0.0017 \\ 0.0017 & -0.0018 & -1.0000 \end{bmatrix}$$

The transformation matrix is as follows:

$$T_{world}^{Robot} = \begin{bmatrix} & R_{world}^{Robot} & & P_{world}^{Robot} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus, the transformation matrix with derived values plugged in is:

$$T_{world}^{Robot} = \begin{bmatrix} 0.9998 & -0.0175 & 0.0017 & -100 \\ -0.0175 & -0.9998 & 0.0017 & 200 \\ 0.0017 & -0.0018 & -1.0000 & -20 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.2 CUBE Approach and Grab

To approach and grab the cube (referred to as 'box')

$$T_{box}^{world} = \begin{bmatrix} cos20 & -sin20 & 0 \\ sin20 & cos20 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Since the Position vector of the box with respect to the world is a variable (as the box can move), the

$$P_{box}^{world} = \begin{bmatrix} -50 \\ 100 \\ 0 \end{bmatrix}$$

matrix is default. Let

$$P_{box}^{world} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

Therefore, the transformation matrix is as follows (by the same method as above described):

$$T_{Robot}^{world} = \begin{bmatrix} & R_{box}^{world} & & P_{box}^{world} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} cos20 & -sin20 & 0 & a \\ sin20 & cos20 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{world}^{Robot} \times T_{box}^{world} = T_{box}^{Robot}$$

$$= \begin{bmatrix} 0.9998 & -0.0175 & 0.0017 & -100 \\ -0.0175 & -0.9998 & 0.0017 & 200 \\ 0.0017 & -0.0018 & -1.0000 & -20 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} cos20 & -sin20 & 0 & a \\ sin20 & cos20 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} X & X & X & 0.9998a - 0.0175b + 0.0017c - 100 \\ X & X & X & -0.0175a - 0.9998b + 0.0017c + 200 \\ X & X & X & 0.0017a - 0.0018b - c - 20 \\ X & X & X & X \end{bmatrix}$$

Therefore:

$$P_{box}^{Robot} = \begin{bmatrix} 0.9998a - 0.0175b + 0.0017c - 100 \\ -0.0175a - 0.9998b + 0.0017c + 200 \\ 0.0017a - 0.0018b - c - 20 \end{bmatrix}$$

Having established the motion of the robot end effector with respect to the box, the model was thus adapted such that the box and end-effector move together. For this to be true, RobotPbox = robotPE. Thus:

$$\begin{bmatrix} 0.9998a - 0.0175b + 0.0017c - 100 \\ -0.0175a - 0.9998b + 0.0017c + 200 \\ 0.0017a - 0.0018b - c - 20 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix}$$

$$\begin{bmatrix} 0.9998 & -0.0175 & 0.0001 \\ -0.0175 & -0.9998 & 0.001 \\ 0.0017 & 0.0018b & -1 \end{bmatrix} \times \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} x_p + 100 \\ y_p - 200 \\ z_p + 20 \end{bmatrix}$$

To isolate the ABC matrix, the inverse of the translation matrix is employed:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0.9998 & -0.0175 & 0.0001 \\ -0.0175 & -0.9998 & 0.001 \\ 0.0017 & 0.0018b & -1 \end{bmatrix} \times \begin{bmatrix} x_p + 100 \\ y_p - 200 \\ z_p + 20 \end{bmatrix}$$

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0.9998(x_p + 100) - 0.0175(y_p - 200) + 0.0017(zp + 20) \\ -0.0175(x_p + 100) - 0.9998(y_p - 200) + 0.0017(z_p + 20) \\ 0.0017(x_p + 100) - 0.0018(y_p - 200) - (z_p + 20) \end{bmatrix}$$

Therefore,

$$P_{box}^{world} = \begin{bmatrix} 0.9998(x_p + 100) - 0.0175(y_p - 200) + 0.0017(zp + 20) \\ -0.0175(x_p + 100) - 0.9998(y_p - 200) + 0.0017(z_p + 20) \\ 0.0017(x_p + 100) - 0.0018(y_p - 200) - (z_p + 20) \end{bmatrix}$$

3.3 CYLINDER Approach and Grab

For the cylinder - the same process as above was applied. Given -

$$P_{cylinder}^{world} = \begin{bmatrix} 100 \\ 50 \\ 0 \end{bmatrix}$$

$$T_{world}^{Robot} = \begin{bmatrix} 0.9998 & -0.0175 & 0.0017 & -100 \\ -0.0175 & -0.9998 & 0.0017 & 200 \\ 0.0017 & -0.0018 & -1.0000 & -20 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

There is no rotation between the world and cylinder frames, and thus

$$R_{cylinder}^{world} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus we derive the same transformation matrix employing parameters A, B and C, and use the same method as above where worldPcylinder is a variable.

$$T_{cylinder}^{world} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{world}^{Robot} \times T_{cylinder}^{world} = T_{cylinder}^{Robot}$$

$$= \begin{bmatrix} 0.9998 & -0.0175 & 0.0017 & -100 \\ -0.0175 & -0.9998 & 0.0017 & 200 \\ 0.0017 & -0.0018 & -1.0000 & -20 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

By observation, the 4th column of robotTcylinder and robotTbox are the same. Thus -

$$P_{cylinder}^{world} = P_{box}^{world} = \begin{bmatrix} 0.9998(x_p + 100) - 0.0175(y_p - 200) + 0.0017(zp + 20) \\ -0.0175(x_p + 100) - 0.9998(y_p - 200) - 0.0017(z_p + 20) \\ 0.0017(x_p + 100) + 0.0018(y_p - 200) - (z_p + 20) \end{bmatrix}$$

$$P_{cylinder}^{Robot} = P_{box}^{Robot} = \begin{bmatrix} 0.9998a - 0.0175b + 0.0017c - 100 \\ -0.0175a - 0.9998b + 0.0017c + 200 \\ 0.0017a - 0.0018b - c - 20 \end{bmatrix}$$

Where:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0.9998(x_p + 100) - 0.0175(y_p - 200) + 0.0017(zp + 20) \\ -0.0175(x_p + 100) - 0.9998(y_p - 200) - 0.0017(z_p + 20) \\ 0.0017(x_p + 100) + 0.0018(y_p - 200) - (z_p + 20) \end{bmatrix}$$

3.4 Final MATLAB Implementation

The final implementation of the approach function (as exemplified in figure 25 by the GrabCubeButtonPushed function) first requires the default position vector of the cube with respect to the world.

```
% Button pushed function: GrabCubeButton
function GrabCubeButtonPushed(app, event)
    % default position vector of cube wrt. world
    a = -50;
    b = 100;
    c = 0;
    % transfer to default position vector of cude wrt. robot
    x_cube = 0.9998*a-0.0175*b+0.0017*c-100;
    y_cube = -0.0175*a-0.9998*b+0.0017*c+200;
    z_cube = 0.0017*a-0.0018*b-c-20;

    % test whether the cube position and current end-effector XYZ
    % are similar or in a suitable range
    if x_cube < app.CartesianXEditField.Value+1 && x_cube > app.CartesianXEditField.Value-
    && y_cube < app.CartesianYEditField.Value+1 && y_cube > app.CartesianYEditField.Value-
    && z_cube < app.CartesianZEditField.Value+1 && z_cube > app.CartesianZEditField.Value-
        app.CubeGrabbedCheckBox.Value = true;
        ForwardButtonPushed(app, event);
    else
        app.CubeGrabbedCheckBox.Value = false;
    end
end
```

Figure 25: MATLAB implementation of Grab Cube Button. The exact same method was employed for the Grab Cylinder function.

In each of the Grab Cube and Grab Cylinder functions, once the default position vector of the cube wrt world is defined and transfered to the default position vector of cube wrt robot (employing the matrices explained above), a test is run to assess whether the object position and current end-effector XYZ coordinate are in a similar (suitable) range to make contact. If they are, then the checkbox value becomes ticked.