Vision and Trajectory Planning

Virtual Robot Group Project

Robotics for EE - ELEC0140

AUTHORS - TEAM 22

Yousef Mahmoud - 18014580

Gianluca Traversa - 18018694

Teng Wang - 18008802

Virtual Robot Project Group Report 2 - 2020/21

Department of Electronic and Electrical Engineering

University College London

January 7, 2021

Table of Contents

Abstract

In this report, we will be reporting the workflow used and interpreting the results achieved whilst conducting the tasks from our vision/image processing and trajectory planning workshops for our virtual robot as part of the ELEC0140 module.

To summarise, Section 1 of this report will be dealing with vision-guided motion and will walk through the steps of camera calibration, robot calibration and image processing. With this, we were able to calibrate our devices and allow our MATLAB code to successfully recognise the shape of our object, calculate the position as well as calculate the joint angles required to arrive at them.

On the other hand, Section 2 will expand on this by following how we generated a smooth trajectory for the robot to follow rather than move instantly towards the target. We used the cubic polynomial method and successfully guided our robot through a smooth trajectory to a point above the target, then down towards the target. This was successful when both using accurately calculate values for the position and the values calculated in the image processing workshop. The final part included activating the gripper on our robot and coding the application such that the object would move with the end effector of our robot. We managed to complete this and move both objects with the gripper and with the correct orientation.

1 Vision Workshop

In the vision workshop, we were given 4 tasks to systematically guide us through calibrating our camera by finding its extrinsic and intrinsic parameters in Tasks 1 and 2 using images of a checkerboard. We were then tasked with performing a robot calibration in Task 3 to find the relationship between the world and robot co-ordinate frames as well as the camera frame. Task 4 involves image processing to find the shape, position, and orientation of our two objects using elements from the previous tasks as well. In each of the subsections below, we will go through the process carried out to complete these tasks and interpret any useful results we might achieve.

1.1 Task 1 - Camera Calibration Given Checkerboard Data

For Task 1, we were asked to carry out a camera calibration using a Microsoft Research dataset created by Zhang complete with the checkerboard data and the corner points had been pre-calculated which is the key difference between Task 1 and 2. In task 2, we will be carrying out similar calculations, however, this was simplified and used to allow us to understand how to find the extrinsic and intrinsic parameters of a camera before asking us to use corner recognition algorithms.

Step 1 - Downloading the files:

We downloaded the items in the Task1Data folder which included txt files called Model.txt which containted the co-ordinates of the corner points for the physical prinout of the checkerboard. Since there were 64 squares in Zhang's checkerboard, there are 256 (64 $\times$ 4) corner points. Each square has 4 corner points which are each made up of 2 co-ordinates so the co-ordinates for all the corner points of a square are represented by 8

numbers. So there were 512 numbers in total (64 × 8) to represent the co-ordinates of the corner points.

Similarly, there were 5 image files of the checkerboard in different orientations which would be used to find the corner points for the calibration if they were not given. The pixel co-ordinates of the corner points for the checkerboard in each of those 5 images can be found across a set of 5 txt files called data1.txt to data5.txt which also included

A final file called ZhangsResults.txt was there to be used at the end to compare the results obtained by our algorithm with those obtained by the creator of the algorithm to assess if we have written our algorithm correctly.

Step 2 - Opening and analysing the CodeStart.m file:

The CodeStart.m file was given to act as a starting point for our code. We analysed to ensure we knew what it did and added some comments. As can be read in our code comments, the file was used to first load Model.txt as a 64 by 8 matrix. The x and y co-ordinates are then extracted into separate 64 by 4 matrices (as done by lines 11 and 12 of Listing 1.1.1). Line 11 takes all the odd-numbered columns to populate the X matrix and Line 12 takes all the even numbered columns to populate Y as those contain the y-co-ordinates. Lines 18 and 19 then reads down each column of the 64x4 matrices to create two large 256 by 1 matrix with a single column containing all the x co-ordinates of the corner points and the other 256 by 1 matrix contains all the y co-ordinates. This is then transposed so that X contains all the x co-ordinates on a single row and Y contains all the y co-ordinates on a single row.

In Listing 1.1.2, a similar process is carried out by loading the corner points found in data1.txt, data2.txt, ... data5.txt and then using them to populate 64 by 8 matrices which are then split into two 64 by 4 matrices with one containing the x co-ordinates and one containing the y co-ordinates. This is then used to populate a matrix with 5 layers with each layer containing the co-ordinates for each of the 5 different images. The co-ordinates from data1.txt go into the first layer, data2.txt into the second layer etc. Since all the code is similar, only the code for data1.txt is included in Listing 1.1.2. The rest can be found in the MATLAB file uploaded with this assignment.

Each of the 5 'layers' of the matrix has a 2 by 256 matrix with the first row containing the x co-ordinates and the second row containing the y co-ordinates for the corner points of the image represented by the 'layer'.

Step 3 - Start developing the algorithm by following the theory instructions from the workshop:

Starting from Line 88 of the 'Vision_Task1.m' file, we began writing our own code to calibrate the camera by finding the intrinsic parameters. Calibrating a camera to find the relationship between the image and world frames, we must find the intrinsic parameters of the camera (e.g. focal length and scaling factor) as well the extrinsic parameters (translation and rotation between the image and world co-ordinate frames). In this case, we are more concerned with extrinsic parameters but will calculate both. Our extrinsic parameters can be called ${}^{C}_{W}P$ and ${}^{C}P_{W_{org}}$ which include r_1, r_2, r_3 and ${}^{C}P_{W_{org}}$. Whilst the intrinsic parameters are contained in the matrix K seen below:

$$K = \begin{bmatrix} \alpha & \gamma & x_0 \\ 0 & \beta & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.1}$$

By looking at Sections 3.2 and 3.3 of the workshop we are able to identify what the H matrix (camera matrix) is and can write is as seen below and in our case is simplified as we use a planar checkerboard so the z-axis value is always 0:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \approx K \begin{bmatrix} r_1 & r_2 & r_3 & {}^{C}P_{W_{org}} \end{bmatrix} \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = K \begin{bmatrix} r_1 & r_2 & {}^{C}P_{W_{org}} \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = H \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \tag{1.2}$$

Using this camera matrix we can eventually go on to find the intrinsic and extrinsic camera parameters. Using Eqn 1.2, we can say that our camera matrix can be written as:

$$H = K \begin{bmatrix} r_1 & r_2 & {}^{C}P_{W_{org}} \end{bmatrix} \tag{1.3}$$

Since for a single image, the values of x, y, X and Y are known we can find the H parameter from there and then use it to calculate K which contains our intrinsic parameters and our extrinsic parameters r_1, r_2 and ${}^{C}P_{W_{org}}$.

Step 4 - Populating the ϕ matrix to find H

To find the H-matrix, we analyse the corner points for an image that we have taken of the checkerboard. In this case, these corner points were already given to us and we loaded them into the matrix imagePoints. However, in Task 2 we will need to find them ourselves using a corner detection algorithm. We need at least four corner points to find H, but in this case we have 256 which is better as more corner points make for a more accurate calculation of the parameters.

For a single corner point i, we have the pixel co-ordinates of the corner point (x_i,y_i) in the images taken as well as the co-ordinates of that corner point on the actual physical printout of the checkerboard (X_i,Y_i). Using equation 1,2, we can thus say we have:

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \approx \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ 1 \end{bmatrix} \tag{1.4}$$

According to Section 3.4 of the workshop, the proportional sign also means that the left hand side of 1.3 is the scalar multiple of the right hand side and hence by performing a cross product we get an answer of zero as seen below:

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \times \left(\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{1.5}$$

Therefore, we can expand this to become:

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \times \begin{bmatrix} h_{11}X_i + h_{12}Y_i + h_{13} \\ h_{21}X_i + h_{22}Y_i + h_{23} \\ h_{31}X_i + h_{32}Y_i + h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{1.6}$$

Knowing that $x_i(h_{11}X_i + h_{12}Y_i + h_{13}) = 0$, $y_i(h_{21}X_i + h_{22}Y_i + h_{23}) = 0$, and $1(h_{31}X_i + h_{32}Y_i + h_{33}) = 0$ then we can write that:

$$\begin{gathered} x_i(h_{31}X_i + h_{32}Y_i + h_{33}) - (h_{11}X_i + h_{12}Y_i + h_{13}) = x_i(0) - (0) = 0 \\ x_i(h_{31}X_i + h_{32}Y_i + h_{33}) - h_{11}X_i - h_{12}Y_i - h_{13} = 0 \end{gathered} \tag{1.7}$$

$$\begin{gathered} y_i(h_{31}X_i + h_{32}Y_i + h_{33}) - (h_{11}X_i + h_{12}Y_i + h_{13}) = y_i(0) - (0) = 0 \\ y_i(h_{31}X_i + h_{32}Y_i + h_{33}) - h_{11}X_i - h_{12}Y_i - h_{13} = 0 \end{gathered} \tag{1.8}$$

When re-written in using the linear parameters form to find solve simultaneous equations (in this case to find the h-values), we can we write the equations from 1.7 and 1.8 above as:

$$\begin{bmatrix} 0 & 0 & 0 & X_i & Y_i & 1 & -y_iX_i & -y_iY_i & -y_i \\ X_i & Y_i & 1 & 0 & 0 & 0 & -x_iX_i & -x_iY_i & -x_i \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{1.9}$$

Equation 1.9 only uses one corner point on the checkerboard, however, we can then stack all the corner points in the left hand matrix which will in turn increase the number of zeros in the right hand matrix. For a matrix with n corner points, this is the general form of the equation as also seen in the workshop theory:

$$\begin{bmatrix} 0 & 0 & 0 & X_1 & Y_1 & 1 & -y_1X_1 & -y_1Y_1 & -y_1 \\ X_1 & Y_1 & 1 & 0 & 0 & 0 & -x_1X_1 & -x_1Y_1 & -x_1 \\ 0 & 0 & 0 & X_2 & Y_2 & 1 & -y_2X_2 & -y_2Y_2 & -y_2 \\ X_2 & Y_2 & 1 & 0 & 0 & 0 & -x_2X_2 & -x_2Y_2 & -x_2 \\ & & & \dots & \dots & \dots & \dots & \dots & \dots \\ & & & \dots & \dots & \dots & \dots & \dots & \dots \\ & & & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & X_n & Y_n & 1 & -y_nX_n & -y_nY_n & -y_n \\ X_n & Y_n & 1 & 0 & 0 & 0 & -x_nX_n & -x_nY_n & -x_n \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ . \\ . \\ . \\ 0 \\ 0 \end{bmatrix} \tag{1.10}$$

The left hand side matrix containing the corner points is usually denoted as ϕ and hence we call it phi on MATLAB and initialise it to be a zeros matrix with dimensions 512 by

9 by 5. The 512 is due to the 256 corner points and there being 2 rows per corner pint and the 9 is due to that there must be 9 columns to multiply ϕ with the 9 by 1 matrix containing the 9 h-values. There are 5 layers as each of these calculations is done for all the corner points of one image but we have 5 images so we make 5 of the 512 by 9 matrices to complete the calculation of H for each image. The loop used to populate the phi matrix from the imagePoints matrix generated in Step 2 is seen in listing 1.1.3.

Step 5 - Carrying out SVD to find the H matrix

The problem with solving an equation like 1.10 is that the right hand side is all zeros and so assuming that all the h-values are 0 is an insignificant solution. To achieve the solution, we therefore must use singular value decomposition (SVD) as we can note that h is the null space of ϕ. This is a long winded process, however, we can use the readily built in MATLAB svd function which outputs 3 matrices (U, S (sometimes denoted as Σ) and V).

The SVD of ϕ will be equal to:

$$\phi = U\Sigma V^T = USV^T \tag{1.11}$$

We are told that the h values will be in the column of V corresponding to the smallest singular value of Σ (written as S in MATLAB). However, in SVD on MATLAB the the singular values are arranged in descending order and so the h values will be found in the final column of V and hence we can call this column using line 4 in Listing 1.1.4. This column has the H values for one of the images of the checkerboard.

Again this is repeated 5 times and the H matrix on our MATLAB program has 5 columns as each column will contain the H values for one of the 5 images (explained better in the comments in the .m file).

Step 6 - Extracting intrinsic parameters from the H matrix:

Now that we have the h values and hence the h-matrix we can begin to extract the intrinsic parameters to calculate the values and populate the matrix K. To begin extracting these parameters, we must begin forming the v matrices which will be manipulated to find the B matrix whose values will be used to find the intrinsic parameters.

Recalling equation 1.3, we know that r_1 and r_2 are orthonormal hence:

$$\begin{gathered} r_1^T r_2 = 0 \\ r_1^T r_1 = r_2^T r_2 (= 1) \end{gathered} \tag{1.12}$$

This along with 1.3 can allow us to make the conclusion that:

$$\begin{gathered} h_1^T K^{-T} K^{-1} h_2 = 0 \\ h_1^T K^{-T} K^{-1} h_1 = h_2^T K^{-T} K^{-1} h_2 \end{gathered} \tag{1.13}$$

A square matrix B is defined to simplify 1.14. This matrix is an upper triangular matrix which is symmetric and whose parameters b_{21}, b_{31} and b_{32} are equal to zero. It is defined below:

$$B = K^{-T}K^{-1} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \tag{1.14}$$

By substituting 1.14 into 1.13, we get:

$$\begin{aligned} h_1^T B h_2 = 0 \\ h_1^T B h_1 = h_2^T B h_2 \end{aligned} \tag{1.15}$$

More generally this can be written as:

$$\begin{aligned} h_i^T B h_j = \begin{bmatrix} h_{1i} & h_{2i} & h_{3i} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \begin{bmatrix} h_{1j} \\ h_{2j} \\ h_{3j} \end{bmatrix} \\ = \begin{bmatrix} h_{1i}h_{1j} & (h_{2i}h_{1j} + h_{1i}h_{2j}) & h_{2i}h_{2j} & (h_{3i}h_{1j} + h_{1i}h_{3j}) & (h_{3i}h_{2j} + h_{2i}h_{3j}) & h_{3i}h_{3j} \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \\ b_{22} \\ b_{23} \\ b_{33} \end{bmatrix} \end{aligned} \tag{1.16}$$

We can then generate our v matrices by saying the v transposed is equal to:

$$v_{ij}^T = \begin{bmatrix} h_{1i}h_{1j} & (h_{2i}h_{1j} + h_{1i}h_{2j}) & h_{2i}h_{2j} & (h_{3i}h_{1j} + h_{1i}h_{3j}) & (h_{3i}h_{2j} + h_{2i}h_{3j}) & h_{3i}h_{3j} \end{bmatrix} \tag{1.17}$$

We can see the following relations between v and b values by combining 1.15 and 1.16:

$$\begin{aligned} h_1^T B h_2 = v_{12}^T b \\ h_1^T B h_1 = v_{11}^T b \\ h_2^T B h_2 = v_{22}^T b \\ h_1^T B h_1 = h_2^T B h_2 \therefore v_{11}^T b - v_{22}^T b = 0 \end{aligned} \tag{1.18}$$

This can be put into matrix form as a simultaneous equation since we have the values for h to be substituted into v and hence these can be used to calculate the b-values:

$$\begin{bmatrix} v_{12}^T \\ (v_{11}^T - v_{22}^T) \end{bmatrix} b = 0 \tag{1.19}$$

Equation 1.19 only uses one H and hence is only using one image. To use all 5, we must again stack the matrices similar to how we were using them when we stacked the corner pints in 1.10. This can be shown as for 5 images with the first two rows of v using h values from the first image, row 3 and 4 using the h values from the second image etc.

$$\begin{bmatrix} v_{12}^T \\ (v_{11}^T - v_{22}^T) \\ v_{12}^T \\ (v_{11}^T - v_{22}^T) \\ v_{12}^T \\ (v_{11}^T - v_{22}^T) \\ v_{12}^T \\ (v_{11}^T - v_{22}^T) \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \\ b_{22} \\ b_{23} \\ b_{33} \end{bmatrix} = 0 \tag{1.20}$$

Similar to step 5, the right hand side is zero and so we must use SVD to find the b values. We recall the equation for SVD from equation 1.11 and use the appropriate MATLAB function which is seen in the Listing 1.1.5. The b values will be in the V column with the lowest singular value of Σ which similar to in Step 5 will be the last column of V in MATLAB and we can see it being called in Line 19 of the Listing 1.1.5.

After finding the B values, we recall equation 1.14 and we are told in the workshop instructions that we know the B values up to a certain scale factor. λ.

Then we use the equations given on page 14 of the workshop to find the intrinsic parameters using the b values. The intrinsic parameters to be found are α, β, x_0, y_0 and skewness (λ). We also calculate the scale factor up to which the elements of B are known (this scale factor is needed to ensure all the parameters a correctly scaled given that the 3,3 element of the K matrix is equal to one.

These equations are used in MATLAB with the B values extracted as seen in lines 1-7 of Listing 1.1.6 and these values are then displayed using lines 10-19 of Listing 1.1.6. These values can then later be written in the K matrix (first introduced in equation 1.1)

Step 7 - Extracting extrinsic parameters from the H matrix

Finding the extrinsic parameters is done by defining a scale as below and then substituting into the relevant equations. We first define our scale factor as:

$$\sigma = \frac{1}{||K^{-1}h||} \tag{1.21}$$

This scale factor is then used along with all the parameters below and their equations which were given on page 15 of the workshop sheet:

$$r_1 = \sigma K^{-1} h_1 \tag{1.22}$$

$$r_2 = \sigma K^{-1} h_2 \tag{1.23}$$

$$r_3 = r_1 \times r_2 \tag{1.24}$$

$${}^{C}P_{W_{org}} = \sigma K^{-1} h_3 \tag{1.25}$$

Unlike with the intrinsic parameters, the set of extrinsic parameters is only valid for one image and so we will have 5 sets of extrinsic parameters to be stored in the . The

calculation of the parameters and populating them in the matrix R is seen in Listing 1.1.7.

Step 8 - Comparing our results

By looking in the command window we can see the results produced and then revisit the ZhangsResults.txt file to compare the parameters we have achieved with his. They are not exactly the same, however, they seem to be very close and this is a good sign that our algorithm and code to achieve the parameters is correct having followed the worksheet. If the differences had been large, it would have meant that we had made a mistake somewhere along the way when completing this Task. The slight differences seen between our results and Zhang's are due to him using optimisation in his calibration algorithm which we did not use and so there will obviously be a slight discrepancy that is not large enough to discredit our results or code.

An overall interpretation of our results and the process is that to calculate the parameters we must first extact the corner points find the h-matrix using SVD. Similarly, we use these h-values to find the B-matrix using SVD. The B-matrix then allows us to find the intrinsic parameters (the K matrix) which in turn when used with the h-values can give us our extrinsic parameters. By comparing our results to Zhang's results, we see that our algorithm is correct as our results were very similar. Any discrepancies were due to Zhang conducting his experiment with optimised settings whilst we did not use this optimisation, however, the difference is small enough for us to have confidence in our code.

Now, we will be moving on to Task 2 which essentially uses the same mathematics but since we will be using our own checkerboard we will have to use corner detection algorithms

Listing 1.1.1: Extracting co-ordinates of corner points for physical checkerboard printout

```
1
2 %This loads Model.txt which includes the co-ordinates of the corner ...
      points
3 %on the physical print of the checkerboard
4
5 load Model.txt; %loads file
6
7 %loads all X co-ordinates into a 64x4 matrix by taking all rows from the
8 %odd numbered columns, for Y-co-ordinates does the same but with the ...
      even
9 %numbered columns
10
11 X = Model(:,1:2:end);
12 Y = Model(:,2:2:end);
13
14 %takes the 64x4 matrices generated before and reads down down each ...
      column
15 %to load all co-ordinates into a single column. This is then ...
      transposed so
16 %the co-ordinates are in a single row
17
18 X = X(:)'; %puts x co-ordinates  of corner points all in one row
19 Y = Y(:)'; %puts y co-ordinates of corner points all in one row
```

Listing 1.1.2: Extracting co-ordinates of corner points for image 1 from data1.txt and loading into imagePoints

```
%loads files containing pixel co-ordinates of corner points from the 5
%images of the checkerboard

load data1.txt;
load data2.txt;
load data3.txt;
load data4.txt;
load data5.txt;

%similar to code for Model.txt, this takes the x-coordinates from ...
    the odd
%numbered columns, y-coordinates from the even numbered columns and puts
%them into 64x4 matrices. Then it places all the x-coordinates in ...
    one row
%and all the y-coordinates in a single row.

%Then these corner points are placed into the 5-layered matrix ...
    called image
%points

x = data1(:,1:2:end);
y = data1(:,2:2:end);
x = x(:)';
y = y(:)';
imagePoints(:,:,1) = [x;y]; %extracts corner points of image 1 into ...
    matrix
```

Listing 1.1.3: Populating the phi matrix en route to finding H

```
%Used to populate 5 separate phi matrices to then be used with SVD to
%find the H matrix
%Loads corner points of physical printout from X and Y
%Loads pixel x-coordinates of corner points from images from 1st row of
%imagePoints
%Loads pixel y-coordinates of corner points from images from 2nd row of
%imagePoints

%For loop to generate 5 layers as there will be 5 layers of 512x9 ...
    matrices
%used to find the H matrix for each of the 5 images used

%The inner loop is used to populate each 512x9 matrix to help find ...
    the H
%matrix for each image

phi = zeros(512,9,5);
for N = 1:numberImages
    for i = 1:2:511
    j = (i+1)/2;
    phi(i,:,N) = [0, 0, 0, X(j), Y(j), 1, -imagePoints(2,j,N)* ...
        X(j),-imagePoints(2,j,N)* Y(j),-imagePoints(2,j,N)];
    phi(i+1,:,N)=[ X(j), Y(j),1, 0, 0, 0, -imagePoints(1,j,N)* ...
        X(j),-imagePoints(1,j,N)* Y(j),-imagePoints(1,j,N)];
    end
```

```
22 end
```

Listing 1.1.4: Performing SVD to find the h-values

```
1 H = zeros(9,5);
2 for N = 1:numberImages
3     [U,S,V] = svd(phi(:,:,N));
4     H(:,N) = V(:,end); %populate columsn of H-matrix using last ...
          column of V
5 end
```

Listing 1.1.5: Performing SVD to find the B values

```
1  %Finding the v and B matrices to extract the intrinsic parameters ...
       from H
2  %Equation for v found in section 3.5 of workshop
3  %Extracting the correct h-values to match from each column
4  %Repeated 5 times as there will be 10 rows in the v matrix (5*2) since
5  %there are 5 images
6
7  v = zeros(10,6);
8  for N = 1:numberImages
9      v(2*N-1,:) = [H(1,N)*H(2,N), H(4,N)*H(2,N)+H(1,N)*H(5,N), ...
           H(4,N)*H(5,N), H(7,N)*H(2,N)+H(1,N)*H(8,N),  ...
           H(7,N)*H(5,N)+H(4,N)*H(8,N), H(7,N)*H(8,N)];
10     v(2*N,:) = [[H(1,N)*H(1,N), H(4,N)*H(1,N)+H(1,N)*H(4,N), ...
           H(4,N)*H(4,N), H(7,N)*H(1,N)+H(1,N)*H(7,N),  ...
           H(7,N)*H(4,N)+H(4,N)*H(7,N), H(7,N)*H(7,N)]-[H(2,N)*H(2,N), ...
           H(5,N)*H(2,N)+H(2,N)*H(5,N), H(5,N)*H(5,N), ...
           H(8,N)*H(2,N)+H(2,N)*H(8,N),  H(8,N)*H(5,N)+H(5,N)*H(8,N), ...
           H(8,N)*H(8,N)]];
11 end
12
13 %Right hand side for equation with the transposes of v multiplied by
14 %b values is 0 so we can use SVD to extract the b-values
15 %As before, these will be in the last column of the V matrix as that ...
       column
16 %will have the smallest singular value of S
17
18 [U,S,V] = svd(v);
19 B = V(:,end); %finds the b-matrix which is then used to find ...
       intrinsic params.
20
21 %Using the b-values and the equations on page 14 of the workshop we can
22 %extract the needed b values and apply them to the equations to find our
23 %extrinsic parameters
```

Listing 1.1.6: Finding the intrinsic parameters using the equations from Pg 14 of the workshop

```
1
2 y_0 = ( B(2)*B(4)-B(1)*B(5) )/( B(1)*B(3)-B(2)^2);
3 lambda = B(6)-(B(4)^2+y_0*(B(2)*B(4)-B(1)*B(5)))/B(1);
4 alpha = sqrt(lambda/B(1));
```

```
beta = sqrt(lambda*B(1)/(B(1)*B(3)-B(2)^2));
skewness = -B(2)*alpha^2*beta/lambda;
x_0 = skewness*y_0/alpha - B(4)*alpha^2/lambda;

%Used to display the 6 instrinsic parameters of our camera
disp('alphax, skew, alphay, x0, y0, k1, k2');

disp(alpha)
disp(skewness)
disp(beta)
disp(x_0)
disp(y_0)

%intrinsic matrix K formed
K = [alpha, skewness, x_0; 0, beta, y_0; 0, 0, 1];
```

Listing 1.1.7: Using K and equations given to find the extrinsic parameters

```
%Loop to find the extrinsic parameters for each image using the ...
    equations
%in Section 3.6 of the workshop.
%5 layers as each image will produce different extrinsic parameters
%Each layer contains a 3x4 matrix with the each column containing a
%separate extrinsic parameter

R = zeros(3,4,5);
for N = 1:numberImages
    sigma = 1/ norm(inv(K)*[H(1,N);H(4,N);H(7,N)]);
    r1 = sigma* inv(K)*[H(1,N);H(4,N);H(7,N)];
    r2 = sigma* inv(K)*[H(2,N);H(5,N);H(8,N)];
    r3 = cross(r1,r2);
    Cpworg = sigma* inv(K) * [H(3,N);H(6,N);H(9,N)];
    R(:,:,N) = [r1, r2, r3, Cpworg];
end

disp(R)
```

1.2 Task 2 - Camera Calibration Experiments

In Task 2, we were asked to calibrate our own camera meaning we would be performing the same work as in Task 1 but the images to find the camera parameters were taken from our webcam instead of being provided from the Microsoft Research data set. In addition, this meant that the corner points were not given and ready to be loaded into MATLAB and hence a corner detection algorithm had to be used. After finding these, we were then required to reuse some of our code in Task 1 to find the camera parameters and compare it with those calculated by MATLAB's built-in tool. The calculated parameters were then saved into matrices for later use and to then replace the parameters in our .mlapp file used to operate the virtual robot.

As with Task 1, let us take a look at the steps performed for Task 2 chronologically:

Step 1 – Download the Task 2 Data folder and access and print the checkerboard image:

The checkerboard image is printed and stuck onto a hard surface. We then run some MATLAB code that is used to take images via our webcams of us holding the checkerboard at 6 different positions and orientations. It is important that we hold the checkerboard steadily so that the corner detection algorithms can work and produce more accurate results. The MATLAB code used to access our webcams and take the 6 photos of the checkerboard in different orientations to be used for calibration is shown in Listing 1.2.1 below.

Step 2 – Use the code provided in the Workshop sheet to find the corner points and camera parameters

Next, the snippet "Code 2" provided in the workshop sheet is used to read the images, detect the corner points of the checkerboards using the detectCheckerboardPoints() and then generate them in the world co-ordinate frame to create physical corners based on the image using generateCheckerboardPoints(). After this, calibration takes place using the estimateCameraParameters() function which is built in to calculate the camera parameters and hence the camera calibration would have been completed by this built-in MATLAB tool and is saved to the workspace as cameraParams to later be compared. The code "Code 2" used to describe all the above in Step 2 can be found in line 5 to 33 in the file 'Vision_Task2Calibration.m' which is attached with this report. This code also showed the accuracy of the calibration and a visualisation of the camera checkerboard placements as seen in Figure 1.1 below.

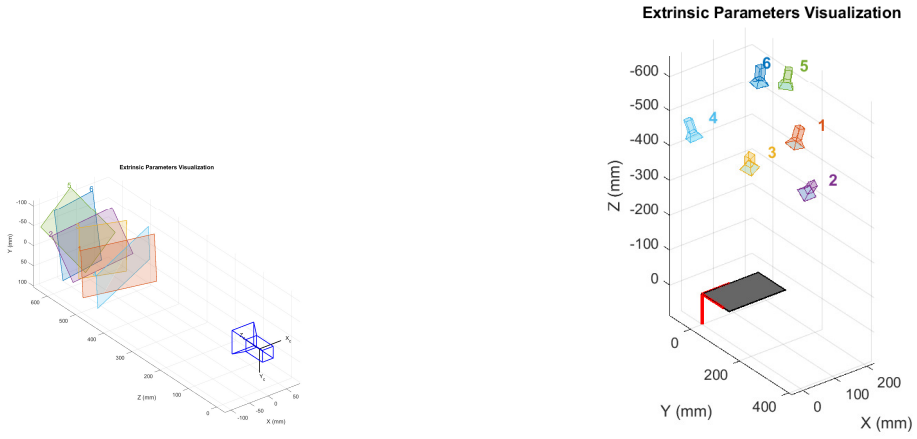

Figure 1.1: The left image shows a visualisation of the camera checkboard placements and the right figure shows a visualisation of where the cameras would be located

Step 3 – Re-use the code developed in Task 1 to calculate the camera parameters ourselves:

After the camera was calibrated using the built in tool, we use the same code used in 'Vision_Task1.m' that was used to calibrate the camera using the checkerboard data in Task 1. The specifics of the procedure was described in plenty of detail in Section 1.1 and so in this section we will only take a look at how it was adapted to match the matrix sizes of the new images captured and the number of images captured.

Similar to the initial step in Task 1, we use the worldPoints variable which contains the generated checkerboard to extract the co-ordinates of the physical corner points in the

generated physical checkerboard. The code for this is seen in Listing 1.2.2 and is similar to that in Listing 1.1.1

As with in Task 1, we want to first use the corner points which were calculated to find the H matrix. In this case, the corner points were saved to imagePoints which had dimension of 256 by 2 by 5. We first re-arrange the dimensions using the permute function such that it becomes a matrix which is seen to be 2 by 54 by 6 (this is done so that it fits in with the calculations we had done previously in Task 1). These dimensions can be deciphered to find the number of corner points and images. The 2 denotes 2 rows (one containing x values and one with y values) whilst 54 denotes the number of corner points and 6 denotes the number of images and hence there are 6 layers to the imagePoints matrix.

Similar to Step 4 in Task 1 where we populated the phi matrix to then find the h-values, we use the code seen in Listing 1.2.3. The only things changed are the number of times the outer for loop goes as we now have 6 images not 5. Furthermore, the dimensions of phi have changed as we are now only analysing 54 corner points so phi only has 108 rows instead of the 512 rows it had when we had 256 corner points to use.

Similar to Step 5 in Task 1, we now use SVD to find the h-values. The exact same maths and function is carried out but H now has 6 columns instead of five since we have 6 images instead of 5 and hence will have one more set of H values. This is because the H-matrix is generated for each image. This is is seen in Listing 1.2.4.

Similar to Step 6 in Task 1, we are extracting the intrinsic parameters from the h values. To do this we first use the same for loop we used to calculate and populate the v matrix using the h-values as shown below. This is identical to what we used in task 1 but v is now as 12 by 6 matrix instead of a 10 by 6 since we now have one more image for there will be 2 more rows in the v matrix. However, the SVD part and the equations to calculate the intrinsic parameters (Listing 1.2.6) are still identical to what they were in task 1.

Similar to Step 7 in Task 1, we extract the extrinsic parameters now that we have the K matrix containing the intrinsic parameters. This is the exact same code used and the same equations used to calculate the parameters but there is one extra column in the R matrix and one permutation of the for loop to fill that extra column. This extra column exists to calculate these parameters for the sixth image and display them in an extra 3 by 4 layer of the matrix. This is because extrinsic parameters can only be calculated for one image at a time.

Step 4 – Compare our parameters with those calculated by the built-in tool:

By accessing the cameraParams in the workspace, we can explore the extrinsic and intrinsic parameters calculated by the built-in tool and compare it to the values calculated by our code.

In terms of the intrinsic parameters, I have summarised my values vs the tool's values below to create a better visual for comparison:

Parameter	Built in tool	Our code
α	867.2467	862.2087
β	867.8549	861.5087
x_0	354.0663	359.8697
y_0	231.9406	231.2575
Skewness λ	0.0000	-1.8128

Table 1.1: Table to compare the parameters obtained by the tool and our code

In this table we can see that the values are all extremely close together apart from the Estimated Skew. All the variables have a percentage difference of less than 2% meaning the results are very similar and are not different enough to discredit our algorithm or method. There is a larger difference in Estimated Skew but it is seen to be 0 in the built in tool meaning it is perhaps not taken into account as it was mentioned previously in the workshop sheet that the skewness parameter is common to add but is not a must meaning that the computer might be assuming more idealistic situations.

In terms of the extrinsic parameters, rather than creating a table I just compared them by due to there being a lot more than with the intrinsic parameters. For the rotation matrices as well as the translation vectors, all the numbers are of the same magnitude, however, there seems to be a switch in sign for some of the parameters (very rare but has been seen). This could be due to differences in the frame in which they were calculated or the definition of a certain direction to the built-in tool compared to us. It might be due to the built in tool perhaps using a different frame or direction convention because as long as the magnitudes are right then it seems that is what is most important and this is probably due to some change in frame directions.

These are the matrices for the first and second image according to our code:

$$\begin{bmatrix} -0.9802 & 0.0409 & 0.1943 & 114.8492 \\ -0.0147 & -0.9920 & 0.1286 & 59.5316 \\ 0.1977 & 0.1230 & 0.9729 & -482.7473 \end{bmatrix} \begin{bmatrix} 0.9953 & 0.0732 & -0.0675 & -99.4270 \\ -0.0153 & 0.8043 & 0.6017 & -33.8837 \\ 0.0953 & -0.5975 & 0.8017 & 606.4172 \end{bmatrix} \quad (1.26)$$

These are the matrices for the first and second image according to the tool:

$$\begin{bmatrix} 0.9784 & 0.0150 & -0.2061 & -111.5082 \\ -0.0429 & 0.9904 & -0.1312 & -59.8583 \\ 0.2021 & 0.1372 & 0.9697 & 487.2759 \end{bmatrix} \begin{bmatrix} 0.9956 & -0.0154 & 0.0926 & -95.2468795818703 \\ 0.0674 & 0.8034 & -0.5916 & -34.2841802681793 \\ -0.0653 & 0.5953 & 0.8009 & 610.123447660521 \end{bmatrix} \quad (1.27)$$

Step 5 – Save any files that will be required for Task 3 and made some edits to our virtual robot .mlapp file

Finally, we saved the cameraParams variable from the workspace to a .mat file so that it can be loaded up later for when we are completing Task 3 as these will also determine some of the values of the camera intrinsics and CameraWorld variables to then be edited in our virtual robot's .mlapp file.

The procedure used for the code is explained in a great amount of detail in Task 1 and so for this section, we only went briefly over the main differences in the code which were mostly adaptations to the variables and loops to accommodate for the sixth image or difference in number of corner points.

1.2.1 Matlab Code

Listing 1.2.1: Operating webcam via MATLAB to take 6 photos of the checkerboard

```
clear all
close all
clc

numImages = 6;
camList = webcamlist;
cam = webcam(1);
preview(cam);

for idx = 1:numImages

    pause(10.0);
    img(:,:,:,idx) = snapshot(cam);
    figure,image(img(:,:,:,idx));

    fname = sprintf('Image%d.png',idx);
    imwrite(img(:,:,:,idx),fname);
end

clear cam
```

Listing 1.2.2: Extracting the co-ordinates of the pixel corner points of the generated image and corner of the images taken via the webcam

```
numberImages = numImages;

worldPoints = worldPoints';
X = worldPoints(1,:);
Y = worldPoints(2,:);

imagePoints = permute(imagePoints,[2,1,3]);
```

Listing 1.2.3: Populating phi matrix with 54 corner point to find H

```
phi = zeros(108,9,6);
for N = 1:numberImages
    for i = 1:2:107
    j = (i+1)/2;
    phi(i,:,N) = [0, 0, 0, X(j), Y(j), 1, -imagePoints(2,j,N)* ...
        X(j),-imagePoints(2,j,N)* Y(j),-imagePoints(2,j,N)];
    phi(i+1,:,N)=[ X(j), Y(j),1, 0, 0, 0, -imagePoints(1,j,N)* ...
        X(j),-imagePoints(1,j,N)* Y(j),-imagePoints(1,j,N)];
    end
end
```

Listing 1.2.4: Performing SVD to find the h-values

```
H = zeros(9,6);
for N = 1:numberImages
```

```
3       [U,S,V] = svd(phi(:,:,N));
4       H(:,N) = V(:,end); %populate column of H-matrix using last ...
            column of V
5   end
```

Listing 1.2.5: Performing SVD to find the B values

```
1   v = zeros(12,6);
2   for N = 1:numberImages
3       v(2*N-1,:) = [H(1,N)*H(2,N), H(4,N)*H(2,N)+H(1,N)*H(5,N), ...
            H(4,N)*H(5,N), H(7,N)*H(2,N)+H(1,N)*H(8,N),  ...
            H(7,N)*H(5,N)+H(4,N)*H(8,N), H(7,N)*H(8,N)];
4       v(2*N,:) = [[H(1,N)*H(1,N), H(4,N)*H(1,N)+H(1,N)*H(4,N), ...
            H(4,N)*H(4,N), H(7,N)*H(1,N)+H(1,N)*H(7,N),  ...
            H(7,N)*H(4,N)+H(4,N)*H(7,N), H(7,N)*H(7,N)]-[H(2,N)*H(2,N), ...
            H(5,N)*H(2,N)+H(2,N)*H(5,N), H(5,N)*H(5,N), ...
            H(8,N)*H(2,N)+H(2,N)*H(8,N),  H(8,N)*H(5,N)+H(5,N)*H(8,N), ...
            H(8,N)*H(8,N)]];
5   end
6
7   [U,S,V] = svd(v);
8   B = V(:,end);
```

Listing 1.2.6: Finding the intrinsic parameters using the equations from Pg 14 of the workshop

```
1
2   y_0 = ( B(2)*B(4)-B(1)*B(5) )/( B(1)*B(3)-B(2)^2);
3   lambda = B(6)-(B(4)^2+y_0*(B(2)*B(4)-B(1)*B(5)))/B(1);
4   alpha = sqrt(lambda/B(1));
5   beta = sqrt(lambda*B(1)/(B(1)*B(3)-B(2)^2));
6   skewness = -B(2)*alpha^2*beta/lambda;
7   x_0 = skewness*y_0/alpha - B(4)*alpha^2/lambda;
8   disp('alphax, skew, alphay, x0, y0, k1, k2');
9
10  disp(alpha)
11  disp(skewness)
12  disp(beta)
13  disp(x_0)
14  disp(y_0)
15
16  %intrinsic matrix
17  K = [alpha, skewness, x_0; 0, beta, y_0; 0, 0, 1];
```

Listing 1.2.7: Using K and equations given to find the extrinsic parameters

```
1   %Loop to find the extrinsic parameters for each image using the ...
        equations
2   %in Section 3.6 of the workshop.
3   %5 layers as each image will produce different extrinsic parameters
4   %Each layer contains a 3x4 matrix with the each column containing a
5   %separate extrinsic parameter
6
7   R = zeros(3,4,5);
```

```
for N = 1:numberImages
    sigma = 1/ norm(inv(K)*[H(1,N);H(4,N);H(7,N)]);
    r1 = sigma* inv(K)*[H(1,N);H(4,N);H(7,N)];
    r2 = sigma* inv(K)*[H(2,N);H(5,N);H(8,N)];
    r3 = cross(r1,r2);
    Cpworg = sigma* inv(K) * [H(3,N);H(6,N);H(9,N)];
    R(:,:,N) = [r1, r2, r3, Cpworg];
end

disp(R)
```

1.2.2 Matlab Code

1.3 Task 3 - Robot Calibration Experiments

The parameters found in 1.2 were updated into the .mlapp file so we could proceed with the calibration of the robot. To calibrate the robot we firstly began by jogging the calibration point to the location of several corners of the checkerboard and recorded the cartesian position of these points with respect to the robot frame. For each of these points we also record the position of the corner point with respect to the world frame. Using these two coordinates we are able to relate the two frames using:

$$\begin{bmatrix} X_{ri} \\ Y_{ri} \\ Z_{ri} \\ 1 \end{bmatrix} = \begin{bmatrix} \rho_{11} & \rho_{12} & \rho_{13} & t_x \\ \rho_{21} & \rho_{22} & \rho_{23} & t_y \\ \rho_{31} & \rho_{32} & \rho_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ 0 \\ 1 \end{bmatrix} \tag{1.28}$$

where the LHS parameters are the coordinates with respect to the robot frame and the RHS parameters are the ones w.r.t. the world frame. our objective is to find the transformation matrix relating these two frames which can be accomplished by solving the following set of equations:

$$X_{ri} = \rho_{11}X_i + \rho_{12}Y_i + t_x \tag{1.29}$$
$$Y_{ri} = \rho_{21}X_i + \rho_{22}Y_i + t_y \tag{1.30}$$
$$Z_{ri} = \rho_{31}X_i + \rho_{32}Y_i + t_z \tag{1.31}$$

We can find the parameters my minimizing the square errors as:

$$E_x^2 = \sum_{i=1}^{n} (X_{ri} - (\rho_{11}X_i + \rho_{12}Y_i + t_x))^2 \tag{1.32}$$
$$E_y^2 = \sum_{i=1}^{n} (Y_{ri} - (\rho_{21}X_i + \rho_{22}Y_i + t_y))^2 \tag{1.33}$$
$$E_z^2 = \sum_{i=1}^{n} (Z_{ri} - (\rho_{31}X_i + \rho_{32}Y_i + t_z))^2 \tag{1.34}$$

The full derivation is proposed by Cashbaugh et al. in [Cashbaugh2018] but we will walk through some steps for finding the parameters in Eqn.1.32. The minimization can be

done by finding the derivative w.r.t. the parameters desired and setting to zero as:

$$\frac{\partial E_x^2}{\partial \rho_{11}} = -2\sum_{i=1}^{n}(X_{ri} - (\rho_{11}X_i + \rho_{12}Y_i + t_x))X_i \qquad = 0 \tag{1.35}$$

$$\frac{\partial E_x^2}{\partial \rho_{12}} = -2\sum_{i=1}^{n}(X_{ri} - (\rho_{11}X_i + \rho_{12}Y_i + t_x))Y_i \qquad = 0 \tag{1.36}$$

$$\frac{\partial E_x^2}{\partial t_x} = -2\sum_{i=1}^{n}(X_{ri} - (\rho_{11}X_i + \rho_{12}Y_i + t_x)) \qquad = 0 \tag{1.37}$$

which can be rewritten as:

$$\sum \rho_{11}X_iX_i + \sum \rho_{12}Y_iX_i + \sum t_xX_i = \sum X_{ri}X_i \tag{1.38}$$

$$\sum \rho_{11}X_iY_i + \sum \rho_{12}Y_iY_i + \sum t_xY_i = \sum X_{ri}Y_i \tag{1.39}$$

$$\sum \rho_{11}X_i + \sum \rho_{12}Y_i + \sum t_x = \sum X_{ri} \tag{1.40}$$

This in turn can be put into matrix form and flipped to find the unknowns as:

$$\begin{bmatrix}\rho_{11}\\ \rho_{12}\\ t_x\end{bmatrix} = \begin{bmatrix}\sum X_iX_i & \sum Y_iX_i & \sum X_i\\ \sum X_iY_i & \sum Y_iY_i & \sum Y_i\\ \sum X_i & \sum Y_i & n\end{bmatrix}^{-1}\begin{bmatrix}X_{ri}X_i\\ X_{ri}Y_i\\ X_{ri}\end{bmatrix} \tag{1.41}$$

Similar equations can be found for the remaining parameters and the final parameters can be calculated using:

$$\begin{bmatrix}\rho_{13}\\ \rho_{23}\\ \rho_{33}\end{bmatrix} = \begin{bmatrix}\rho_{11}\\ \rho_{21}\\ \rho_{31}\end{bmatrix} \times \begin{bmatrix}\rho_{12}\\ \rho_{22}\\ \rho_{32}\end{bmatrix} \tag{1.42}$$

as the minimization of E_y^2 and E_z^2 yields the missing parameters. The implementation of these equations in the code can be seen in Listing 1.3.1 where the subscript of the variable indicates the first subscript m of each parameter ρ_{mn} and, by indexing the variables $\text{rho}_m \in \mathbb{R}^3$ we obtain the individual parameters $\rho_{mn} \leftarrow \text{rho}_m[n]$. The remaining parameters are found using Eqn.1.41. The way this is done in the code is shown in Listing.1.3.2. The full transformation matrix can now be assembled as seen in Listing.1.3.3

$$T_{Robotworld} = \begin{bmatrix}\rho_{11} & \rho_{12} & \rho_{13} & t_x\\ \rho_{21} & \rho_{22} & \rho_{23} & t_y\\ \rho_{31} & \rho_{32} & \rho_{33} & t_z\\ 0 & 0 & 0 & 1\end{bmatrix} \tag{1.43}$$

1.3.1 Matlab Code

Listing 1.3.1: ρ parameter matrix calculation

```
1 rho1 = Matrix * [sum(Xrbt.*Xwld); sum(Xrbt.* Ywld); sum(Xrbt)];
2 rho2 = Matrix * [sum(Yrbt.*Xwld); sum(Yrbt.* Ywld); sum(Yrbt)];
3 rho3 = Matrix * [sum(Zrbt.*Xwld); sum(Zrbt.* Ywld); sum(Zrbt)];
```

Listing 1.3.2: ρ parameter isolaton

```
rho4 = cross([rho11;rho21;rho31], [rho12; rho22; rho32]);
rho13 = rho4(1);
rho23 = rho4(2);
rho33 = rho4(3);
```

Listing 1.3.3: Transformation matrix assembly

```
T_RobotWorld = [rho11, rho12, rho13, tx; rho21, rho22, rho23, ty; ...
    rho31, rho32, rho33, tz; 0, 0, 0, 1];

Rrob = T_RobotWorld (1:3,1:3);
trob = T_RobotWorld (1:3,4);
```

1.4 Task 4 - Image Processing

Once we have obtained the image of the objects we can find the locations of them to then be able to move the robot to those positions and pick them up. Firstly we isolate the two objects by making use of their color. We create two new maps of pixels using the colors of the objects and subsequently use them as masks to select out the gray-scale versions of the cube and cylinder (Listing. 1.4.1.

The same thing is repeated for the cube and both are subsequently transformed into binary (1 or 0 pixel values) for the edge and center detection.

The center of each object is found by utilizing the moment of the shape, essentially the 'center of mass'. To find the central moment of the object we use the following formula:

$$M_{pq} = \sum_{(x,y \in \text{Image})} x^p y -^q I(x,y) \tag{1.44}$$

We also note that M_{00} is simply the area of the shape for a binary image. To find the centroid of the shape we use:

$$X_c = \frac{M_{10}}{M_{00}}, \text{ and } Y_c = \frac{M_{01}}{M_{00}} \tag{1.45}$$

Finally we have the location of the center the objects and our end effector is able to move to the object. Next we must tackle the issue of object recognition and orientation.

To recognize the object we use the trick of counting the corners which, in our case, would result in a value of 0 if the object is the cylinder and a value > 0 if the object is the cube. Note we do not specify the number of corners for the cube as we are limited by resolution and by the cross present in the shape throwing off the calculation. To count the corners we must first find the edges using some sort of convolution mask filter, in our case the Laplacian (omidirectional) given by:

$$M_{x,y} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{1.46}$$

The corners are then found by setting a threshold value and verifying if a given point of the edge-detected image is above that threshold value. A small tweak is done to avoid two corners being detected too close to each other which could be a result of a misinterpretation of the edge map.

Another way the shape identification could be completed is by calculating the *Circularity* of each shape; in the case of our shapes this would be of value 1 for the cylinder unsurprisingly, and would be $\frac{\pi}{4}$ for the cube as their 2D projections captured by the camera are of a circle and a square respectively.

To calculate the circularity we must first find the perimeter of each shape which can be done using the code in Listing 1.4.2 which detects the boundary of the object and adds the pixels which make up the boundary of the object. With the perimeter

$$C = \frac{4\pi M_{00}}{p^2} \tag{1.47}$$

Recall that M_{00} is the area of the object hence the circularity simply represents the ratio between area and perimeter squared of a shape which is where the values for each shape can be calculated. Unfortunately when dealing with discrete values for area and perimeter caused by the discrete nature of pixel-wise image processing we are increasingly limited by the resolution of the images captured as they increase the granularity of our measurements of area and perimeter. Because of this the values we can calculate from out images are not representative enough to determine the shape of the object for either the cylinder or the cube which is why we think corner counting suits this specific task better. With higher resolution imaging, circularity would be a more robust method for shape recognition.

The last piece of information is the orientation of the object which can be found using the concept of moment once again, and finding both axis and the angle of an ellipse which is inscribed by the object. To accomplish this we use:

$$\mu_{pq} = \sum_{(x,y \in \text{Image})} (x - X_c)^p (y - Y_c) -^q I(x,y) \tag{1.48}$$

Where X_c, Y_c are the values found in equation 1.45. Next we calculate the inertia matrix:

$$\begin{aligned}
\mu_{00} &= M_{00} \\
\mu_{01} &= 0 \\
\mu_{10} &= 0 \\
\mu_{11} &= M_{11} - X_c M_{01} = M_{11} - Y_c M_{10} \\
\mu_{20} &= M_{20} - X_c M_{10} \\
\mu_{02} &= M_{02} - Y_c M_{01} \\
J &= \begin{bmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{bmatrix}
\end{aligned}$$

With the eigenvectors and eigenvalues of J we can get the axes of the ellipse and the

angle as:

$$V = [V_x\ V_y]^\top$$
$$a = 2\sqrt{\frac{\lambda_1}{M_{00}}}$$
$$b = 2\sqrt{\frac{\lambda_2}{M_{00}}}$$
$$\theta = \arctan\left(\frac{V_y}{V_x}\right)$$

Where λ_i are the eigenvalues of J, $\lambda_1 > \lambda_2$ and V is the eigenvector corresponding to λ_1. With the axes of the inscribing ellipse we can determine the orientation of the major and minor axes of the object and also the inclination.

The joint angles for the robot to reach the specified point are also calculated taking the resulting cartesian coordinates of the object and using the inverse kinematics function used in the .mlapp file. This way we get the resulting joint angles as output which can be input into the fields of the robot to move the tool to the object. The joint angles found using the machine-vision generated coordinates was incredibly similar to the values we calculated using vector algebra reinforcing the validity of the methods for camera calibration, robot calibration and image processing used throughout the previous tasks.

1.4.1 Matlab Code

Listing 1.4.1: Isolate object by color

```
for i = 1:m
    for j = 1:n
        if (IRed(i,j) > 250) && (IGreen(i,j) > 250) && (IBlue(i,j) < 50)
            I_cyl(i,j) = 255;
        else
            I_cyl(i,j) = 0;
        end
    end
end
```

Listing 1.4.2: Calculating the perimeter of a shape

```
%{
Move from left to right, row by row.
If current pixel is white
   If left pixel is black and right pixel is white --> Take as boundary
   Else if left pixel is white and right pixel is black --> Boundary
   If top pixel is black and bottom pixel is white --> Take as boundary
   Else if top pixel is white and bottom pixel is black --> Boundary
Add the boundaries up

%}
Perimeter = 0;
for i = 2:m-1 % rows from top to bottom
```

```
13      for j = 2:n-1 % columns from left to right
14          if Ibw_cube(i,j) == 1
15              if (Ibw_cube(i,j-1) == 0) && (Ibw_cube(i,j+1) == 1)
16                  Perimeter = Perimeter + 1;
17              elseif (Ibw_cube(i,j-1) == 1) && (Ibw_cube(i,j+1) == 0)
18                  Perimeter = Perimeter + 1;
19              elseif (Ibw_cube(i-1,j) == 0) && (Ibw_cube(i+1,j) == 1)
20                  Perimeter = Perimeter + 1;
21              elseif (Ibw_cube(i-1,j) == 1) && (Ibw_cube(i+1,j) == 0)
22                  Perimeter = Perimeter + 1;
23              end
24          end
25
26      end
27  end
```

2 Trajectory Workshop

In the trajectory planning workshop, we were tasked with planning out a smooth trajectory for the robot to follow rather than approach the target instantly. This was carried out and then integrated into a pick-and-place demonstration which involved the robot moving towards a point of the object, then towards the object. We also had to ensure the robot was able to grip the object and move it around as the robotic arm was jogged. The process behind these activities is highlighted in the subsections below complete with snippets of code used.

2.1 Trajectory Planning for Vision-Guided Pick and Place

Due to time constrains during the first term, we had to initially perform trajectory planning based on a position calculated using vector addition using co-ordinates found in our .mlapp file. This should have ideally been done using the values of the position calculated by the image processing algorithm in Task 4 of the vision workshop (Section 1.4). Regardless, the techniques are essentially the same as they both utilise inverse kinematics and both will be highlighted below. The only difference is that with vector addition we had calculated an exact position ourselves, whilst with image processing the position is less accurate. In retrospect, it seems that having to do it both ways was useful as initially doing the vector addition calculations allowed us to use it for comparison to verify our results in Task 4.

2.1.1 Calculating the object position ourselves

To calculate the position of the objects ourselves we had to do some vector addition using the parameters given which described the position of the robot with respect to the world as well as the positions of the objects with respect to world. Through vector addition, we were able to find the positions of these objects with respect to the robot frame. However, it was not just simple vector addition as the world frame was rotated with respect to the robot frame.

The robot frame was rotated 180 degrees about the x-axis compared to the robot frame such that we had to include a factor of [-1,1,1] in front of PWorldObject variables. This

is because the x-axis is the only axis which remains facing the same direction as the other and so going backwards would have to represented by a negative number. However, since the y and z axis are flipped already for the frame, there is no need to include a factor of -1 as that would counteract the change.

These calculations are seen in Listing 2.3.1 (Lines 27 and 28) and gave use the final results that the Cylinder was at (0, 150, -20) and the Cube was at (-150, 100, -20).

2.1.2 Automating the calculation of the position

In Task 4 of the vision workshop (Section 1.4), we used image processing techniques to find the position of both of the objects and manage to calculate its XYZ co-ordinates with respect to the robot frame. We also calculate the joint angles in the same '.m' file. However, rather than having to manually enter the numbers found by the image processing. We create a variable called filename, and we use the save function to save one or more variables from the workspace to a .mat file which will be later accessible by the mlapp file.

In the mlapp file we use the load function, . However, to ensure that the robot still reaches the desired target, we include an if statement using the isfile function which checks if a file exists. If the file exists, then it loads the .mat file and we can call upon the coordinates saved in 'Vision_Task4.m'. However, if the desired .mat file doesn't exist (meaning the vision and image processing program hasn't been run), we call the regular ApproachCylinder and ApproachCube functions which will move the robot to the points calculated by the vector addition. This is seen in Listing 2.1.1 below.

Listing 2.1.1: Approaching the object using position values calculated in another file and loaded to the mlapp file

```
1   % Button pushed function: VisionGuidedApproachCylinderButton
2         function VisionGuidedApproachCylinderButtonPushed(app, event)
3
4             if isfile('ImageProcessingObjectPositions.mat')
5              load ImageProcessingObjectPositions.mat;
6             else
7               ApproachCylinderButtonPushed(app,event);
8             end
9
10            app.XEditField.Value = robotCylinder(1);
11            app.YEditField.Value = robotCylinder(2);
12            app.ZEditField.Value = robotCylinder(3) + 40;
13            InverseTrajectoryButtonPushed(app,event);
14            pause(2)
15            app.ZEditField.Value = robotCylinder(3);
16            InverseTrajectoryButtonPushed(app,event);
17
18        end
19
20        % Button pushed function: VisionGuidedApproachCubeButton
21        function VisionGuidedApproachCubeButtonPushed(app, event)
22            if isfile('ImageProcessingObjectPositions.mat')
23            load ImageProcessingObjectPositions.mat;
24            else
25              ApproachCubeButtonPushed(app,event);
```

```
26              end
27
28              app.XEditField.Value = robotCube(1);
29              app.YEditField.Value = robotCube(2);
30              app.ZEditField.Value = robotCube(3) + 40;
31              InverseTrajectoryButtonPushed(app,event);
32              pause(2)
33              app.ZEditField.Value = robotCube(3);
34              InverseTrajectoryButtonPushed(app,event);
35          end
```

2.2 Plotting the trajectory

Regardless of the method used to find the coordinates of the object and enter it into the mlapp file, we must generate a smooth trajectory for the robot to follow. A trajectory is essentially a time profile of the position, velocity and acceleration of a movement. The numerical values of these parameters should be extractable at any point to act as a reference for the next movement in the trajectory. In trajectory planning, you use the current position, desired position, time to reach desired position and the general shape of the path to control the trajectory.

Our task was to use the cubic polynomial technique to generate a cubic path for the robot to reach a point above the target in 2 seconds. The robot would then wait there for two seconds and then follow another cubic path down to the target and stay still there. The cubic polynomial is advantageous as it starts and ends at a velocity (however, unfortunately acceleration and deceleration at start and end are not zero so motion might be a bit jerky). To ensure the start and end velocities are 0 we can say:

$$|u(t)| = \begin{cases} a_0 + a_1 t + a_2 t^2 + a_3 t^3 & \text{if } t < t_f \\ u_f & \text{if } t \geq t_f \end{cases} \tag{2.1}$$

We have 4 constraints to satisfy which are that the position at zero is the initial position (u_0) whilst the position at t_f is the final destination u_f. The velocity at both positions is zero so the differential of both positions is zero:

$$\begin{aligned} u(0) &= u_0 \\ u(t_f) &= u_f \\ \dot{u}(0) &= 0 \\ \dot{u}(t_f) &= 0 \end{aligned} \tag{2.2}$$

We can calculate the coefficients by solving the equations:

$$u(0) = a_0 + a_1 0 + a_2 0^2 + a_3 0^3 = a_0 \qquad = u_0 \tag{2.3}$$
$$u(t_f) = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 \qquad = u_f \tag{2.4}$$
$$\dot{u}(0) = a_1 + 2a_2 0 + 3a_3 0^2 = a_1 \qquad = 0 \tag{2.5}$$
$$\dot{u}(t_f) = a_1 + 2a_2 t_f + 3a_3 t_f^2 \qquad = 0 \tag{2.6}$$

The solutions to these equations are found to be:

$$a_0 = u_0 \tag{2.7}$$
$$a_1 = 0 \tag{2.8}$$
$$a_2 = \frac{3}{t_f^2}(u_f - u_0) \tag{2.9}$$
$$a_3 = -\frac{2}{t_f^3}(u_f - u_0) \tag{2.10}$$

Finally we implement the maths into code beginning from forward trajectory calculation; here we are given the joint angles as inputs. The final angles are taken from the numerical edit fields for the angles hence the slider values remain at the initial positions as the button is pushed. This allows us to take the initial conditions from the slider values before updating them and scrapping the information on initial position. Once the initial and final positions for the robot we set the desired length of the trajectory in seconds and subsequently find the angle values for each time update using the equations described above. When all of the values for the three angles are calculated and cahced they are looped through and, after each new point is updated in the theta edit fields, the forward button pushed is called moving the robot joints to the new locations edited in the angle fields. This is shown in Listing **??**.

Next we must consider the inverse trajectory calculation where the cartesian coordinates of the desired final position are given. The initial positions are taken from the theta edit fields as they have not been changed but we must convert the cartesian coordinated from the edit fields into angles which will be updated into the theta edit fields for the forward trajectory to use them. The way this is done is quite simple as it makes use of the inverse kinematics code we had completed before and which is shown in Listing. 2.2.2 Following the conversion we have ibtained initial and final joint angles and the process is not identical to the code shown in Listing **??** for the forward trajectory

2.2.1 Matlab code

Listing 2.2.1: Forward trajectory calculations

```
1
2               %% TRAJECTORY PLANNING
3
4              % Boundary Conditions
5              u0_1 = app.Theta1Slider.Value;
6              u0_2 = app.Theta2Slider.Value;
7              u0_3 = app.Theta3Slider.Value;
8
9              uf_1 = app.Theta1EditField.Value;
10             uf_2 = app.Theta2EditField.Value;
11             uf_3 = app.Theta3EditField.Value;
12
13             tfirst = 2;
14             trest = 2;
15             tfinal = 2;
16
17             %Create Time Array
18             time = 0.1;
```

```
            t = 0:time:tfirst;
            t = t';

            %Cubic Polynomial for Theta 1

            a0 = [u0_1, u0_2, u0_3];
            a1 = [0, 0, 0];
            a2 = [(3/tfirst^2)*(uf_1-u0_1), ...
                (3/tfirst^2)*(uf_2-u0_2), (3/tfirst^2)*(uf_3-u0_3)];
            a3 = [(-2/tfirst^3)*(uf_1-u0_1), ...
                (-2/tfirst^3)*(uf_2-u0_2), (-2/tfirst^3)*(uf_3-u0_3)];

            uCubic = ...
                [a0(1)*ones(length(t),1)+a1(1)*t+a2(1)*t.^2+a3(1)*t.^3,
                 a0(2)*ones(length(t),1)+a1(2)*t+a2(2)*t.^2+a3(2)*t.^3,
                 a0(3)*ones(length(t),1)+a1(3)*t+a2(3)*t.^2+a3(3)*t.^3];

            %%%
            i = 0;
            for i = 1:numel(t)
                app.Theta1EditField.Value = uCubic(i,1);
                app.Theta1Slider.Value = uCubic(i,1);

                app.Theta2EditField.Value = uCubic(i,2);
                app.Theta2Slider.Value = uCubic(i,2);

                app.Theta3EditField.Value = uCubic(i,3);
                app.Theta3Slider.Value = uCubic(i,3);
                ForwardButtonPushed(app,event)
                pause(time)
            end
```

Listing 2.2.2: Inverse trajectory conversion to joint angles

```
            x = app.XEditField.Value;
            y = app.YEditField.Value;
            z = app.ZEditField.Value;

            global L2;
            flag = 0;

            if sqrt(x^2+y^2+z^2)<= 2*L2
                flag = 1;
            end

            if flag == 1
                % For theta 1
                Theta1_final = atan2(y,x)*180/pi;


                % For theta 3
                C3 = (x^2+y^2+z^2)/(2*L2^2)-1;
                S3 = sqrt(1-C3^2);
                theta3_positive = atan2 (S3,C3) *180 /pi;
                theta3_negative = atan2 (-S3,C3) *180 /pi;

                if (theta3_negative < theta3_positive && ...
```

```
                    app.Theta3EditField.Value <= 0)
24                      Theta3_final = theta3_negative;
25                      S3 = -S3;
26                  else
27                      Theta3_final = theta3_positive;
28                  end
29                  % For theta 2
30
31                  k1 = L2*(1+C3);
32                  k2 = L2*S3;
33
34                  r = sqrt(k1^2+k2^2);
35                  beta = atan2(k2,k1);
36
37                  Theta2_final_pos = (atan2(z/r,sqrt(x^2+y^2)/r)-beta) ...
                        *180/pi;
38                  Theta2_final_neg = ...
                        (atan2(z/r,-sqrt(x^2+y^2)/r)-beta) *180/pi;
39
40              else
41              f = msgbox("The target cannot be reached","Error","error");
42              end
43
44              u0_1 = app.Theta1EditField.Value;
45              u0_2 = app.Theta2EditField.Value;
46              u0_3 = app.Theta3EditField.Value;
47
48              uf_1 = Theta1_final;
49              uf_2 = Theta2_final_pos;
50              uf_3 = Theta3_final;
```

2.3 Pick and Place Demo

For the pick-and-place demo, we were expected to extend upon the work carried out in the trajectory planning section by creating a sequence of motions by which the robot would move towards a point of above the object, wait 2 seconds then move to the object and grip it. Once the gripper is activated and the robot is jogged in any direction, the object should move with it whilst maintaining its orientation in the gripper. When the gripper is activated, the object is expected to be returned directly to its initial position when the gripper is deactivated no matter where the robot is.

To accomplish this, rather than getting the robot to move towards the target directly. We add a constant (40) to our z-axis value (regardless if it is obtained from vector addition or from one of the vision/image processing programs) so that the gripper is directly above the object, the X, Y an Z values are decided and the InverseTrajectoryButtonPushed function is called to go through with the first part of the movement. This was done in the following line of code implemented in the ApproachCubeButtonPushed and ApproachCylinderButtonPushed functions as seen below in Listing 2.3.1. After reaching the point above the target it would pause for 2 seconds using the "pause(2)" line of code. Then the InverseTrajectoryButtonPushed function (Listing 2.2.2) would be called again and the robot would move down towards the object in the specified time. The code for the Approach Buttons is seen below in Listing 2.3.2. The vision guided approach buttons use the same technique but are demonstrated earlier.

Firstly, to create something that could allow us to check if the object was grabbed we created the Grab buttons and linked them to the object grabbed check boxes. The buttons act like toggle buttons where everytime you press them you toggle between grabbed and not grabbed. This is seen in the code in Listing 2.3.3. Then, to get the object to move we altered some lines in the ForwardButtonPushed function that would allow us to constantly if the gripper was activated. To know if the gripper was activated, we check if the checkbox is ticked or not. If it is, then we get the object to move with the end effector. In the ForwardButtonPushed function, we were constantly updating the position of the object to match the position of the end effector using the code found in lines 6 and 9 of Listing 2.3.4.

This code multiplies the both the scale factor of [1,-1,-1] to account for differences in frame orientation. However, there was an offset and so this is corrected by adding PRobotWorld multiplied by the scale factor [-1,1,1] to correct for frame orientations. We then update TRobotObjectCylinder (line 9) to include NewCylinderPos which then allows it to move with the end effector.

However, a problem was that the object's orientation was staying constant and wasn't changing correctly with the tool frame. To overcome this, we developped the code below which creates Rx, Ry and Rz matrices (lines 2, 3 and 4 of Listing 2.3.4) and uses them to constantly update the orientation of the object as it is moving. We call on the rotation matrix from the overall T03 transformation matrix using T03(1:3,1:3) and multiply it by the generic 90 degree rotation matrices Rx, Ry and Rz and then by the rotation of the cylinder RWorldObjectCylinder and then by the scale factor [-1,1,1] to correct for difference in frame orientation between world. This is found in Listing 2.3.4 as well.

The example in Listing 2.3.4 shows the code for a cylinder.The code for the cube is almost the same but is shown in Listing 2.3.5. In both, the if condition checks that the object is grabbed by checking if the checkbox is ticked before allowing them to move with the end effector.

Finally, once the gripper is deactivated, the object automatically returns to its original position by calling the ForwardButtonPushed function whenever the GrabObject checkbox is unticked as this will prompt the object to go back to its original position (Lines 6-7 and 17-18 in Listing 2.3.3). To help with jogging the robot more easily in the demo we created sliders and +10, +1, -1 and -10 buttons.

2.3.1 Matlab Code

Listing 2.3.1: Vector Addition to find object position with respect to robot frame

```
1 L1 = 0;    % mm, Robot link lengths
2              global L2;
3              L2 = 150; % mm
4              L3 = 100; % mm
5              Le = 50;  % mm, end-effector, assume L2 = L3 + Le so ...
                   that calculation of workspace easier
6
7              thetaxRobotWorld = 180.1;           % Transformation ...
                   between robot and world frames
8              thetayRobotWorld = 0.1;             % thetax needs to be ...
                   around 180 because world frame is flipped
```

```
9               thetazRobotWorld = 1;                % thetay and thetaz ...
                    small so that world and robot somewhat aligned apart
10              PRobotWorld = [-100;200;-20];        % from being upside ...
                    down.
11
12              RzWorldObjectBox = 20;               % Transformation ...
                    between world and box (cube)
13                                                   % Rotation should ...
                                                        only be about ...
                                                        z-axis because ...
                                                        during calibration,
14                                                   % we put ...
                                                        checkerboard on ...
                                                        object.
15                                                   % This is not an ...
                                                        assumption but truth.
16              PWorldObjectBox = [-50;100;0];       % Box frame wrt. ...
                    world frame (fixed)
17                                                   % z-value should be ...
                                                        zero because we ...
                                                        assumed
18                                                   % checkerboard ...
                                                        (world frame) was ...
                                                        on top of
19                                                   % object.
20              PWorldObjectCylinder = [100;50;0];   % Cylinder frame ...
                    wrt. world frame (fixed)
21                                                   % z-value should be ...
                                                        zero because we ...
                                                        assumed
22                                                   % checkerboard ...
                                                        (world frame) was ...
                                                        on top of
23                                                   % object.
24
25              global PRobotCylinder
26              global PRobotBox
27              PRobotCylinder = PRobotWorld - ...
                    [-1;1;1].*PWorldObjectCylinder;
28              PRobotBox = PRobotWorld - [-1;1;1].*PWorldObjectBox;
```

Listing 2.3.2: Showing the Approach Object Buttons which are used to implement the two-step path to the object with the 2 second pause in the middle

```
1  function ApproachCubeButtonPushed(app, event)
2              global PRobotBox
3              app.XEditField.Value = PRobotBox(1);
4              app.YEditField.Value = PRobotBox(2);
5              app.ZEditField.Value = PRobotBox(3)+40;
6              InverseTrajectoryButtonPushed(app,event);
7              pause(2)
8              app.ZEditField.Value = PRobotBox(3);
9              InverseTrajectoryButtonPushed(app,event);
10         end
11
12 % Button pushed function: ApproachCylinderButton
13 function ApproachCylinderButtonPushed(app, event)
```

```
            global PRobotCylinder
            app.XEditField.Value = PRobotCylinder(1);
            app.YEditField.Value = PRobotCylinder(2);
            app.ZEditField.Value = PRobotCylinder(3) + 40;
            InverseTrajectoryButtonPushed(app,event);
            pause(2)
            app.ZEditField.Value = PRobotCylinder(3);
            InverseTrajectoryButtonPushed(app,event);
        end
```

Listing 2.3.3: Showing the GrabButton functions which alter the checkboxes and instruct the object to start moving with it or return to initial position

```
% Button pushed function: GrabCubeButton
        function GrabCubeButtonPushed(app, event)
            if app.CubeGrabbedCheckBox.Value == 0
                app.CubeGrabbedCheckBox.Value = 1;
            else
                app.CubeGrabbedCheckBox.Value = 0;
                ForwardButtonPushed(app,event);
            end
        end

% Button pushed function: GrabCylinderButton
        function GrabCylinderButtonPushed(app, event)

            if app.CylinderGrabbedCheckBox.Value == 0
                app.CylinderGrabbedCheckBox.Value = 1;
            else
                app.CylinderGrabbedCheckBox.Value = 0;
                ForwardButtonPushed(app,event);
            end
        end
```

Listing 2.3.4: The code which gets the cylinder to move with the end effector

```
if app.CylinderGrabbedCheckBox.Value
                Rx = -[1, 0 ,0; 0 cosd(90) sind(90); 0 -sind(90) ...
                    cosd(90)];
                Ry = -[cosd(90),0, sind(90); ...
                    0,1,0;-sind(90),0,cosd(90)];
                Rz = -[cosd(90) sind(90) 0; -sind(90) cosd(90) 0; 0 ...
                    0 1];

            NewCylinderPos = ...
                (P03(1:3).*[1;-1;-1])+(PRobotWorld.*[-1;1;1]); ...
                %Setting cylinder pos. = tool frame pos.
            NewRotCyl = ...
                T03(1:3,1:3)*Rx*Ry*Rz*RWorldObjectCylinder.*[-1;1;1];

            TRobotObjectCylinder = TRobotWorld*[(NewRotCyl) ...
                NewCylinderPos;0 0 0 1]; % world wrt. robot (fixed) * ...
                object wrt. world (fixed) = object wrt. robot (fixed)
```

Listing 2.3.5: The code which gets the cube to move with the end effector

```
if app.CubeGrabbedCheckBox.Value
                    ObjectBoxCorners = [-10 -10 20 1; % The 1 in last ...
                        column is just for homogeneous coordinates
                10 -10 20 1;
                10 10 20 1;
                -10 10 20 1;
                -10 -10 0 1;
                10 -10 0 1;
                10 10 0 1;
                -10 10 0 1];
            NewBoxPos = ...
                (P03(1:3).*[1;-1;-1])+(PRobotWorld.*[-1;1;1]); %Set ...
                new box pos. = tool frame pos.
            Rx = -[1, 0 ,0; 0 cosd(90) sind(90); 0 -sind(90) cosd(90)];
            Ry = -[cosd(90),0, sind(90); 0,1,0;-sind(90),0,cosd(90)];
            Rz = -[cosd(90) sind(90) 0; -sind(90) cosd(90) 0; 0 0 1];
            RWorldObjectBox = [cosd(RzWorldObjectBox) ...
                -sind(RzWorldObjectBox) 0;
                 sind(RzWorldObjectBox) cosd(RzWorldObjectBox) 0;
                 0 0 1];
            NewRotBox = ...
                T03(1:3,1:3)*Rx*Ry*Rz*RWorldObjectBox.*[-1;1;1];%Set ...
                rot. box = rot. tool frame
            TRobotObjectBox = TRobotWorld*[(NewRotBox) NewBoxPos;0 0 ...
                0 1]; % world wrt. robot (fixed) * object wrt. world ...
                (fixed) = object wrt. robot (fixed)
            ObjectBoxCornersTransformed = ...
                TRobotObjectBox*ObjectBoxCorners';
            ObjectBoxCornersTransformed =  ObjectBoxCornersTransformed';
```