Robotics Vision – Group 5

Task 1 - Camera Calibration Given Simulated Checkerboard Data

Method and Derivation

We perform the camera calibration to obtain the intrinsic and extrinsic parameters of the camera. With these parameters we will then be able to get the position and the orientation of the World frame with respect to the camera. The intrinsic and extrinsic equations are shown below:

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

$$Where\ \alpha, \beta, \gamma, x_0, y_0 are\ the\ intrinsic\ paramters$$

$$Extrinsic: \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = {}_W^C R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + {}^C P_{Worg}$$

$$Where\ {}_W^C R\ and + {}^C P_{Worg}\ are\ the\ extrinsic\ paramters$$

With these equations, we can obtain a camera matrix H:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \sim K[r_1 \quad r_2 \quad r_3 \qquad {}^C P_{Worg}] * \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = K[r_1 \quad r_2 \qquad {}^C P_{Worg}] * \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = H \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

We know the values of x, y, X, Y but not H, so to do the camera calibration, we we need to take as many pictures as possible of a checkerboard in different positions and orientations where we can get the (X_i, Y_i) coordinates with respect to the world frame. After acquiring the pictures, we then need the pixel position coordinates (x_i,y_i) of the corner points of each image for which we get the equation:

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

The proportional sign means we cannot calculate h values, however it also means the left side is a scalar multiple of the right side and therefore the cross product is 0:

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

And, from this, we can form a matrix with all the points of the checkerboard forming the following equation:

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

We can then find the h vector using Singular Value Decomposition thus we will calculate the matrix H. This is only for one image of the checkerboard, so we need to repeat this process for each image.

Using the know h values we can then calculate $v_{11}^T, v_{22}^T, v_{12}^T$ so we get:

$$v_{ij}^T = |h_{1i}h_{1j} \quad h_{2i}h_{1j} + h_{1i}h_{2j} \quad h_{2i}h_{2j} \quad h_{3i}h_{1j} + h_{1i}h_{3j} \quad h_{3i}h_{2j} + h_{2i}h_{3j} \quad h_{3i}h_{3j}|$$

We then form the following equation putting in the variables obtained from each image:

$$\begin{bmatrix} v_{12}^T \\ v_{11}^T - v_{22}^T \\ . \\ . \\ . \\ v_{12}^T \\ v_{11}^T - v_{22}^T \end{bmatrix} * \begin{bmatrix} b_{11} \\ b_{12} \\ b_{22} \\ b_{13} \\ b_{23} \\ b_{33} \end{bmatrix} = 0$$

Now we find the vector b using again, Singular Value Decomposition.

We also know that:

$$B = K^{-T}K^{-1} = \begin{bmatrix} \frac{1}{\alpha^2} & -\frac{\gamma}{\alpha^2\beta} & \frac{y_0\gamma - x_0\beta}{\alpha^2\beta} \\ -\frac{\gamma}{\alpha^2\beta} & \frac{\gamma^2}{\alpha^2\beta^2} + \frac{1}{\beta^2} & -\frac{\gamma(y_0\gamma - x_0\beta)}{\alpha^2\beta^2} - \frac{y_0}{\beta^2} \\ \frac{y_0\gamma - x_0\beta}{\alpha^2\beta} & -\frac{\gamma(y_0\gamma - x_0\beta)}{\alpha^2\beta^2} - \frac{y_0}{\beta^2} & \frac{(y_0\gamma - x_0\beta)^2}{\alpha^2\beta^2} + \frac{y_0^2}{\beta^2} + 1 \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{12} & b_{22} & b_{23} \\ b_{13} & b_{23} & b_{33} \end{bmatrix}$$

After obtaining the vectors h and b, we can go and recover the intrinsic parameters using the following equations:

$$y_0 = \frac{b_{11}b_{13} - b_{11}b_{23}}{b_{11}b_{22} - b_{12}^2}$$

$$\lambda = b_{33} - \frac{b_{13}^2 + y_0(b_{12}b_{13} - b_{11}b_{23})}{b_{11}}$$

$$\alpha = \sqrt{\frac{\lambda}{b_{11}}}$$

$$\beta = \sqrt{\frac{\lambda b_{11}}{b_{11}b_{22} - b_{12}^2}}$$

$$\gamma = -\frac{b_{12}\alpha^2\beta}{\lambda}$$

$$x_0 = \frac{\gamma y_0}{\alpha} - \frac{b_{13}\alpha^2}{\lambda}$$

So, now that we have the intrinsic parameters and with them, we form the matrix K:

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

And then go onto find the extrinsic parameters. We know that for each image we have

$$H = [h_1 \quad h_2 \quad h_3] = K\left[r_1 \quad r_2 \quad {}^{C}P_{Worg}\right]$$

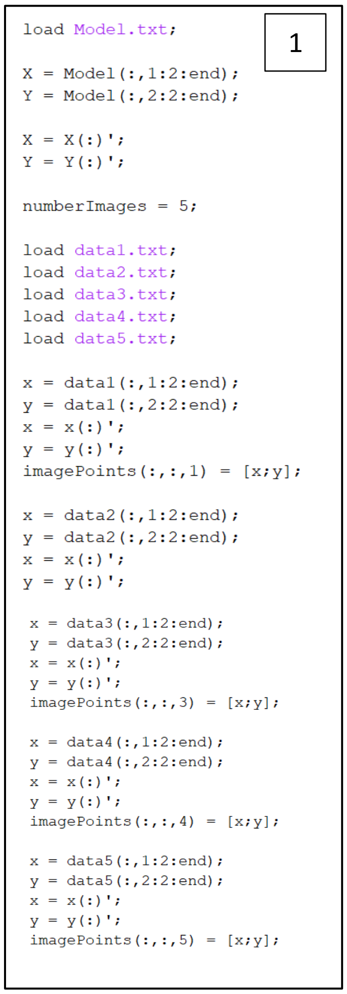
We find the extrinsic parameters using a scaling factor $\sigma = \frac{1}{\|K^{-1}h_1\|}$ to be:

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

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

$${}^{C}P_{Worg} = \sigma K^{-1}h_3$$

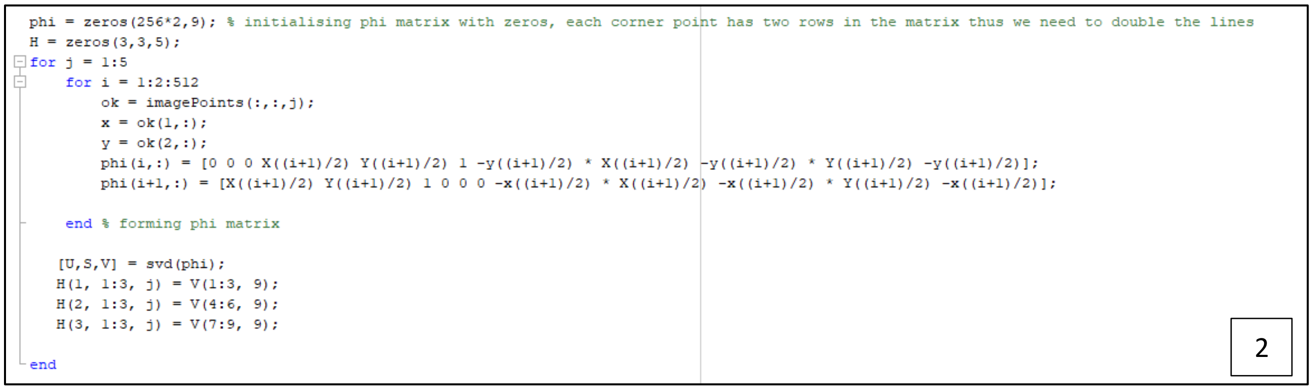
MATLAB code



1

```
load Model.txt;

X = Model(:,1:2:end);
Y = Model(:,2:2:end);

X = X(:)';
Y = Y(:)';

numberImages = 5;

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

x = data1(:,1:2:end);
y = data1(:,2:2:end);
x = x(:)';
y = y(:)';
imagePoints(:,:,1) = [x;y];

x = data2(:,1:2:end);
y = data2(:,2:2:end);
x = x(:)';
y = y(:)';

 x = data3(:,1:2:end);
 y = data3(:,2:2:end);
 x = x(:)';
 y = y(:)';
 imagePoints(:,:,3) = [x;y];

 x = data4(:,1:2:end);
 y = data4(:,2:2:end);
 x = x(:)';
 y = y(:)';
 imagePoints(:,:,4) = [x;y];

 x = data5(:,1:2:end);
 y = data5(:,2:2:end);
 x = x(:)';
 y = y(:)';
 imagePoints(:,:,5) = [x;y];
```

1. We first load the corner points of the physical printout checkerboard given in 'Model.txt'. 'data1.txt' to 'data5.txt' gives the pixel coordinates of the 5 images. Typically, if the corner points are not given, a corner detection algorithm is used to achieve these points instead.

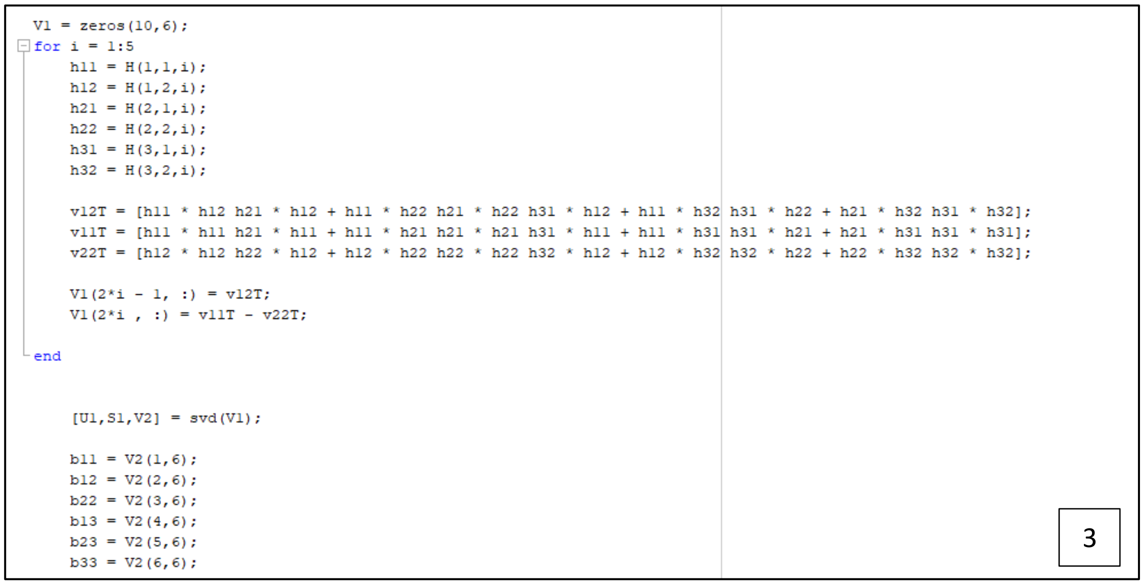


```
phi = zeros(256*2,9); % initialising phi matrix with zeros, each corner point has two rows in the matrix thus we need to double the lines
H = zeros(3,3,5);
for j = 1:5
    for i = 1:2:512
        ok = imagePoints(:,:,j);
        x = ok(1,:);
        y = ok(2,:);
        phi(i,:) = [0 0 0 X((i+1)/2) Y((i+1)/2) 1 -y((i+1)/2) * X((i+1)/2) -y((i+1)/2) * Y((i+1)/2) -y((i+1)/2)];
        phi(i+1,:) = [X((i+1)/2) Y((i+1)/2) 1 0 0 0 -x((i+1)/2) * X((i+1)/2) -x((i+1)/2) * Y((i+1)/2) -x((i+1)/2)];

    end % forming phi matrix

   [U,S,V] = svd(phi);
   H(1, 1:3, j) = V(1:3, 9);
   H(2, 1:3, j) = V(4:6, 9);
   H(3, 1:3, j) = V(7:9, 9);

end
```

2

2. We then build the matrix phi for each image to find the h vector using singular value decomposition. The for loop goes from one to five as there are 5 images, and phi is double the lines as each corner point has two rows in the matrix.

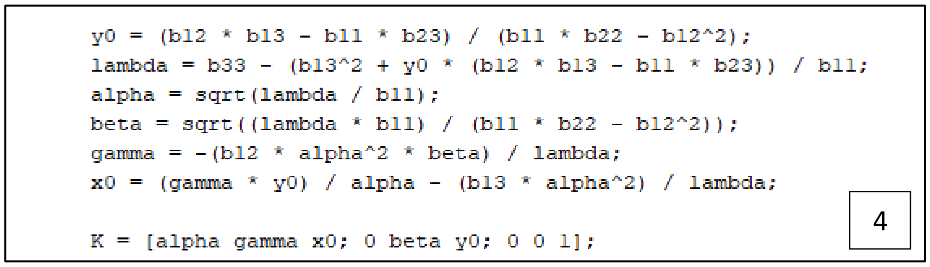
   In order to calculate the h vector we use the "svd()" function in matlab, after which we form the vector h by taking the last column of V.



```
V1 = zeros(10,6);
for i = 1:5
    h11 = H(1,1,i);
    h12 = H(1,2,i);
    h21 = H(2,1,i);
    h22 = H(2,2,i);
    h31 = H(3,1,i);
    h32 = H(3,2,i);

    v12T = [h11 * h12 h21 * h12 + h11 * h22 h21 * h22 h31 * h12 + h11 * h32 h31 * h22 + h21 * h32 h31 * h32];
    v11T = [h11 * h11 h21 * h11 + h11 * h21 h21 * h21 h31 * h11 + h11 * h31 h31 * h21 + h21 * h31 h31 * h31];
    v22T = [h12 * h12 h22 * h12 + h12 * h22 h22 * h22 h32 * h12 + h12 * h32 h32 * h22 + h22 * h32 h32 * h32];

    V1(2*i - 1, :) = v12T;
    V1(2*i , :) = v11T - v22T;

end


    [U1,S1,V2] = svd(V1);

    b11 = V2(1,6);
    b12 = V2(2,6);
    b22 = V2(3,6);
    b13 = V2(4,6);
    b23 = V2(5,6);
    b33 = V2(6,6);
```

3

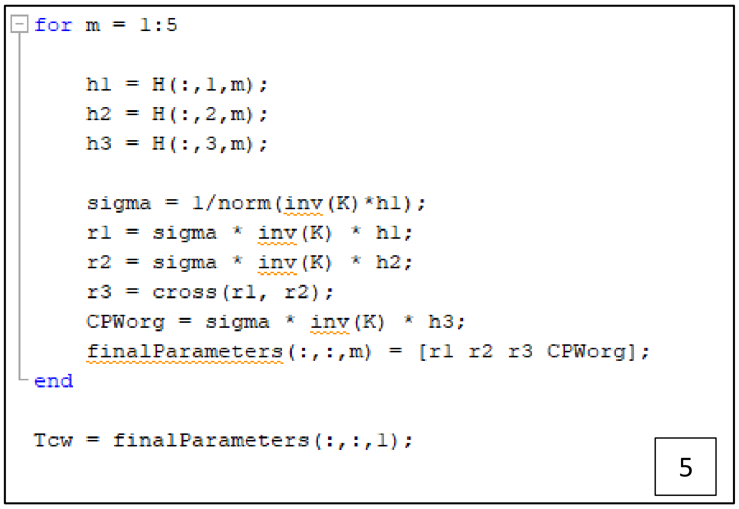
3. After finding h, we can go and find the values $v_{11}^T, v_{22}^T, v_{12}^T$. To do that we just follow the equations. After finding these values we form the matrix v for each image, and in the end we find the b vector by again applying the “svd()” function on the v matrix, and taking the last column of V2.



```
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];
```

4

4. After finding the b vector we were able to recover the intrinsic parameters (and put them in matrix K), by just following the equations.

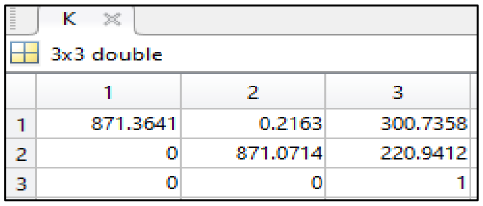


```
for m = 1:5

    h1 = H(:,1,m);
    h2 = H(:,2,m);
    h3 = H(:,3,m);

    sigma = 1/norm(inv(K)*h1);
    r1 = sigma * inv(K) * h1;
    r2 = sigma * inv(K) * h2;
    r3 = cross(r1, r2);
    CPWorg = sigma * inv(K) * h3;
    finalParameters(:,:,m) = [r1 r2 r3 CPWorg];
end

Tcw = finalParameters(:,:,1);
```

5

5.In the end we recovered the extrinsic parameters for each image in “finalParameters” and also saved the extrinsic parameters of the first images as “Tcw” as this is going to be for the world coordinates calibration.

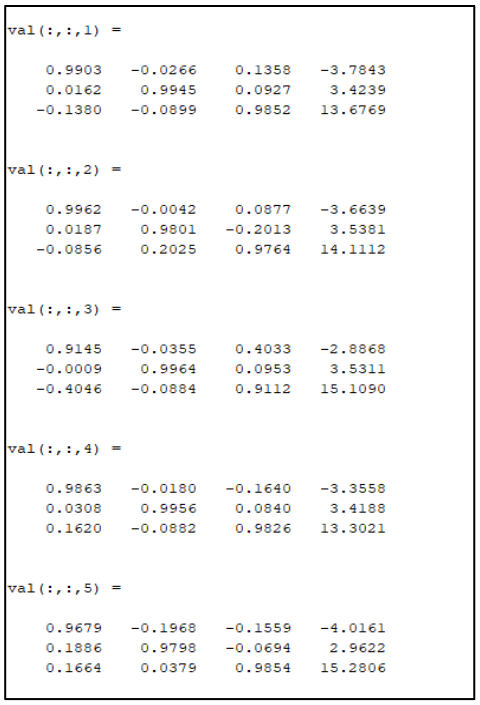
Results



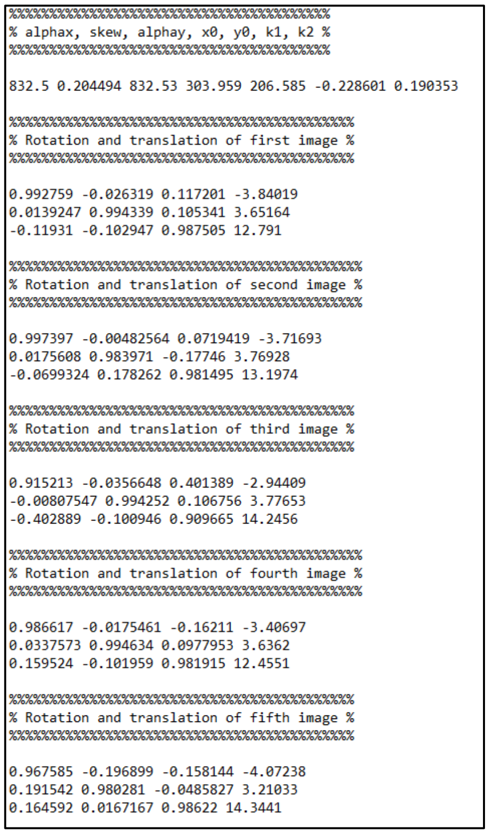
K

3x3 double

	1	2	3
1	871.3641	0.2163	300.7358
2	0	871.0714	220.9412
3	0	0	1



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

    0.9903   -0.0266    0.1358   -3.7843
    0.0162    0.9945    0.0927    3.4239
   -0.1380   -0.0899    0.9852   13.6769


val(:,:,2) =

    0.9962   -0.0042    0.0877   -3.6639
    0.0187    0.9801   -0.2013    3.5381
   -0.0856    0.2025    0.9764   14.1112


val(:,:,3) =

    0.9145   -0.0355    0.4033   -2.8868
   -0.0009    0.9964    0.0953    3.5311
   -0.4046   -0.0884    0.9112   15.1090


val(:,:,4) =

    0.9863   -0.0180   -0.1640   -3.3558
    0.0308    0.9956    0.0840    3.4188
    0.1620   -0.0882    0.9826   13.3021


val(:,:,5) =

    0.9679   -0.1968   -0.1559   -4.0161
    0.1886    0.9798   -0.0694    2.9622
    0.1664    0.0379    0.9854   15.2806
```



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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
```

For the results we can see that the values we obtained (left) are very similar to Zhang's results (right), thus we can say that our code works properly.

Task 2 – Robot Calibration given simulated TCP data

Method and Derivation

After camera calibration (Task 1) is done, we will now do the robot calibration. For task 2, instead of placing the end-effector TCP on the corner points of the checker board. We were instead given sample positions of the checkerboard corner points according to the robot coordinate system (X_{ri}, Y_{ri}, Z_{ri}) shown in table 1.0, as well as the positions of each corner point of the checkerboard (X_i, Y_i, 0) with respect to the world frame shown in table 1.1.

Point	**1**	**2**	**3**	**4**	**5**	**6**
X_{ri}	-121.6	-100.4	-79.16	-56.39	-35.55	54.32
Y_{ri}	222.4	178	133.6	177.2	110.9	219.3
Z_{ri}	-20	-20.04	-20.08	-19.96	-20.04	-19.7

Table 1.0 Coordinates of simulated probed positions of the TCP

Point	1	2	3	4	5	6
X$_i$	-22	0	22	44	66	154
Y$_i$	-22	22	66	22	88	-22

Table 1.1 Coordinates of checkerboard corner points

Since we have the same 'points' expressed in 2 different frames – world and robot frames. We use the transformation matrix to find the relationship between these frames. In the world frame, there is no 'Z' because we are looking at a 2D plane. In the transformation matrix we need to solve for 2 parameters: ρ_{ij} and $t_{x,y,z}$.

From here, we can find the relationship between the 2 frames as:

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

$$To\ solve\ for\ the\ unknowns\ we\ can\ firstly\ convert\ them\ into\ individual\ linear\ equations$$

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

$$Y_{ri} = \rho_{21} X_i + \rho_{22} Y_i + t_y$$

$$Z_{ri} = \rho_{31} X_i + \rho_{32} Y_i + t_z$$

$$Unknowns\ can\ be\ obtained\ by\ minimizing\ the\ squre\ of\ the\ errors, where\ minimum$$

$$square\ of\ errors\ occurs\ when\ derivative\ are\ zero, which\ can\ eventually\ be\ represented$$

$$in\ a\ matrix\ to\ find\ the\ rotation\ and\ translation\ parameters:$$

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

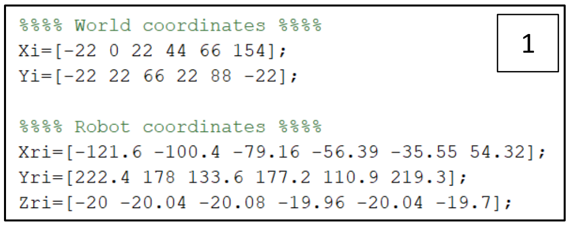
$$\begin{bmatrix} \rho_{21} \\ \rho_{22} \\ t_y \end{bmatrix} = \begin{bmatrix} \sum X_i X_i & \sum Y_i X_i & \sum X_i \\ \sum X_i Y_i & \sum Y_i Y_i & \sum Y_i \\ \sum X_i & \sum Y_i & n \end{bmatrix}^{-1} * \begin{bmatrix} \sum Y_{ri} X_i \\ \sum Y_{ri} Y_i \\ \sum Y_{ri} \end{bmatrix}$$

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

$$And\ finally\ combining, we\ get\ \rho_{13}, \rho_{23}, \rho_{33}:$$

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

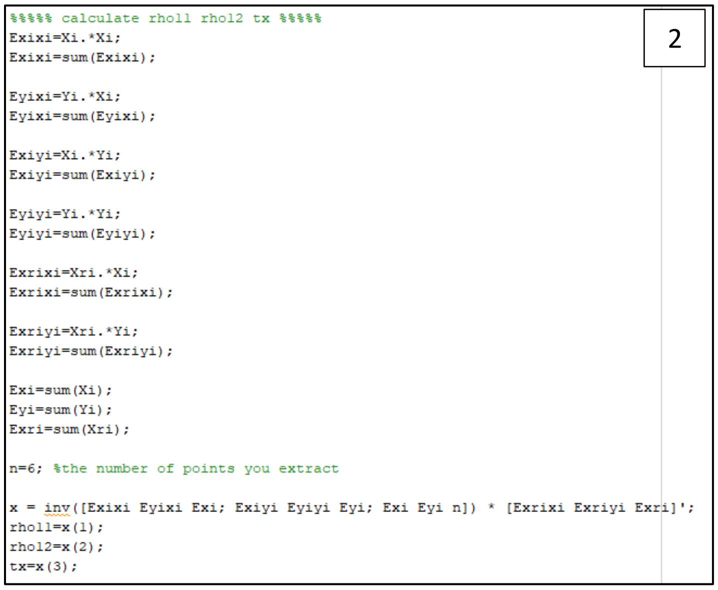
MATLAB code



```
%%%% World coordinates %%%%
Xi=[-22 0 22 44 66 154];
Yi=[-22 22 66 22 88 -22];

%%%% Robot coordinates %%%%
Xri=[-121.6 -100.4 -79.16 -56.39 -35.55 54.32];
Yri=[222.4 178 133.6 177.2 110.9 219.3];
Zri=[-20 -20.04 -20.08 -19.96 -20.04 -19.7];
```

1

1. Firstly, we define the given example coordinates of the world and robot coordinates.

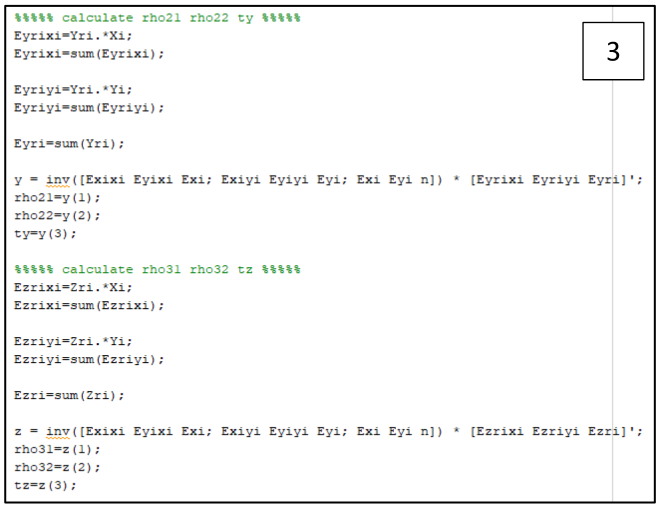


```
%%%%% calculate rho11 rho12 tx %%%%%
Exixi=Xi.*Xi;
Exixi=sum(Exixi);

Eyixi=Yi.*Xi;
Eyixi=sum(Eyixi);

Exiyi=Xi.*Yi;
Exiyi=sum(Exiyi);

Eyiyi=Yi.*Yi;
Eyiyi=sum(Eyiyi);

Exrixi=Xri.*Xi;
Exrixi=sum(Exrixi);

Exriyi=Xri.*Yi;
Exriyi=sum(Exriyi);

Exi=sum(Xi);
Eyi=sum(Yi);
Exri=sum(Xri);

n=6; %the number of points you extract

x = inv([Exixi Eyixi Exi; Exiyi Eyiyi Eyi; Exi Eyi n]) * [Exrixi Exriyi Exri]';
rho11=x(1);
rho12=x(2);
tx=x(3);
```

2

2. We then wrote out the variables that appear in the equations proven above, so that they could be substituted in a matrix.

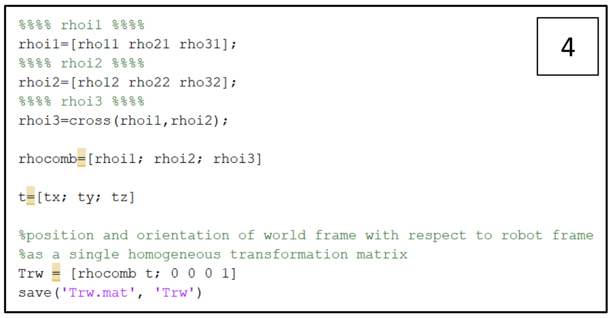
Then did the first calculation for ρ_{11}, ρ_{12}, t_x.



```
%%%%% calculate rho21 rho22 ty %%%%%
Eyrixi=Yri.*Xi;
Eyrixi=sum(Eyrixi);

Eyriyi=Yri.*Yi;
Eyriyi=sum(Eyriyi);

Eyri=sum(Yri);

y = inv([Exixi Eyixi Exi; Exiyi Eyiyi Eyi; Exi Eyi n]) * [Eyrixi Eyriyi Eyri]';
rho21=y(1);
rho22=y(2);
ty=y(3);

%%%%% calculate rho31 rho32 tz %%%%%
Ezrixi=Zri.*Xi;
Ezrixi=sum(Ezrixi);

Ezriyi=Zri.*Yi;
Ezriyi=sum(Ezriyi);

Ezri=sum(Zri);

z = inv([Exixi Eyixi Exi; Exiyi Eyiyi Eyi; Exi Eyi n]) * [Ezrixi Ezriyi Ezri]';
rho31=z(1);
rho32=z(2);
tz=z(3);
```

3

3. The variables were changed to calculate ρ_{21}, ρ_{22}, t_y and the same method was applied to ρ_{31}, ρ_{32}, t_z.



```
%%%% rhoi1 %%%%
rhoi1=[rho11 rho21 rho31];
%%%% rhoi2 %%%%
rhoi2=[rho12 rho22 rho32];
%%%% rhoi3 %%%%
rhoi3=cross(rhoi1,rhoi2);

rhocomb=[rhoi1; rhoi2; rhoi3]

t=[tx; ty; tz]

%position and orientation of world frame with respect to robot frame
%as a single homogeneous transformation matrix
Trw = [rhocomb t; 0 0 0 1]
save('Trw.mat', 'Trw')
```

4

4. Lastly, the values calculated can be summarized in their respective matrixes and saved as a single homogenous transformation matrix.

Results

We finally get the result for Trw:

```
Trw =

    0.9996   -0.0175    0.0017 -100.0000
   -0.0174   -0.9998   -0.0017  199.9962
    0.0017    0.0017   -0.9997  -20.0006
         0         0         0    1.0000
```

Which is very close to the expected final answer we were given of:

$$\rho_{ij} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \; And \; t_{x,y,z} = \begin{bmatrix} -100 \\ 200 \\ -20 \end{bmatrix}$$

Task 3 – Image processing given simulated image

Method and Derivation

For task 3, our goal was to find the shape colour, position and orientation of the objects in a simulated image (figure 1.0).

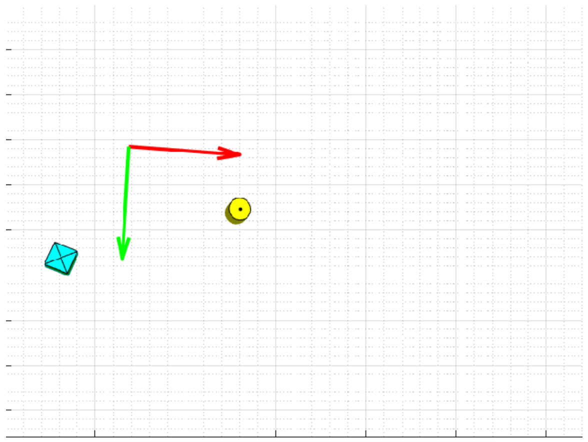


Figure 1.0: CameraView.bmp

In order to find the 2D position of the object in a pixel frame, we had to go through a method of image processing. With image processing, we adapt the image to allow us to analyse the picture better, so that we can perform tasks such as object recognition.

The first step in image processing is to convert the colour image to a grey-scale image. Colour images have 3 layers of red, green and blue. This is quite complex so we will grey-scale the colour image. Typically, the colour we see is an additive property of the 3 colours so taking an average of the 3 colours:

$$I(grey) = \frac{I(Red) + I(Green) + I(Blue)}{3}$$

We can get an average value of the 3 colours. Through grey-scaling, the brightness of the pixel is determined by its value, where max value of 255 is for white and 0 is for black. Therefore we can use this to perform thresholding which allows us to highlight important information such as the object we are trying to capture in the image.

Afterwards, we binarize the image by making values above a threshold in the grey-scale image = 1 and values below = 0 which would give a purely black and white image.

As can be seen from the given image, there is a black dot in the centre of the circle and cross in the square which can be interpreted as 'noise' which we want to remove. Therefore we use a method of pixel group processing called Median filtering, which is a nonlinear processing method. Averaging methods can be susceptible to outliers which can heavily affect the outcome, therefore using median filters, no outliers will affect it and instead will be almost completely removed. In our case, our median filter takes into account 3 pixels ahead and behind of the current (i,j) pixel coordinate because the 'noise' was quite large and couldn't be filtered with anything less.

Next, we performed blob detection, finding all the properties of the object such as centroid, circularity, perimeter, orientation.

To find the center of the object we used moments where the p-qth moment of an image is defined as:

$$M_{pq} = \sum_{(x,y)\epsilon Image} x^p * y^q * I(x,y)$$

This essentially is the sum of x location to power p and y location power q, multiplied by intensity of pixel (=1) at that x,y location (distance). For example, at p=q=0 we get:

$$M_{00} = \sum_{(x,y)\epsilon Image} x^0 * y^0 * I(x,y)$$

$$M_{00} = \sum_{(x,y)\epsilon Image} I(x,y)$$

Which is the sum of all the pixels in the object and therefore is the area.

Furthermore, using moments, we can calculate even more parameters, starting with centroid, the formula is:

$$X_c = \frac{M_{10}}{M_{00}}, Y_c = \frac{M_{01}}{M_{00}}$$

Now that we have the centroid, we can calculate all moments with respect to the center of the object using a formula for the central moment:

$$\mu_{pq} = \sum_{(x,y)\epsilon Image} (x - X_c)^p * (y - Y_c)^q * I(x,y)$$

$$From\ here, we\ can\ deduce\ the\ following:$$

$$\mu_{00} = M_{00}$$

$$\mu_{01} = 0$$

$$\mu_{10} = 0$$

$$\mu_{11} = M_{11} - X_c M_{01} = M_{11} - Y_c M_{10}$$

$$\mu_{02} = M_{20} - X_c M_{10}$$

$$\mu_{20} = M_{02} - Y_c M_{01}$$

$$And\ now\ we\ also\ have\ the\ inertia\ matrix\ of\ the\ blob:$$

$$J = \begin{bmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{bmatrix}$$

$$For\ which\ we\ can\ calculate\ all\ ellipse\ parameters\ a(major\ axis)$$

$$and\ b(minor\ axis)\ as\ well\ as\ orientation\ \theta:$$

$$a = 2\sqrt{\frac{\lambda_1}{M_{00}}}, b = 2\sqrt{\frac{\lambda_2}{M_{00}}}, \theta = \arctan\left(\frac{V_y}{V_x}\right)$$

$$Where\ \lambda_i\ are\ the\ eigenvectors\ of\ J\ with\ \lambda_1 > \lambda_2$$

$$and\ V = \begin{bmatrix} V_x & V_y \end{bmatrix}^T\ is\ the\ eigenvector\ corresponding\ to\ \lambda_1$$

As we now have the 2 essential properties, centroid and orientation, the robot in theory can pick up the object. However, to make things easier for us to recognize and detect objects, we can perform a few calculations to get corners, or circularity of the object.

Here we will focus on the circularity calculation which has a formula of:

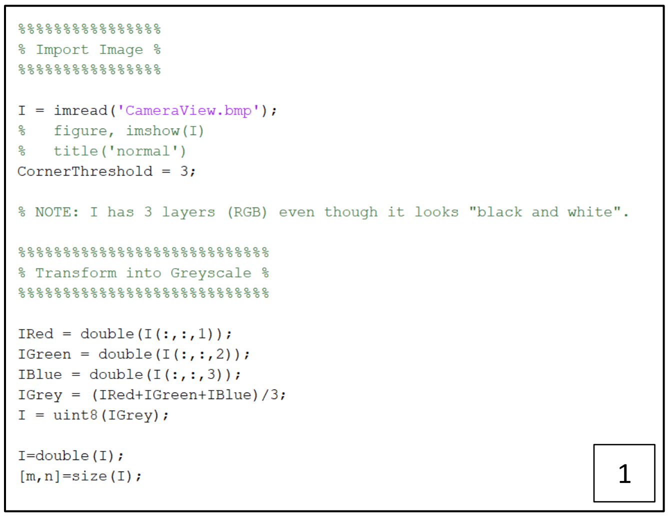
$$Circularity = \frac{4\pi M_{00}}{p^2}$$

$$Where\ M_{00}\ is\ the\ area, and\ p\ is\ the\ perimeter$$

For example, in an ideal case, the circularity for a circle is 1 and π/4 for square.

MATLAB code

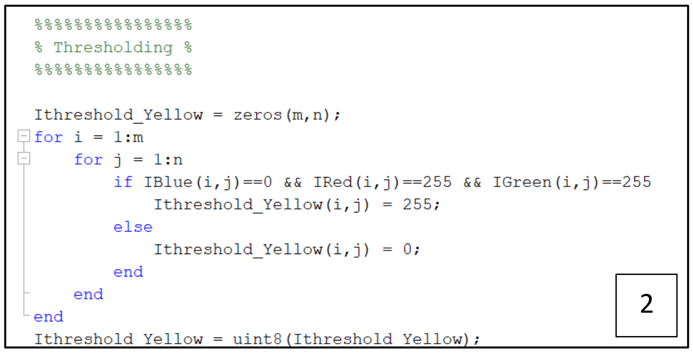
talk about the filter for circle and how not used because it makes circularity worse



```
%%%%%%%%%%%%%%%%
% Import Image %
%%%%%%%%%%%%%%%%

I = imread('CameraView.bmp');
%   figure, imshow(I)
%   title('normal')
CornerThreshold = 3;

% NOTE: I has 3 layers (RGB) even though it looks "black and white".

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Transform into Greyscale %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IRed = double(I(:,:,1));
IGreen = double(I(:,:,2));
IBlue = double(I(:,:,3));
IGrey = (IRed+IGreen+IBlue)/3;
I = uint8(IGrey);

I=double(I);
[m,n]=size(I);
```

1

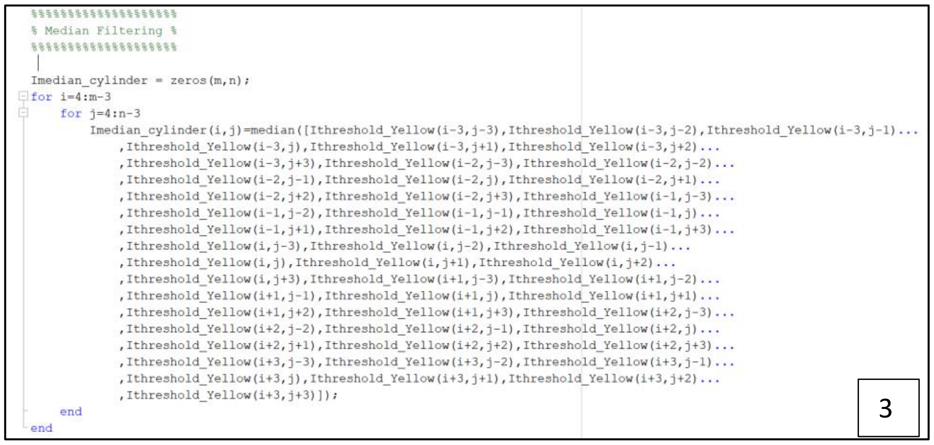
1. The camera image ‘CameraView.bmp’ is imported. Then the image is transformed to greyscale using the ‘unsigned 8-bit integer’ where all elements in the vector have a range from 0-255



```
%%%%%%%%%%%%%%%%
% Thresholding %
%%%%%%%%%%%%%%%%

Ithreshold_Yellow = zeros(m,n);
for i = 1:m
    for j = 1:n
        if IBlue(i,j)==0 && IRed(i,j)==255 && IGreen(i,j)==255
            Ithreshold_Yellow(i,j) = 255;
        else
            Ithreshold_Yellow(i,j) = 0;
        end
    end
end
Ithreshold_Yellow = uint8(Ithreshold_Yellow);
```

2

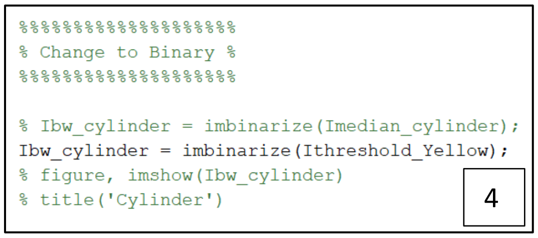
2. Firstly, we will process the cylinder. As we know the cylinder is yellow, this means RGB(255,255,0) which is put as a condition statement. The code will iterate every column for each row, turning all yellow pixels white and other pixels black.



```
%%%%%%%%%%%%%%%%%%%%
% Median Filtering %
%%%%%%%%%%%%%%%%%%%%

Imedian_cylinder = zeros(m,n);
for i=4:m-3
    for j=4:n-3
        Imedian_cylinder(i,j)=median([Ithreshold_Yellow(i-3,j-3),Ithreshold_Yellow(i-3,j-2),Ithreshold_Yellow(i-3,j-1)...
            ,Ithreshold_Yellow(i-3,j),Ithreshold_Yellow(i-3,j+1),Ithreshold_Yellow(i-3,j+2)...
            ,Ithreshold_Yellow(i-3,j+3),Ithreshold_Yellow(i-2,j-3),Ithreshold_Yellow(i-2,j-2)...
            ,Ithreshold_Yellow(i-2,j-1),Ithreshold_Yellow(i-2,j),Ithreshold_Yellow(i-2,j+1)...
            ,Ithreshold_Yellow(i-2,j+2),Ithreshold_Yellow(i-2,j+3),Ithreshold_Yellow(i-1,j-3)...
            ,Ithreshold_Yellow(i-1,j-2),Ithreshold_Yellow(i-1,j-1),Ithreshold_Yellow(i-1,j)...
            ,Ithreshold_Yellow(i-1,j+1),Ithreshold_Yellow(i-1,j+2),Ithreshold_Yellow(i-1,j+3)...
            ,Ithreshold_Yellow(i,j-3),Ithreshold_Yellow(i,j-2),Ithreshold_Yellow(i,j-1)...
            ,Ithreshold_Yellow(i,j),Ithreshold_Yellow(i,j+1),Ithreshold_Yellow(i,j+2)...
            ,Ithreshold_Yellow(i,j+3),Ithreshold_Yellow(i+1,j-3),Ithreshold_Yellow(i+1,j-2)...
            ,Ithreshold_Yellow(i+1,j-1),Ithreshold_Yellow(i+1,j),Ithreshold_Yellow(i+1,j+1)...
            ,Ithreshold_Yellow(i+1,j+2),Ithreshold_Yellow(i+1,j+3),Ithreshold_Yellow(i+2,j-3)...
            ,Ithreshold_Yellow(i+2,j-2),Ithreshold_Yellow(i+2,j-1),Ithreshold_Yellow(i+2,j)...
            ,Ithreshold_Yellow(i+2,j+1),Ithreshold_Yellow(i+2,j+2),Ithreshold_Yellow(i+2,j+3)...
            ,Ithreshold_Yellow(i+3,j-3),Ithreshold_Yellow(i+3,j-2),Ithreshold_Yellow(i+3,j-1)...
            ,Ithreshold_Yellow(i+3,j),Ithreshold_Yellow(i+3,j+1),Ithreshold_Yellow(i+3,j+2)...
            ,Ithreshold_Yellow(i+3,j+3)]);
    end
end
```

3

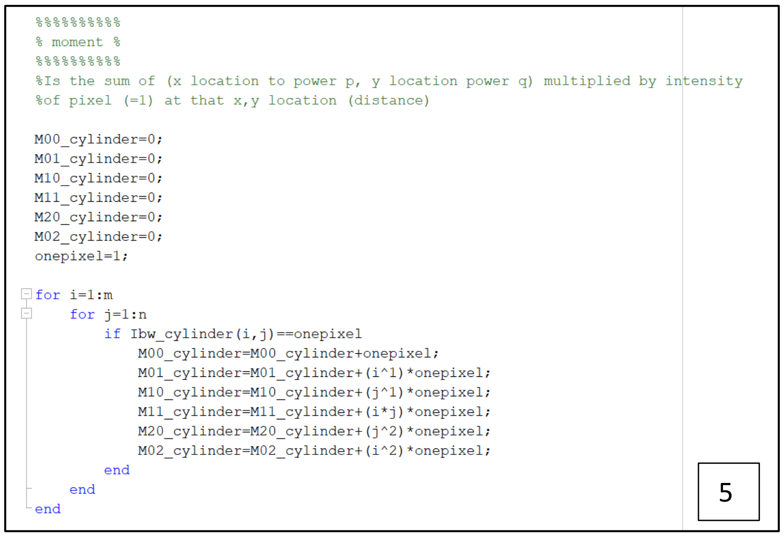
3. A Median filter is then applied to remove noise (the black dot in the centre of the circle). In this case, we had to look 3 pixels ahead and 3 pixels behind in order to remove it.



```
%%%%%%%%%%%%%%%%%%%%
% Change to Binary %
%%%%%%%%%%%%%%%%%%%%

% Ibw_cylinder = imbinarize(Imedian_cylinder);
Ibw_cylinder = imbinarize(Ithreshold_Yellow);
% figure, imshow(Ibw_cylinder)
% title('Cylinder')
```

4

4. Next, we convert the image to binary. As can be seen, there is 2 Ibw_cylinder, the 1 currently commented out is the filtered image and the other is non-filtered image. (see figure 2.0 for the difference between filtered and non-filtered)

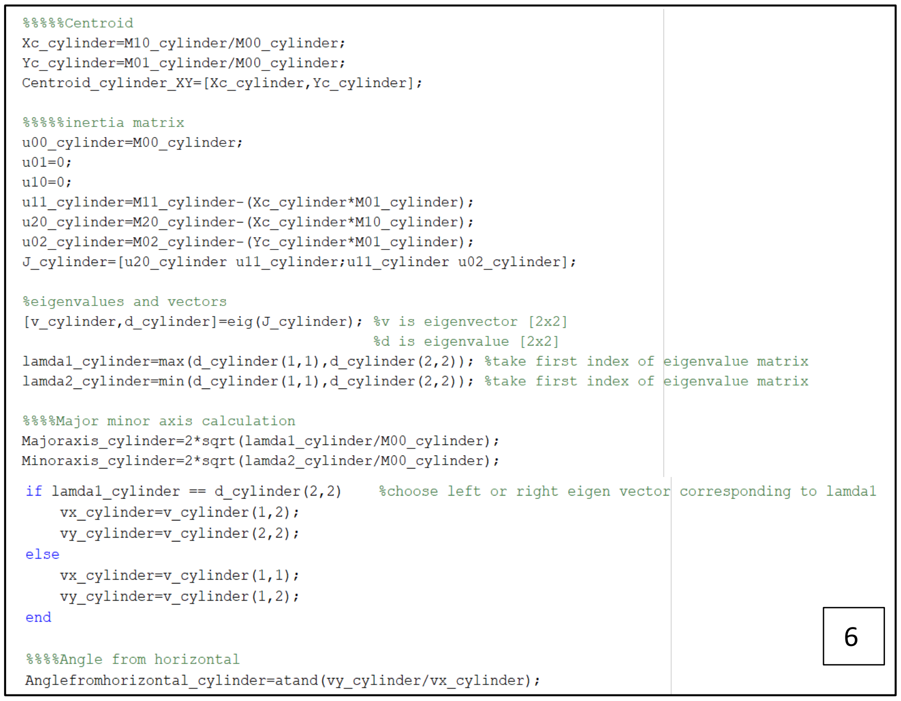


```
%%%%%%%%%%
% moment %
%%%%%%%%%%
%Is the sum of (x location to power p, y location power q) multiplied by intensity
%of pixel (=1) at that x,y location (distance)

M00_cylinder=0;
M01_cylinder=0;
M10_cylinder=0;
M11_cylinder=0;
M20_cylinder=0;
M02_cylinder=0;
onepixel=1;

for i=1:m
    for j=1:n
        if Ibw_cylinder(i,j)==onepixel
            M00_cylinder=M00_cylinder+onepixel;
            M01_cylinder=M01_cylinder+(i^1)*onepixel;
            M10_cylinder=M10_cylinder+(j^1)*onepixel;
            M11_cylinder=M11_cylinder+(i*j)*onepixel;
            M20_cylinder=M20_cylinder+(j^2)*onepixel;
            M02_cylinder=M02_cylinder+(i^2)*onepixel;
        end
    end
end
```

5

5. The moment is calculated next. Which is necessary to allow us to find the centre of the shape and thus which pixel coordinate the object is in in the image.

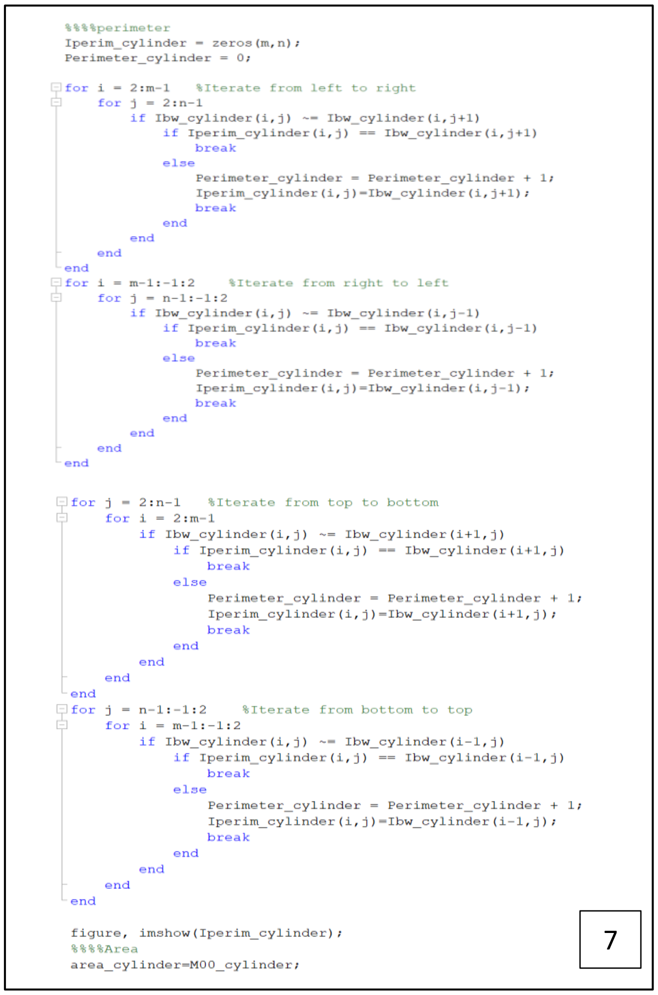
The p-qth moments of the image is calculated by the sum of the 'i' location to the power p or 'j' location to the power q, multiplied by the intensity of the pixel (in this case 1) at that i,j location



```
%%%%%Centroid
Xc_cylinder=M10_cylinder/M00_cylinder;
Yc_cylinder=M01_cylinder/M00_cylinder;
Centroid_cylinder_XY=[Xc_cylinder,Yc_cylinder];

%%%%%inertia matrix
u00_cylinder=M00_cylinder;
u01=0;
u10=0;
u11_cylinder=M11_cylinder-(Xc_cylinder*M01_cylinder);
u20_cylinder=M20_cylinder-(Xc_cylinder*M10_cylinder);
u02_cylinder=M02_cylinder-(Yc_cylinder*M01_cylinder);
J_cylinder=[u20_cylinder u11_cylinder;u11_cylinder u02_cylinder];

%eigenvalues and vectors
[v_cylinder,d_cylinder]=eig(J_cylinder); %v is eigenvector [2x2]
                                          %d is eigenvalue [2x2]
lamda1_cylinder=max(d_cylinder(1,1),d_cylinder(2,2)); %take first index of eigenvalue matrix
lamda2_cylinder=min(d_cylinder(1,1),d_cylinder(2,2)); %take first index of eigenvalue matrix

%%%%Major minor axis calculation
Majoraxis_cylinder=2*sqrt(lamda1_cylinder/M00_cylinder);
Minoraxis_cylinder=2*sqrt(lamda2_cylinder/M00_cylinder);

if lamda1_cylinder == d_cylinder(2,2)    %choose left or right eigen vector corresponding to lamda1
    vx_cylinder=v_cylinder(1,2);
    vy_cylinder=v_cylinder(2,2);
else
    vx_cylinder=v_cylinder(1,1);
    vy_cylinder=v_cylinder(1,2);
end

%%%%Angle from horizontal
Anglefromhorizontal_cylinder=atand(vy_cylinder/vx_cylinder);
```

6

6. Here, we calculate a few important properties of the shape such as centroid, major/minor axis of shape, and the orientation of the object.
This is essentially done by inputting values into the formulas derived above.



```
%%%%perimeter
Iperim_cylinder = zeros(m,n);
Perimeter_cylinder = 0;

for i = 2:m-1    %Iterate from left to right
    for j = 2:n-1
        if Ibw_cylinder(i,j) ~= Ibw_cylinder(i,j+1)
            if Iperim_cylinder(i,j) == Ibw_cylinder(i,j+1)
                break
            else
                Perimeter_cylinder = Perimeter_cylinder + 1;
                Iperim_cylinder(i,j)=Ibw_cylinder(i,j+1);
                break
            end
        end
    end
end
for i = m-1:-1:2    %Iterate from right to left
    for j = n-1:-1:2
        if Ibw_cylinder(i,j) ~= Ibw_cylinder(i,j-1)
            if Iperim_cylinder(i,j) == Ibw_cylinder(i,j-1)
                break
            else
                Perimeter_cylinder = Perimeter_cylinder + 1;
                Iperim_cylinder(i,j)=Ibw_cylinder(i,j-1);
                break
            end
        end
    end
end

for j = 2:n-1    %Iterate from top to bottom
    for i = 2:m-1
        if Ibw_cylinder(i,j) ~= Ibw_cylinder(i+1,j)
            if Iperim_cylinder(i,j) == Ibw_cylinder(i+1,j)
                break
            else
                Perimeter_cylinder = Perimeter_cylinder + 1;
                Iperim_cylinder(i,j)=Ibw_cylinder(i+1,j);
                break
            end
        end
    end
end
for j = n-1:-1:2    %Iterate from bottom to top
    for i = m-1:-1:2
        if Ibw_cylinder(i,j) ~= Ibw_cylinder(i-1,j)
            if Iperim_cylinder(i,j) == Ibw_cylinder(i-1,j)
                break
            else
                Perimeter_cylinder = Perimeter_cylinder + 1;
                Iperim_cylinder(i,j)=Ibw_cylinder(i-1,j);
                break
            end
        end
    end
end

figure, imshow(Iperim_cylinder);
%%%%Area
area_cylinder=M00_cylinder;
```

7

7. Next we wanted to calculate the perimeter. In order to do this, we decided to iterate from left to right, right to left, top to bottom and bottom to top, see figure 2.1 for details. If the pixel after the current was different, then it would add +1 to the counter, then start again on the next row. In order to not count the same pixels, if a pixel found in a separate iteration, then the line is skipped.
I then added each counted pixel to a new blank grid so that the perimeter could be plot and visualised (figure 2.0).

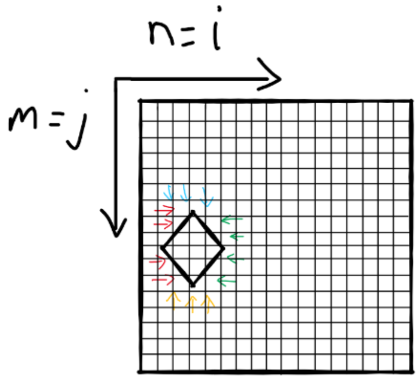
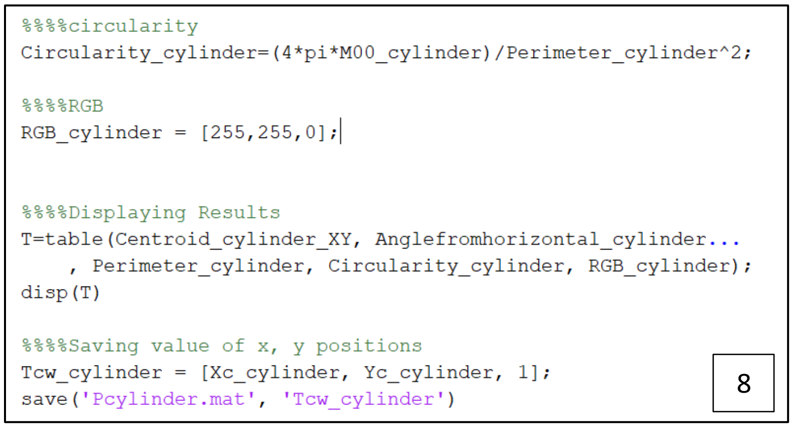


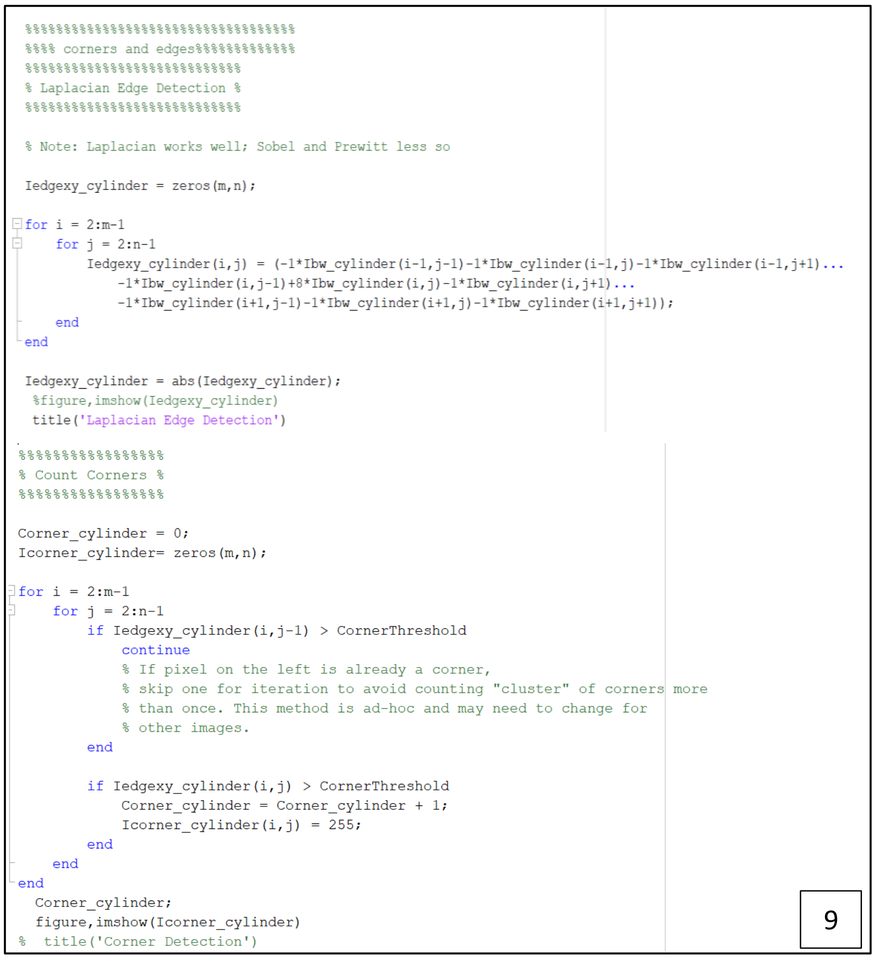
Figure 2.1: Diagram showing direction of iteration



```
%%%%circularity
Circularity_cylinder=(4*pi*M00_cylinder)/Perimeter_cylinder^2;

%%%%RGB
RGB_cylinder = [255,255,0];


%%%%Displaying Results
T=table(Centroid_cylinder_XY, Anglefromhorizontal_cylinder...
    , Perimeter_cylinder, Circularity_cylinder, RGB_cylinder);
disp(T)

%%%%Saving value of x, y positions
Tcw_cylinder = [Xc_cylinder, Yc_cylinder, 1];
save('Pcylinder.mat', 'Tcw_cylinder')
```

8

8. Lastly we calculate and show the circularity, RGB values, displayed in a table, see table 1.0, 1.1

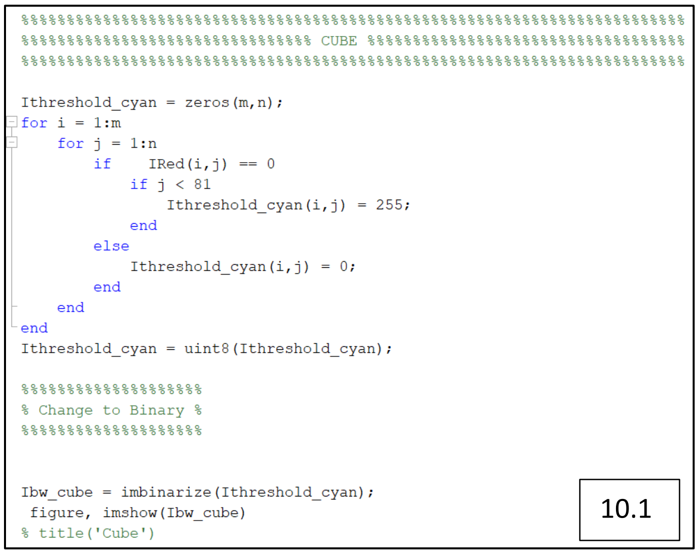


```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% corners and edges%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Laplacian Edge Detection %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Note: Laplacian works well; Sobel and Prewitt less so

Iedgexy_cylinder = zeros(m,n);

for i = 2:m-1
    for j = 2:n-1
        Iedgexy_cylinder(i,j) = (-1*Ibw_cylinder(i-1,j-1)-1*Ibw_cylinder(i-1,j)-1*Ibw_cylinder(i-1,j+1)...
            -1*Ibw_cylinder(i,j-1)+8*Ibw_cylinder(i,j)-1*Ibw_cylinder(i,j+1)...
            -1*Ibw_cylinder(i+1,j-1)-1*Ibw_cylinder(i+1,j)-1*Ibw_cylinder(i+1,j+1));
    end
end

Iedgexy_cylinder = abs(Iedgexy_cylinder);
 %figure,imshow(Iedgexy_cylinder)
 title('Laplacian Edge Detection')

%%%%%%%%%%%%%%%%
% Count Corners %
%%%%%%%%%%%%%%%%

Corner_cylinder = 0;
Icorner_cylinder= zeros(m,n);

for i = 2:m-1
    for j = 2:n-1
        if Iedgexy_cylinder(i,j-1) > CornerThreshold
            continue
            % If pixel on the left is already a corner,
            % skip one for iteration to avoid counting "cluster" of corners more
            % than once. This method is ad-hoc and may need to change for
            % other images.
        end

        if Iedgexy_cylinder(i,j) > CornerThreshold
            Corner_cylinder = Corner_cylinder + 1;
            Icorner_cylinder(i,j) = 255;
        end
    end
end
  Corner_cylinder;
  figure,imshow(Icorner_cylinder)
%  title('Corner Detection')
```

9

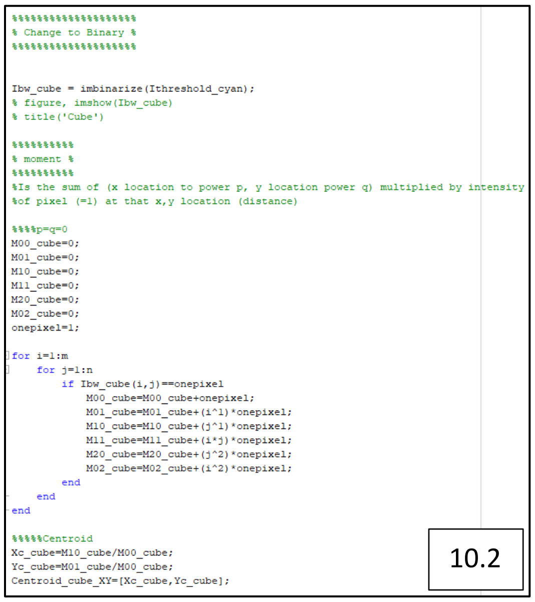
9. We also attempted to do corner and edge detection of the shape, using Laplacian edge detection and corner counting.

10. The exact same process to the cylinder was performed on the cube, however the only difference being, to detect the threshold, the pixels were limited to detecting the left side only.



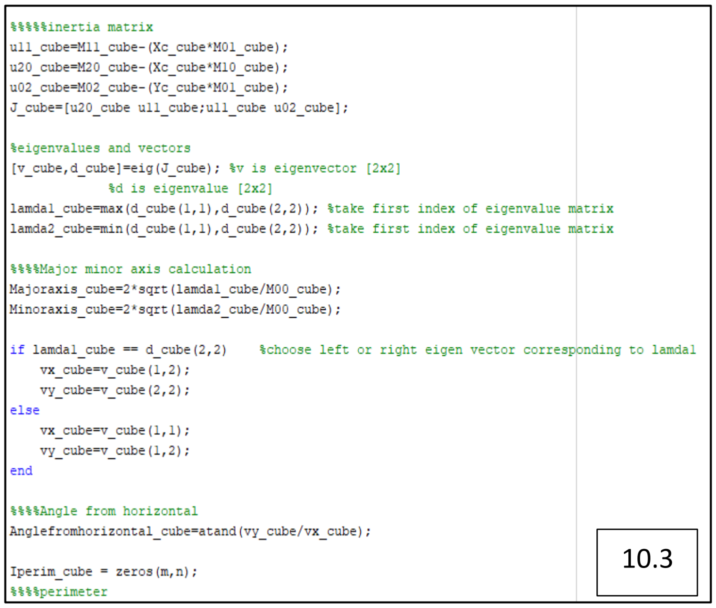
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CUBE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Ithreshold_cyan = zeros(m,n);
for i = 1:m
    for j = 1:n
        if    IRed(i,j) == 0
            if j < 81
                Ithreshold_cyan(i,j) = 255;
            end
        else
            Ithreshold_cyan(i,j) = 0;
        end
    end
end
Ithreshold_cyan = uint8(Ithreshold_cyan);

%%%%%%%%%%%%%%%%%%%%
% Change to Binary %
%%%%%%%%%%%%%%%%%%%%


Ibw_cube = imbinarize(Ithreshold_cyan);
 figure, imshow(Ibw_cube)
% title('Cube')
```

10.1



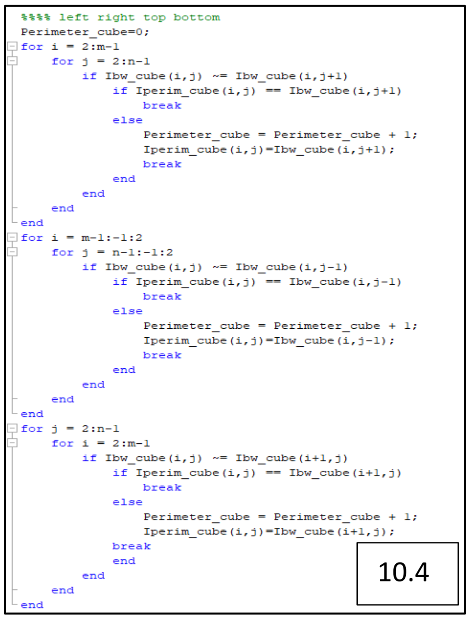
```
%%%%%%%%%%%%%%%%%%%%
% Change to Binary %
%%%%%%%%%%%%%%%%%%%%


Ibw_cube = imbinarize(Ithreshold_cyan);
% figure, imshow(Ibw_cube)
% title('Cube')

%%%%%%%%%%
% moment %
%%%%%%%%%%
%Is the sum of (x location to power p, y location power q) multiplied by intensity
%of pixel (=1) at that x,y location (distance)

%%%%p=q=0
M00_cube=0;
M01_cube=0;
M10_cube=0;
M11_cube=0;
M20_cube=0;
M02_cube=0;
onepixel=1;

for i=1:m
    for j=1:n
        if Ibw_cube(i,j)==onepixel
            M00_cube=M00_cube+onepixel;
            M01_cube=M01_cube+(i^1)*onepixel;
            M10_cube=M10_cube+(j^1)*onepixel;
            M11_cube=M11_cube+(i*j)*onepixel;
            M20_cube=M20_cube+(j^2)*onepixel;
            M02_cube=M02_cube+(i^2)*onepixel;
        end
    end
end

%%%%%Centroid
Xc_cube=M10_cube/M00_cube;
Yc_cube=M01_cube/M00_cube;
Centroid_cube_XY=[Xc_cube,Yc_cube];
```

10.2



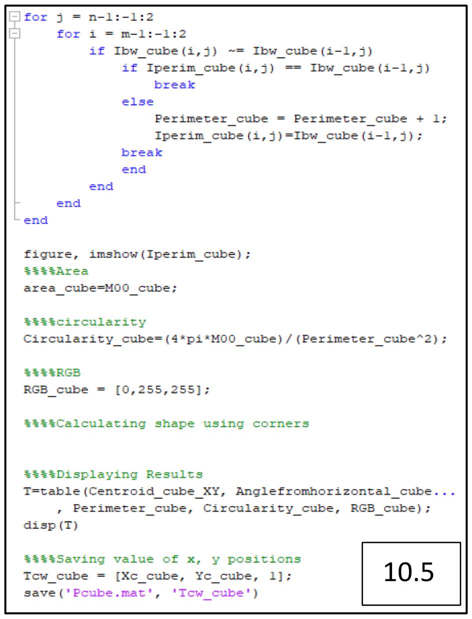
```
%%%%%inertia matrix
u11_cube=M11_cube-(Xc_cube*M01_cube);
u20_cube=M20_cube-(Xc_cube*M10_cube);
u02_cube=M02_cube-(Yc_cube*M01_cube);
J_cube=[u20_cube u11_cube;u11_cube u02_cube];

%eigenvalues and vectors
[v_cube,d_cube]=eig(J_cube); %v is eigenvector [2x2]
                             %d is eigenvalue [2x2]
lamda1_cube=max(d_cube(1,1),d_cube(2,2)); %take first index of eigenvalue matrix
lamda2_cube=min(d_cube(1,1),d_cube(2,2)); %take first index of eigenvalue matrix

%%%%Major minor axis calculation
Majoraxis_cube=2*sqrt(lamda1_cube/M00_cube);
Minoraxis_cube=2*sqrt(lamda2_cube/M00_cube);

if lamda1_cube == d_cube(2,2)    %choose left or right eigen vector corresponding to lamda1
    vx_cube=v_cube(1,2);
    vy_cube=v_cube(2,2);
else
    vx_cube=v_cube(1,1);
    vy_cube=v_cube(1,2);
end

%%%%Angle from horizontal
Anglefromhorizontal_cube=atand(vy_cube/vx_cube);

Iperim_cube = zeros(m,n);
%%%%perimeter
```

10.3



```
%%%% left right top bottom
Perimeter_cube=0;
for i = 2:m-1
    for j = 2:n-1
        if Ibw_cube(i,j) ~= Ibw_cube(i,j+1)
            if Iperim_cube(i,j) == Ibw_cube(i,j+1)
                break
            else
                Perimeter_cube = Perimeter_cube + 1;
                Iperim_cube(i,j)=Ibw_cube(i,j+1);
                break
            end
        end
    end
end
for i = m-1:-1:2
    for j = n-1:-1:2
        if Ibw_cube(i,j) ~= Ibw_cube(i,j-1)
            if Iperim_cube(i,j) == Ibw_cube(i,j-1)
                break
            else
                Perimeter_cube = Perimeter_cube + 1;
                Iperim_cube(i,j)=Ibw_cube(i,j-1);
                break
            end
        end
    end
end
for j = 2:n-1
    for i = 2:m-1
        if Ibw_cube(i,j) ~= Ibw_cube(i+1,j)
            if Iperim_cube(i,j) == Ibw_cube(i+1,j)
                break
            else
                Perimeter_cube = Perimeter_cube + 1;
                Iperim_cube(i,j)=Ibw_cube(i+1,j);
            break
            end
        end
    end
end
```

10.4



```
for j = n-1:-1:2
    for i = m-1:-1:2
        if Ibw_cube(i,j) ~= Ibw_cube(i-1,j)
            if Iperim_cube(i,j) == Ibw_cube(i-1,j)
                break
            else
                Perimeter_cube = Perimeter_cube + 1;
                Iperim_cube(i,j)=Ibw_cube(i-1,j);
            break
            end
        end
    end
end

figure, imshow(Iperim_cube);
%%%%Area
area_cube=M00_cube;

%%%%circularity
Circularity_cube=(4*pi*M00_cube)/(Perimeter_cube^2);

%%%%RGB
RGB_cube = [0,255,255];

%%%%Calculating shape using corners


%%%%Displaying Results
T=table(Centroid_cube_XY, Anglefromhorizontal_cube...
    , Perimeter_cube, Circularity_cube, RGB_cube);
disp(T)

%%%%Saving value of x, y positions
Tcw_cube = [Xc_cube, Yc_cube, 1];
save('Pcube.mat', 'Tcw_cube')
```

10.5

<u>**Results - Cylinder**</u>

As can be seen from figure 2.0, we can see that without filter, there is a black dot in the centre of the circle, whereas, with filter the dot was filtered out, however the circular area was visibly smaller. In addition, the table of properties for each cylinder is also shown below (table 1.0, 1.1). An interesting observation is that the circularity of the cylinder without filter is 'better' than the circularity of the cylinder with the filter as an ideal circularity of a circle is 1. The reason this could be is because the cylinder is too small so that the ratio between area and perimeter isn't significant enough, therefore messing up the calculation.

Other than this, the results seemed to be correct, such as the centroid coordinates that match with the MATLAB 'data tips' function (Figure 2.0 right).

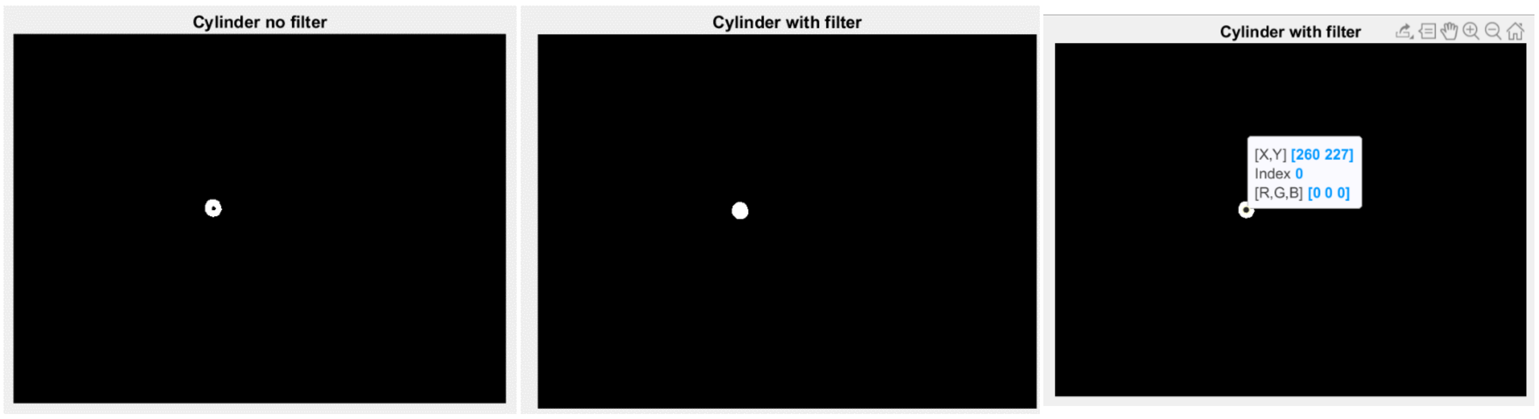


Figure 2.0 - output after, thresholding, filtering and binarizing the image. Also used MATLAB data tips function to find center of cylinder coordinates (Right)

```
Centroid_cylinder_XY    Anglefromhorizontal_cylinder    Perimeter_cylinder    Circularity_cylinder        RGB_cylinder
____________________    ____________________________    __________________    ____________________    ________________

  260.02    226.76                 75.05                        67                   1.0442           255    255    0
```

Table 1.0 – properties of cylinder with no filter

```
Centroid_cylinder_XY    Anglefromhorizontal_cylinder    Perimeter_cylinder    Circularity_cylinder      RGB_cylinder
____________________    ____________________________    __________________    ____________________    ______________

  260.05    226.72                 73.945                       63                   1.1778           255    255    0
```

Table 1.1 – properties of cylinder with filter

Below, shows figures 2.1 of the perimeter, edge detection and corner detection. The perimeter method of counting pixels seemed to work well because on comparison, the counted pixels seemed to form an exact perimeter of the cylinder. In corner and edge detection, the results also seemed to work well.

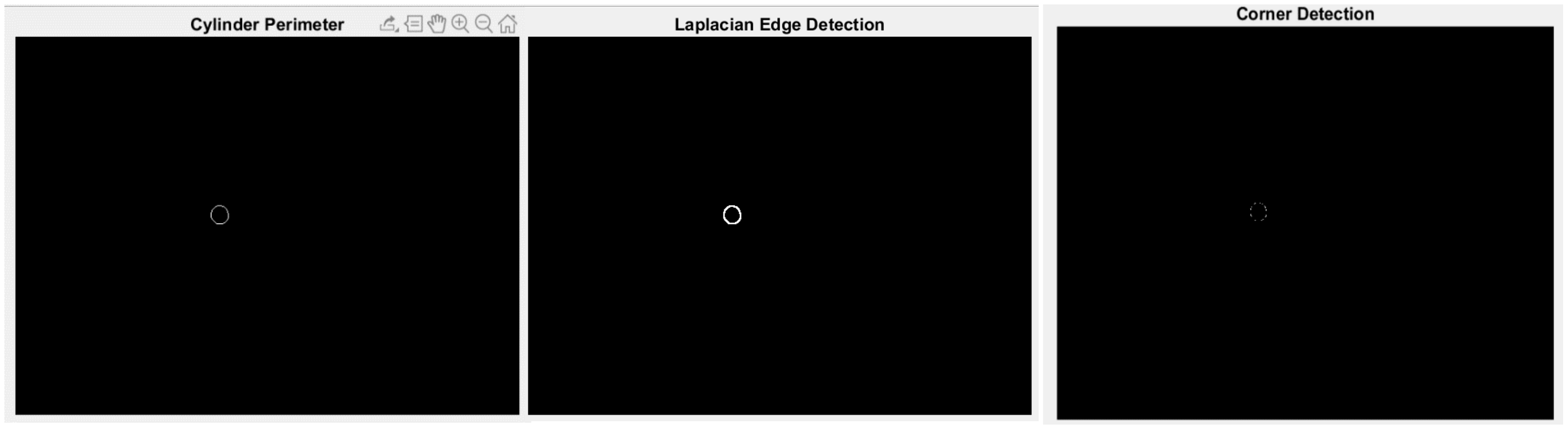


Figure 2.1 – Cylinder perimeter, Laplacian edge detection and corner detection algorithms

Results - Cube

Because the cube did not require filtering, the binarized figure showed the exact shape of the cube which made the process of calculating the perimeter and its properties much easier and accurate.

However, once again the circularity of the cube (table 1.2) seemed to be incorrect as the value we should be expecting is around π/4, but instead we get 0.99391 which is what the circle should get. Again, this is likely due to the cube being too small so that the ratio between area and perimeter isn't significant enough.

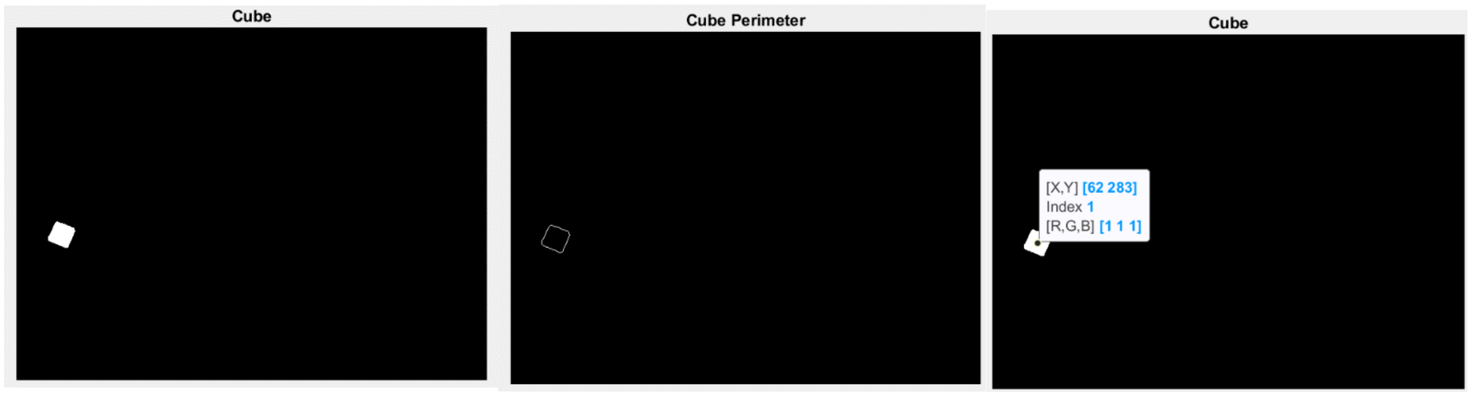
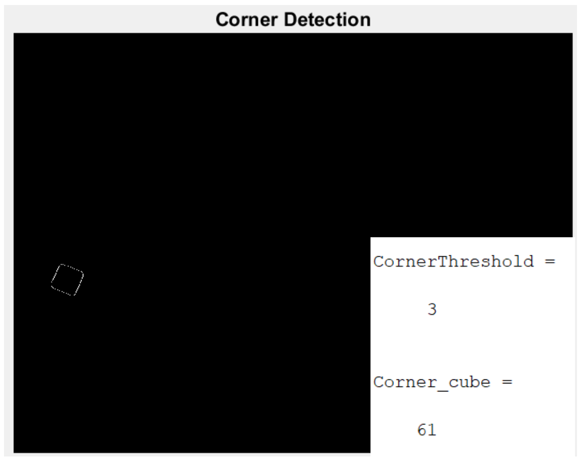


Figure 2.2 - output after, thresholding, filtering and binarizing the image. Also used MATLAB data tips function to find center of cube coordinates (Right)

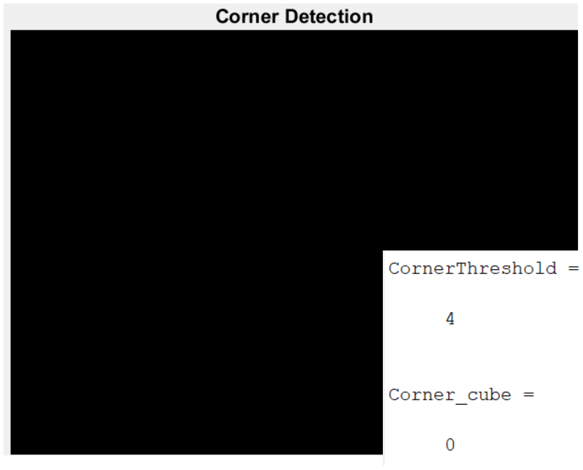
```
Centroid_cube_XY    Anglefromhorizontal_cube    Perimeter_cube    Circularity_cube        RGB_cube
________________    ________________________    ______________    ________________    ________________

61.999    282.54             -10.174                 105              0.99391         0     255     255
```

Table 1.2 – properties of cube

Corner detection was attempted with the cube however, as can be seen in figure 2.3 there is a significant change in the number of corners detected when the corner threshold differs between 3 or 4 pixels. This is likely due to the size of the shape and therefore this result was ignored.



```
CornerThreshold =

     3

Corner_cube =

    61
```



```
CornerThreshold =

     4

Corner_cube =

     0
```

Figure 2.3 – Corner algorithm for cube with corner threshold of 3 (left) and 4 (right)

Task 4 – Relating pixel position to robot position using simulated data

Method and Derivation

After finding the 2D position of the image in the pixel frame (x,y) from task 3, we can now find the relationship between the pixel coordinate and the world coordinate (X,Y). Then using all the parameters we calculated previously we can find the relationship between camera and world coordinate to get 3D position of the object (X_r, Y_r, Z_r).

This can be expressed mathematically as shown:

$$Relationship\ between\ pixel\ coordinate\ and\ world\ coordinate:$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \sim K[r_1 \quad r_2 \quad CP_{Worg}] \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = H \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

$$Where\ H\ is\ the\ extrinsic\ parameter\ and\ is\ the\ product\ of\ intrinisic\ parameter\ and$$

$$camera\ to\ world\ extrinisic\ parameters.$$

$$From\ this\ we\ can\ obtain\ the\ world\ coordinates:$$

$$\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \sim H^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$Camera\ to\ world\ coordinates\ relationship\ can\ be\ expressed\ as:$$

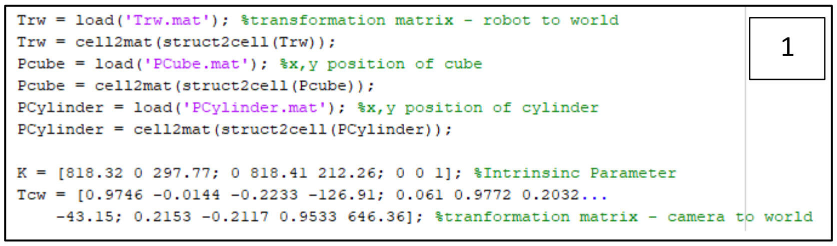
$$\begin{bmatrix} X_r \\ Y_r \\ Z_r \\ 1 \end{bmatrix} = \begin{bmatrix} \rho_{11} & \rho_{12} & \rho_{13} & t_x \\ \rho_{21} & \rho_{22} & \rho_{23} & t_y \\ \rho_{31} & \rho_{32} & \rho_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix}$$

$$Substituing\ in\ X, Y\ values\ from\ the\ world\ coordinates\ above, we\ can\ finally\ obtain$$

$$the\ objects\ coordinates\ in\ the\ robot\ frame\ X_r, Y_r, Z_r\ and\ the\ robot\ can\ accurately\ find$$

$$the\ position\ of\ the\ object\ based\ off\ camera\ images.$$

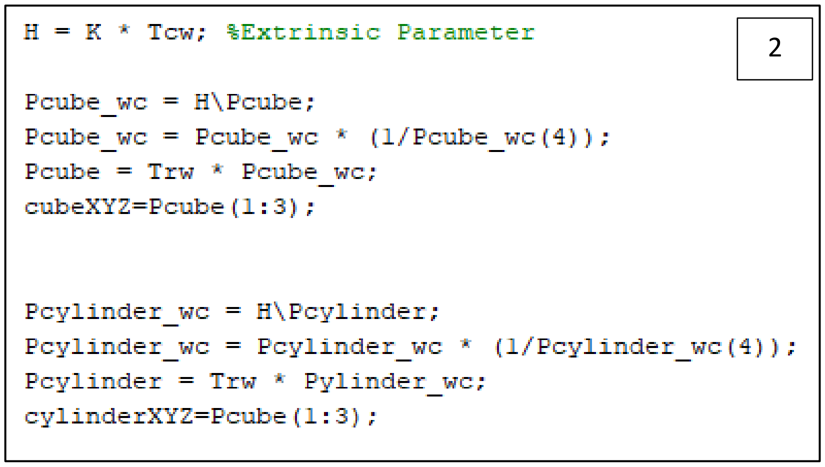
MATLAB code



```
Trw = load('Trw.mat'); %transformation matrix - robot to world
Trw = cell2mat(struct2cell(Trw));
Pcube = load('PCube.mat'); %x,y position of cube
Pcube = cell2mat(struct2cell(Pcube));
PCylinder = load('PCylinder.mat'); %x,y position of cylinder
PCylinder = cell2mat(struct2cell(PCylinder));

K = [818.32 0 297.77; 0 818.41 212.26; 0 0 1]; %Intrinsinc Parameter
Tcw = [0.9746 -0.0144 -0.2233 -126.91; 0.061 0.9772 0.2032...
    -43.15; 0.2153 -0.2117 0.9533 646.36]; %tranformation matrix - camera to world
```

1

1. Firstly we load in the matrices calculated previously as well as redefine K and Tcw parameters.



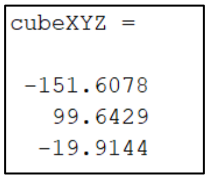
```
H = K * Tcw; %Extrinsic Parameter

Pcube_wc = H\Pcube;
Pcube_wc = Pcube_wc * (1/Pcube_wc(4));
Pcube = Trw * Pcube_wc;
cubeXYZ=Pcube(1:3);


Pcylinder_wc = H\Pcylinder;
Pcylinder_wc = Pcylinder_wc * (1/Pcylinder_wc(4));
Pcylinder = Trw * Pylinder_wc;
cylinderXYZ=Pcube(1:3);
```

2

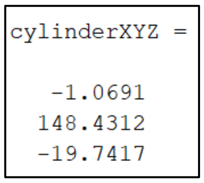
2. Then after finding the extrinsic parameter H, we can calculate the object in real 3D space. We do this by finding the world coordinate which is the inverse of H*Pixel coordinate of cube/cylinder. The last row is then normalised.
Finally to calculate the position of the object in the robot frame, the transformation matrix (imported before), is multiplied by the world coordinates to get the coordinates of the object in the robot frame and we just take the first 3 rows to remove the '1'.

Results

The results we get from this calculation is:



```
cubeXYZ =

 -151.6078
   99.6429
  -19.9144
```



```
cylinderXYZ =

  -1.0691
 148.4312
 -19.7417
```

Which makes sense as the Z coordinates for both cube and cylinder are on the same plane (remains around -20), and the X,Y coordinates match proportionally to the 2D image we were given.

Task 5 - Camera Calibration Experiment

Method and Derivation

First, we measured the length of a checkboard square which turned out to be 21mm(figure 3.0), we then went ahead and took 6 images of the checkerboard with different orientations and positions using MATLAB code provided to us, with an additional 7th image of the checkerboard in the intended working position. After having all the data needed, we went ahead and adapted the code from task one to our conditions.

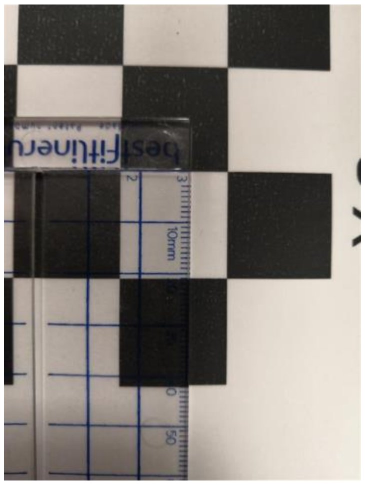


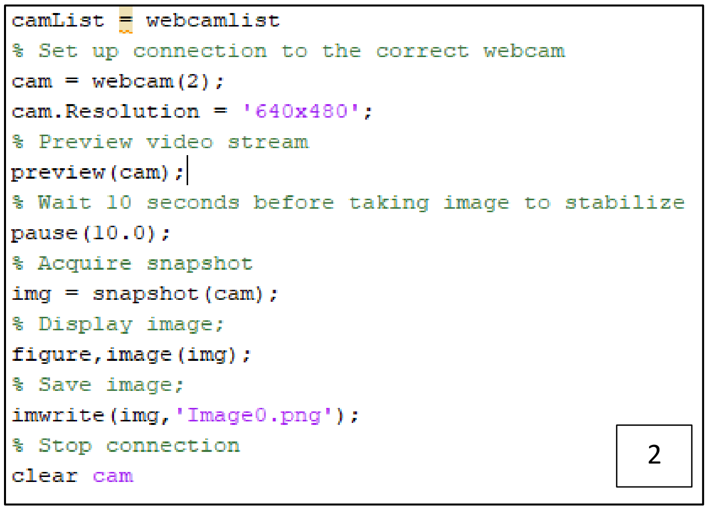
Figure 3.0 – Measurement of the checkerboard

MATLAB code

```
% General setting
numImages = 6;

% Identify available webcam
camList = webcamlist

% Set up connection to the correct webcam
% You may need to change the number in the brackets
cam = webcam(2);
cam.Resolution = '640x480';

% Preview video stream
preview(cam);

% Repeat numImages times
for idx = 1:numImages
    % Wait 10 seconds before taking image to stabilize
    pause(10.0);

    % Acquire snapshot
    img(:,:,:,idx) = snapshot(cam);

    % Display image;
    figure,image(img(:,:,:,idx));

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

% Stop connection
clear cam
```

1

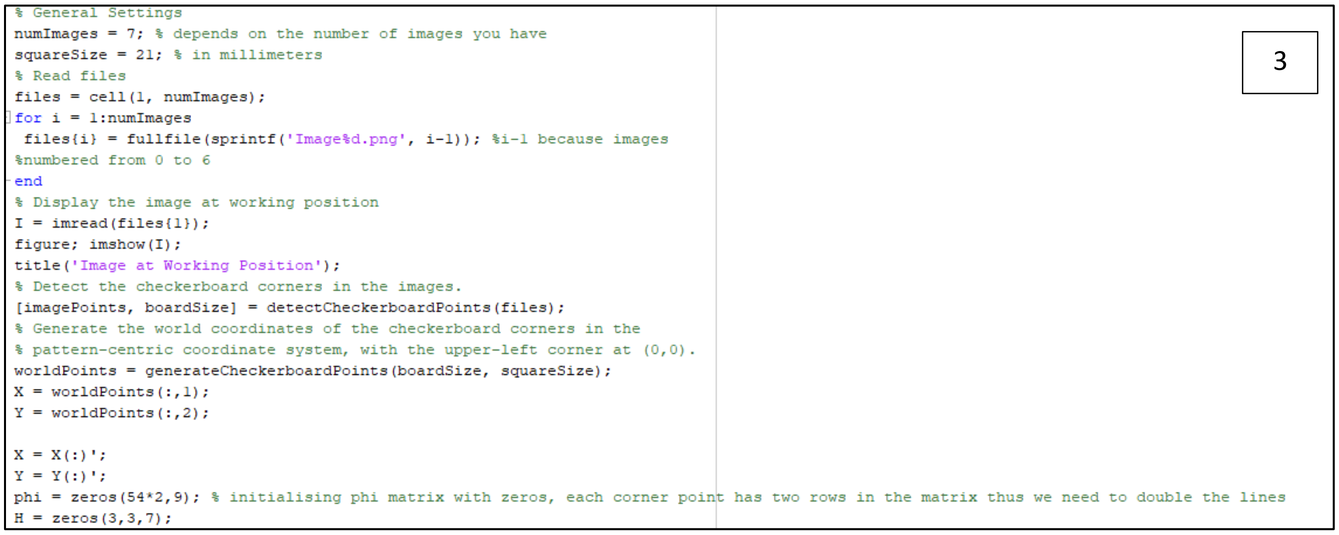
1. We firstly take 6 different images of the checkerboard for the camera calibration.



```
camList = webcamlist
% Set up connection to the correct webcam
cam = webcam(2);
cam.Resolution = '640x480';
% Preview video stream
preview(cam);
% Wait 10 seconds before taking image to stabilize
pause(10.0);
% Acquire snapshot
img = snapshot(cam);
% Display image;
figure,image(img);
% Save image;
imwrite(img,'Image0.png');
% Stop connection
clear cam
```

2

2. After that we take another image with the checkerboard in a fixed position, as this is the intended working position.



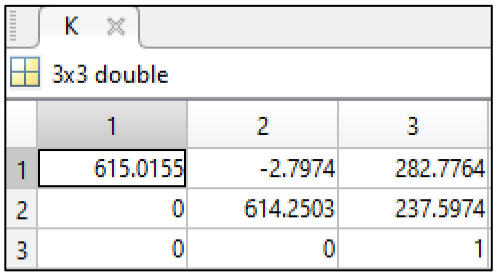
```
% General Settings
numImages = 7; % depends on the number of images you have
squareSize = 21; % in millimeters
% Read files
files = cell(1, numImages);
for i = 1:numImages
 files{i} = fullfile(sprintf('Image%d.png', i-1)); %i-1 because images
%numbered from 0 to 6
end
% Display the image at working position
I = imread(files{1});
figure; imshow(I);
title('Image at Working Position');
% Detect the checkerboard corners in the images.
[imagePoints, boardSize] = detectCheckerboardPoints(files);
% Generate the world coordinates of the checkerboard corners in the
% pattern-centric coordinate system, with the upper-left corner at (0,0).
worldPoints = generateCheckerboardPoints(boardSize, squareSize);
X = worldPoints(:,1);
Y = worldPoints(:,2);

X = X(:)';
Y = Y(:)';
phi = zeros(54*2,9); % initialising phi matrix with zeros, each corner point has two rows in the matrix thus we need to double the lines
H = zeros(3,3,7);
```

3

3. After that we extract the checkerboard points and reuse the code used in task 1. The only changes made are the number of images, and the size of matrix phi now becomes “phi(54*2, 9)” while H becomes H(3,3,7) (there are 7 images).

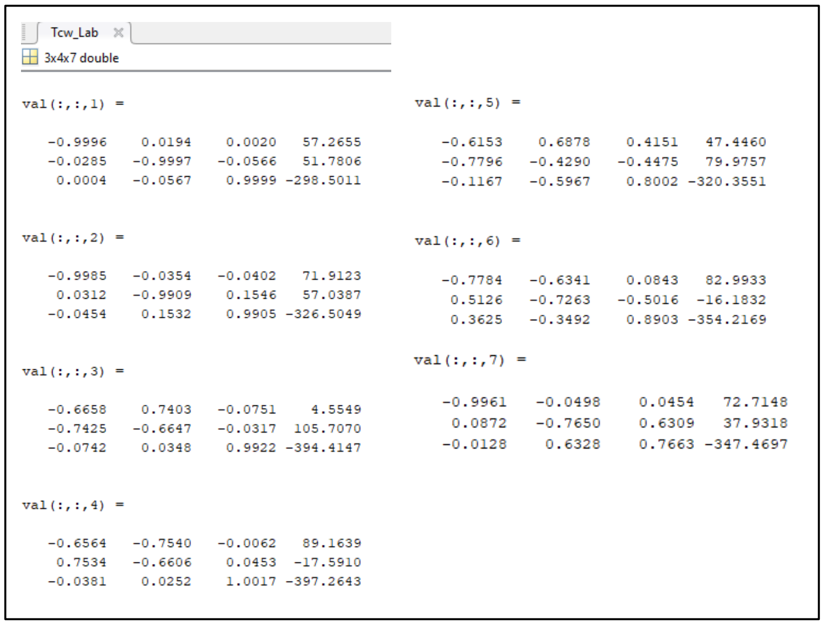
Results

For our 7 images we then got the following results for K, and Tcw which we save as a .mat file to be used later in task8. There are 7 Tcw_lab matrices because of the 7 images however we only take the first one which corresponds to the first working image.


K

3x3 double

	1	2	3
1	615.0155	-2.7974	282.7764
2	0	614.2503	237.5974
3	0	0	1


Tcw_Lab

3x4x7 double

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

   -0.9996    0.0194    0.0020   57.2655
   -0.0285   -0.9997   -0.0566   51.7806
    0.0004   -0.0567    0.9999 -298.5011


val(:,:,2) =

   -0.9985   -0.0354   -0.0402   71.9123
    0.0312   -0.9909    0.1546   57.0387
   -0.0454    0.1532    0.9905 -326.5049


val(:,:,3) =

   -0.6658    0.7403   -0.0751    4.5549
   -0.7425   -0.6647   -0.0317  105.7070
   -0.0742    0.0348    0.9922 -394.4147


val(:,:,4) =

   -0.6564   -0.7540   -0.0062   89.1639
    0.7534   -0.6606    0.0453  -17.5910
   -0.0381    0.0252    1.0017 -397.2643


val(:,:,5) =

   -0.6153    0.6878    0.4151   47.4460
   -0.7796   -0.4290   -0.4475   79.9757
   -0.1167   -0.5967    0.8002 -320.3551


val(:,:,6) =

   -0.7784   -0.6341    0.0843   82.9933
    0.5126   -0.7263   -0.5016  -16.1832
    0.3625   -0.3492    0.8903 -354.2169

val(:,:,7) =

   -0.9961   -0.0498    0.0454   72.7148
    0.0872   -0.7650    0.6309   37.9318
   -0.0128    0.6328    0.7663 -347.4697
```

Task 6 – Robot calibration experiment

Method and Derivation

Task 6 was the robot calibration. For this we used the inverse kinematics code from the last report to control the end-effector using potentiometers to move it to certain locations on the checkerboard and touching the corners with a toothpick grabbed by the end-effector and measure the coordinates of the point P with respect to frame {3} shown in figure 4.0. As the robot moved to each location, the serial monitor on the Arduino was turned on so we could take readings of the angles for each joint of the robot $\theta_{1,2,3}$.

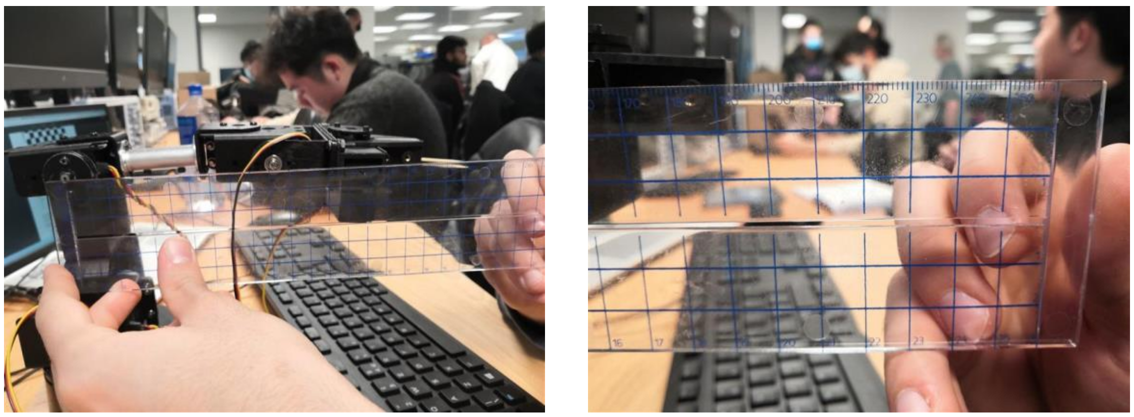
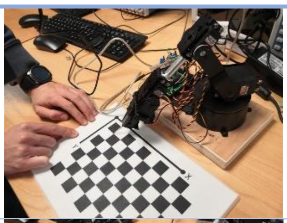
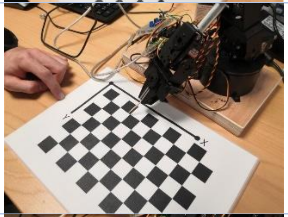
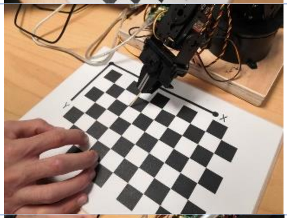
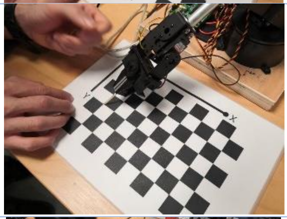
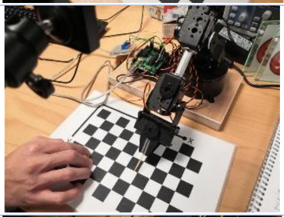
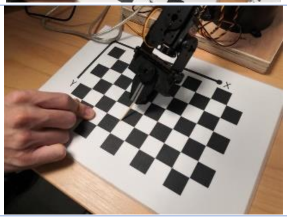


Figure 4.0 – Image captured when measuring length from frame {3} to point P

Then using forward kinematics code from last report, we substituted in the each angle to find the cartesian positions of the tool tip with respect to the robot frame X_{ri}, Y_{ri}, Z_{ri}.

Finally we can substitute in all the both the world and robot coordinates into the MATLAB code which does the calculation shown in task 2, so that we will get a single homogenous transformation matrix for the position and orientation of the world frame wrt to the robot frame.

Point	**Xi**	**Yi**	**θ_{1i}**	**θ_{2i}**	**θ_{3i}**	**Xri**	**Yri**	**Zri**	**Picture**
1	0	0	126.41	62.98	1015.01	-11.99	16.25	-6.11	
2	42	0	115.04	70.50	118.33	-7.42	15.89	-7.17	
3	42	21	113.62	62.19	103.97	-8.18	18.66	-6.10	
4	0	63	116.34	40.41	69.24	-11.50	23.23	-4.34	
5	63	84	100.96	41.6	70.58	-4.90	25.29	-4.24	
6	105	63	92.00	54.03	89.66	-0.8	22.92	-5.00	

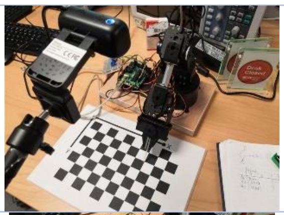
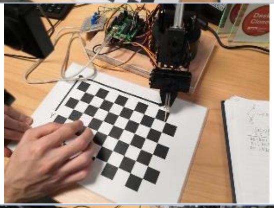
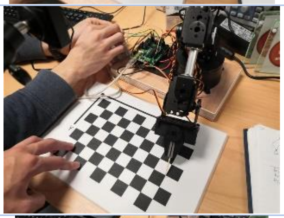
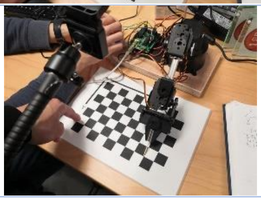
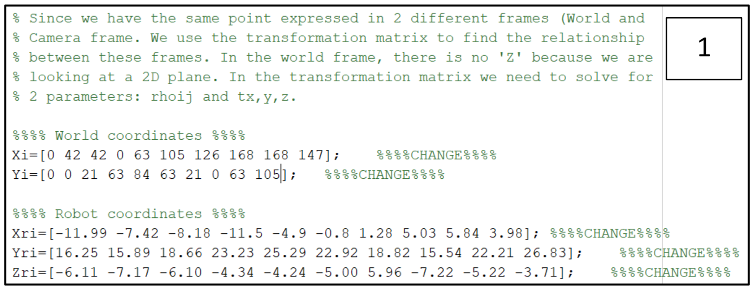
7	126	21	86.12	70.03	113.17	1.28	18.82	-5.96	
8	168	0	72.08	75.99	125.06	5.03	15.54	-7.22	
9	168	63	75.27	53.07	89.13	5.84	22.21	-5.22	
10	147	105	81.56	34.98	59.86	3.98	26.83	-3.71	

Table 2.0 – Summary of all coordinates and corresponding photo taken during lab

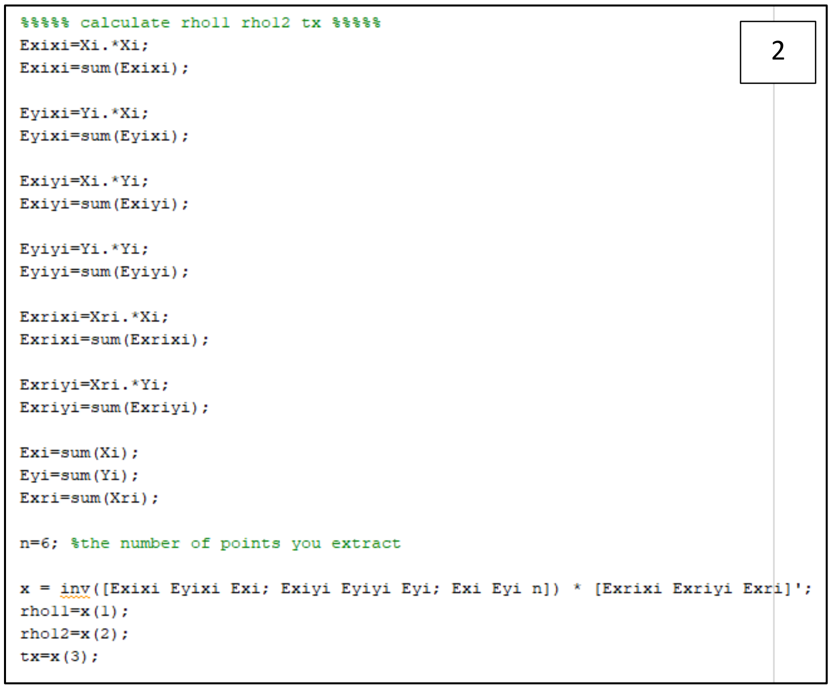
Arduino and MATLAB code



```
% Since we have the same point expressed in 2 different frames (World and
% Camera frame. We use the transformation matrix to find the relationship
% between these frames. In the world frame, there is no 'Z' because we are
% looking at a 2D plane. In the transformation matrix we need to solve for
% 2 parameters: rhoij and tx,y,z.

%%%% World coordinates %%%%
Xi=[0 42 42 0 63 105 126 168 168 147];    %%%%CHANGE%%%%
Yi=[0 0 21 63 84 63 21 0 63 105];   %%%%CHANGE%%%%

%%%% Robot coordinates %%%%
Xri=[-11.99 -7.42 -8.18 -11.5 -4.9 -0.8 1.28 5.03 5.84 3.98]; %%%%CHANGE%%%%
Yri=[16.25 15.89 18.66 23.23 25.29 22.92 18.82 15.54 22.21 26.83];    %%%%CHANGE%%%%
Zri=[-6.11 -7.17 -6.10 -4.34 -4.24 -5.00 5.96 -7.22 -5.22 -3.71];    %%%%CHANGE%%%%
```

1

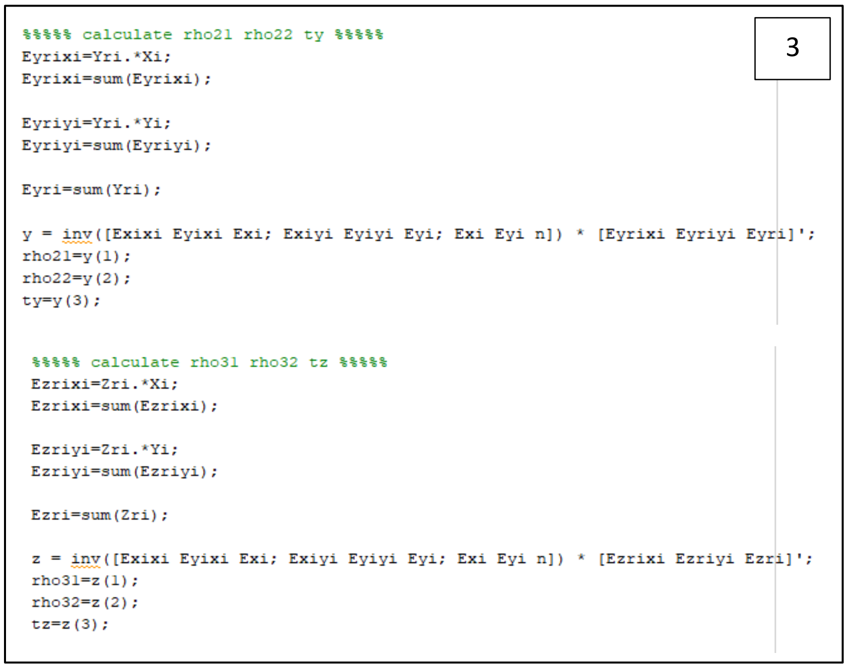
1. Firstly, the world coordinate (corners of checkerboard) and robot coordinates (calculated from forward kinematics) are input into its respective vectors.



```
%%%%% calculate rho11 rho12 tx %%%%%
Exixi=Xi.*Xi;
Exixi=sum(Exixi);

Eyixi=Yi.*Xi;
Eyixi=sum(Eyixi);

Exiyi=Xi.*Yi;
Exiyi=sum(Exiyi);

Eyiyi=Yi.*Yi;
Eyiyi=sum(Eyiyi);

Exrixi=Xri.*Xi;
Exrixi=sum(Exrixi);

Exriyi=Xri.*Yi;
Exriyi=sum(Exriyi);

Exi=sum(Xi);
Eyi=sum(Yi);
Exri=sum(Xri);

n=6; %the number of points you extract

x = inv([Exixi Eyixi Exi; Exiyi Eyiyi Eyi; Exi Eyi n]) * [Exrixi Exriyi Exri]';
rho11=x(1);
rho12=x(2);
tx=x(3);
```

2

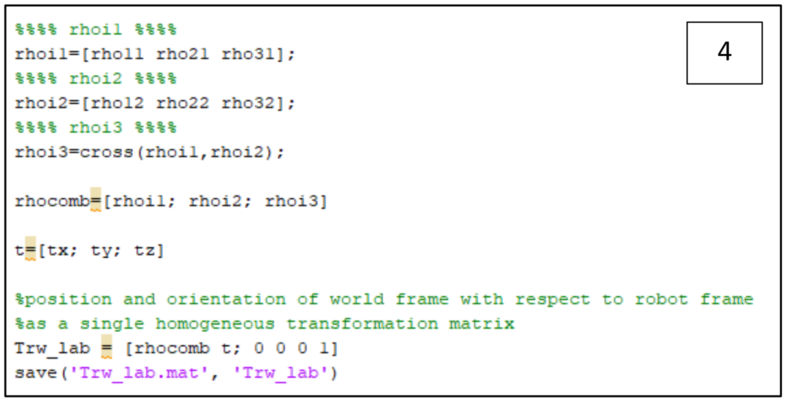
2. Then we wrote out the variables for ρ_{11}, ρ_{12}, t_x and that were to go into the matrix as well as extracting the results from the matrix.



```
%%%%% calculate rho21 rho22 ty %%%%%
Eyrixi=Yri.*Xi;
Eyrixi=sum(Eyrixi);

Eyriyi=Yri.*Yi;
Eyriyi=sum(Eyriyi);

Eyri=sum(Yri);

y = inv([Exixi Eyixi Exi; Exiyi Eyiyi Eyi; Exi Eyi n]) * [Eyrixi Eyriyi Eyri]';
rho21=y(1);
rho22=y(2);
ty=y(3);


 %%%%% calculate rho31 rho32 tz %%%%%
 Ezrixi=Zri.*Xi;
 Ezrixi=sum(Ezrixi);

 Ezriyi=Zri.*Yi;
 Ezriyi=sum(Ezriyi);

 Ezri=sum(Zri);

 z = inv([Exixi Eyixi Exi; Exiyi Eyiyi Eyi; Exi Eyi n]) * [Ezrixi Ezriyi Ezri]';
 rho31=z(1);
 rho32=z(2);
 tz=z(3);
```

3

3. And like task 2, the variables were changed for ρ_{21}, ρ_{22}, t_y and as well as ρ_{31}, ρ_{32}, t_z



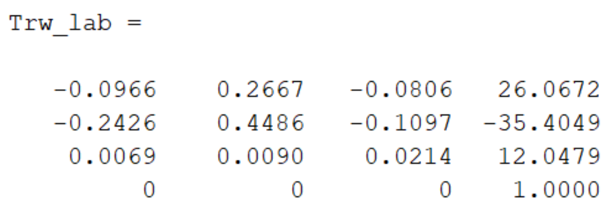
```
%%%% rhoi1 %%%%
rhoi1=[rho11 rho21 rho31];
%%%% rhoi2 %%%%
rhoi2=[rho12 rho22 rho32];
%%%% rhoi3 %%%%
rhoi3=cross(rhoi1,rhoi2);

rhocomb=[rhoi1; rhoi2; rhoi3]

t=[tx; ty; tz]

%position and orientation of world frame with respect to robot frame
%as a single homogeneous transformation matrix
Trw_lab = [rhocomb t; 0 0 0 1]
save('Trw_lab.mat', 'Trw_lab')
```

4

4. Finally the values calculated are combined to form the a homogenous transformation matrix Trw_lab.mat.

Results

After completing the full procedure, we get the a homogenous transformation matrix which is demonstrates the robot-to-world relationship figure 4.1 Which we will use in task 8 to relate pixel position to robot position:



```
Trw_lab =

   -0.0966    0.2667   -0.0806   26.0672
   -0.2426    0.4486   -0.1097  -35.4049
    0.0069    0.0090    0.0214   12.0479
         0         0         0    1.0000
```

Figure 4.1 – Final matrix for robot-to-world relationship

Task 7 – Image processing experiment

Method and Derivation

After task 6, the robot is now calibrated and we can remove the checkerboard. We use exactly the same code as task 5 to take photo at the working place but change only the image name into object1. After doing this, we get a three-dimensional RGB image. What we now need to do is to use image processing method to determine the shape, position and orientation of the object.

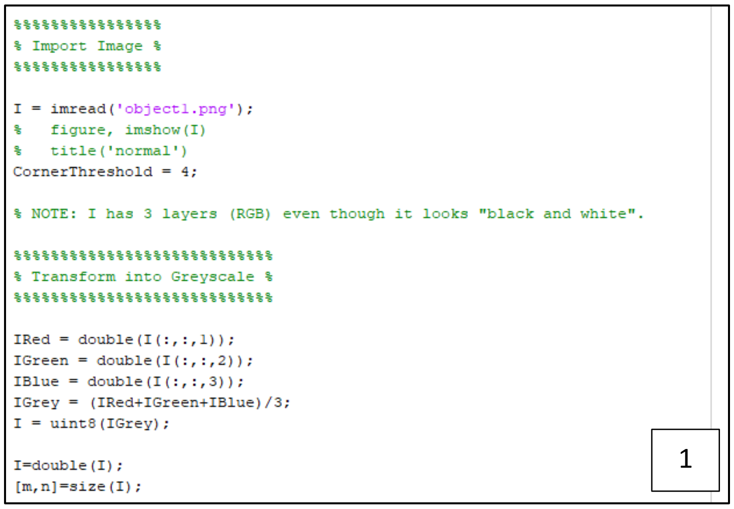
First, we turn the original RGB image into grey scale image. Doing this is because, firstly, the original image is three-dimensional image, and it is complicated to do the further image processing. Grey scale image makes the implementation of filter and threshold become much easier. Secondly, problem might occur as we use the RGB image to determine the edge. This is because the edge detection is based on the detection of changing gradient. However, at the same pixel point, the gradient for different colour might be different as well (because the implement of the detection carries out individually for red, blue and green), that means the edge are different as well. Thus, error might occur.

After doing the grey scale image, we use threshold method to convert the image into black and white image. Instead of using the histogram to determine the threshold, we use the RGB finder to find the target RGB parameter, and we use this to adjust the value of threshold. The threshold gave a range of approximate values that the object could take on depending on the lighting conditions and when the correct threshold range is found, the object finding location is quite robust.

Finally, there may be some noise, which we need to get rid of if we want to get a clear edge. Thus, we use a median filter. Then, after getting rid of noise, we put the image into binary, so that this can be doing the further blob detection.

After getting a clear image and convert it into binary, we calculate the key parameters such as moments which was derived in task 3 which can help us to get the centre of object.

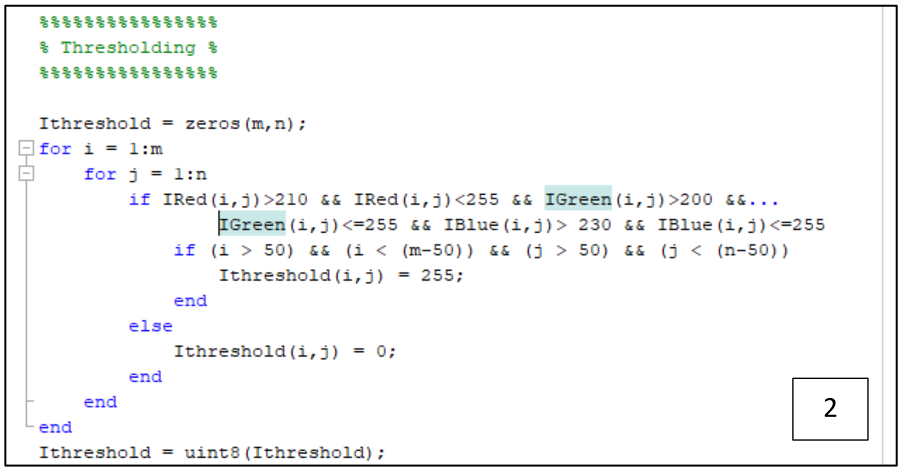
MATLAB code



```
%%%%%%%%%%%%%%%%
% Import Image %
%%%%%%%%%%%%%%%%

I = imread('object1.png');
%   figure, imshow(I)
%   title('normal')
CornerThreshold = 4;

% NOTE: I has 3 layers (RGB) even though it looks "black and white".

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Transform into Greyscale %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IRed = double(I(:,:,1));
IGreen = double(I(:,:,2));
IBlue = double(I(:,:,3));
IGrey = (IRed+IGreen+IBlue)/3;
I = uint8(IGrey);

I=double(I);
[m,n]=size(I);
```

1

1. This part is importing the image and convert the original 3D image into 1D grayscale image. The formular is to add each element for RGB together and divided them by 3. which gives the mean of the RGB elements pixel by pixel.

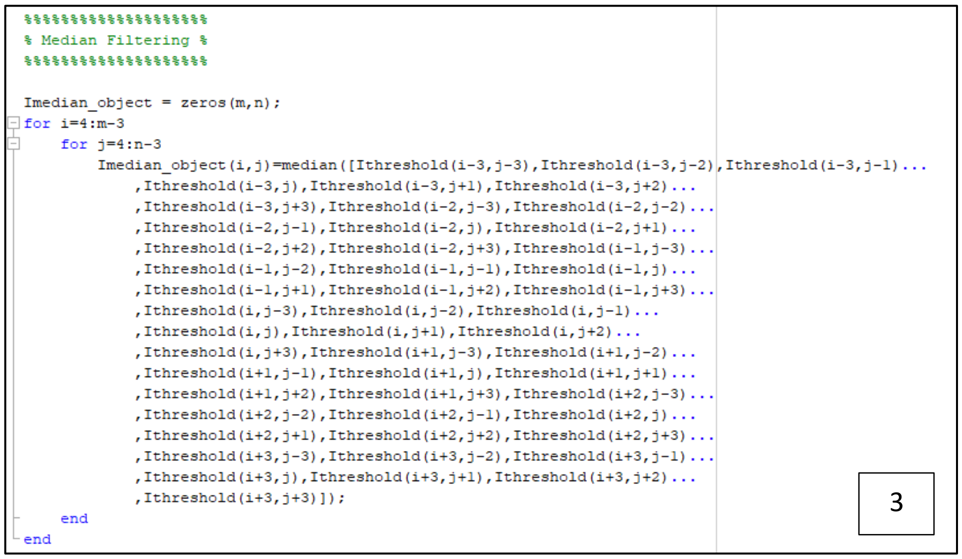


```
%%%%%%%%%%%%%%%%
% Thresholding %
%%%%%%%%%%%%%%%%

Ithreshold = zeros(m,n);
for i = 1:m
    for j = 1:n
        if IRed(i,j)>210 && IRed(i,j)<255 && IGreen(i,j)>200 &&...
                IGreen(i,j)<=255 && IBlue(i,j)> 230 && IBlue(i,j)<=255
            if (i > 50) && (i < (m-50)) && (j > 50) && (j < (n-50))
                Ithreshold(i,j) = 255;
            end
        else
            Ithreshold(i,j) = 0;
        end
    end
end
Ithreshold = uint8(Ithreshold);
```

2

2. This part we use two ‘for loop’ to scan through the image. After getting the grey scale image, we generate a zero matrix with same size as the 2D grayscale image. After that, according to the threshold we determined from the RGB finder, we use several condition statements to threshold the image. Above the threshold, where if the pixel is in the range, we set it to 255, and below the threshold we set to 0, which then gives us a black and white image.
A border is set up around the image of size 50 pixels from sides, top and bottom because

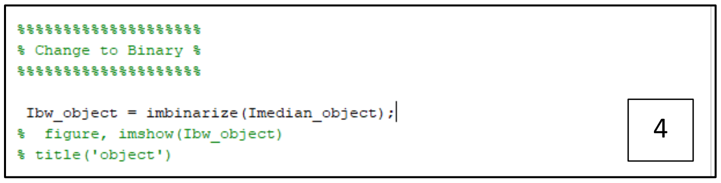
otherwise the object would be out of range of the robot arm and camera.



```
%%%%%%%%%%%%%%%%%%%%
% Median Filtering %
%%%%%%%%%%%%%%%%%%%%

Imedian_object = zeros(m,n);
for i=4:m-3
    for j=4:n-3
        Imedian_object(i,j)=median([Ithreshold(i-3,j-3),Ithreshold(i-3,j-2),Ithreshold(i-3,j-1)...
            ,Ithreshold(i-3,j),Ithreshold(i-3,j+1),Ithreshold(i-3,j+2)...
            ,Ithreshold(i-3,j+3),Ithreshold(i-2,j-3),Ithreshold(i-2,j-2)...
            ,Ithreshold(i-2,j-1),Ithreshold(i-2,j),Ithreshold(i-2,j+1)...
            ,Ithreshold(i-2,j+2),Ithreshold(i-2,j+3),Ithreshold(i-1,j-3)...
            ,Ithreshold(i-1,j-2),Ithreshold(i-1,j-1),Ithreshold(i-1,j)...
            ,Ithreshold(i-1,j+1),Ithreshold(i-1,j+2),Ithreshold(i-1,j+3)...
            ,Ithreshold(i,j-3),Ithreshold(i,j-2),Ithreshold(i,j-1)...
            ,Ithreshold(i,j),Ithreshold(i,j+1),Ithreshold(i,j+2)...
            ,Ithreshold(i,j+3),Ithreshold(i+1,j-3),Ithreshold(i+1,j-2)...
            ,Ithreshold(i+1,j-1),Ithreshold(i+1,j),Ithreshold(i+1,j+1)...
            ,Ithreshold(i+1,j+2),Ithreshold(i+1,j+3),Ithreshold(i+2,j-3)...
            ,Ithreshold(i+2,j-2),Ithreshold(i+2,j-1),Ithreshold(i+2,j)...
            ,Ithreshold(i+2,j+1),Ithreshold(i+2,j+2),Ithreshold(i+2,j+3)...
            ,Ithreshold(i+3,j-3),Ithreshold(i+3,j-2),Ithreshold(i+3,j-1)...
            ,Ithreshold(i+3,j),Ithreshold(i+3,j+1),Ithreshold(i+3,j+2)...
            ,Ithreshold(i+3,j+3)]);
    end
end
```

3

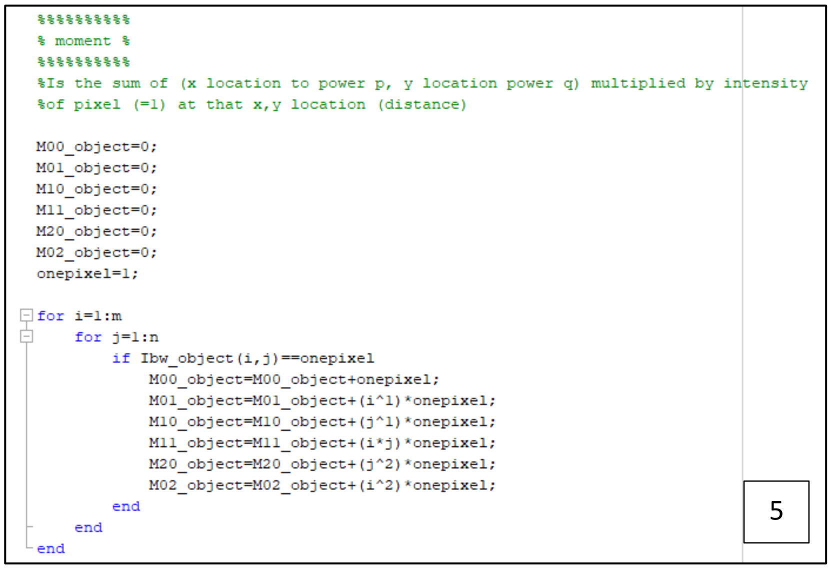
3. For this one, we scan through the image, and take the median for the adjacent pixel (PGP method). We divide the entire image into group pixels and take median in terms of groups.



```
%%%%%%%%%%%%%%%%%%%%
% Change to Binary %
%%%%%%%%%%%%%%%%%%%%

 Ibw_object = imbinarize(Imedian_object);
%  figure, imshow(Ibw_object)
% title('object')
```

4

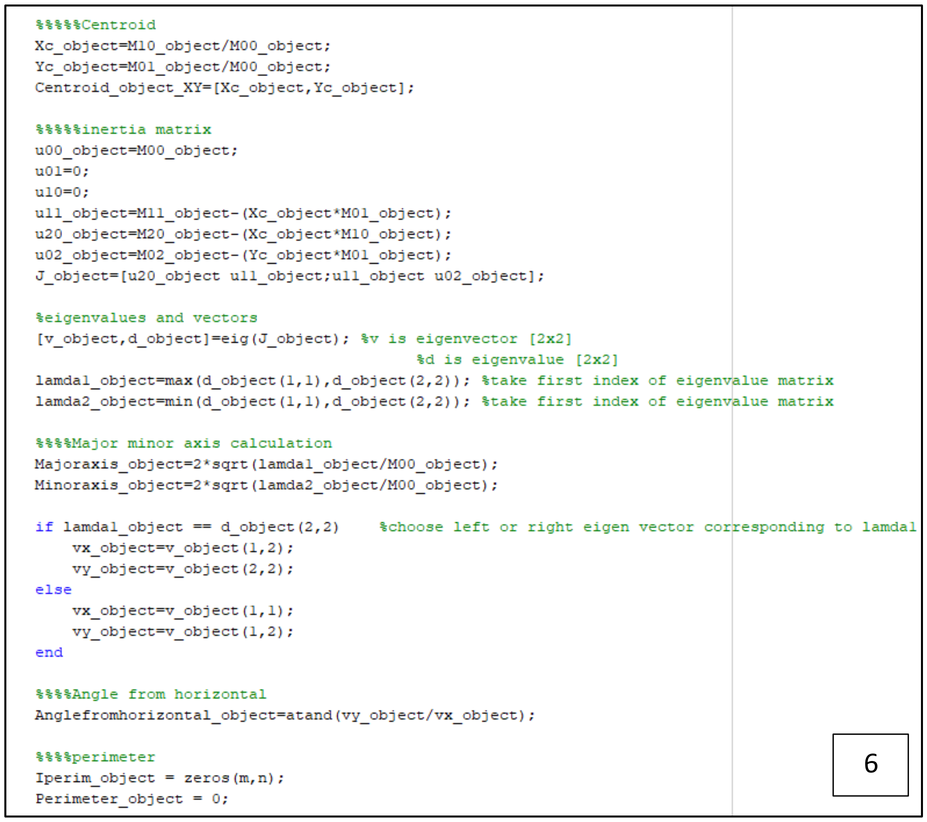
4. Change from the black and white image (0 or 255) into binary image (0 and 1) using imbinarize function



```
%%%%%%%%%%
% moment %
%%%%%%%%%%
%Is the sum of (x location to power p, y location power q) multiplied by intensity
%of pixel (=1) at that x,y location (distance)

M00_object=0;
M01_object=0;
M10_object=0;
M11_object=0;
M20_object=0;
M02_object=0;
onepixel=1;

for i=1:m
    for j=1:n
        if Ibw_object(i,j)==onepixel
            M00_object=M00_object+onepixel;
            M01_object=M01_object+(i^1)*onepixel;
            M10_object=M10_object+(j^1)*onepixel;
            M11_object=M11_object+(i*j)*onepixel;
            M20_object=M20_object+(j^2)*onepixel;
            M02_object=M02_object+(i^2)*onepixel;
        end
    end
end
```

5

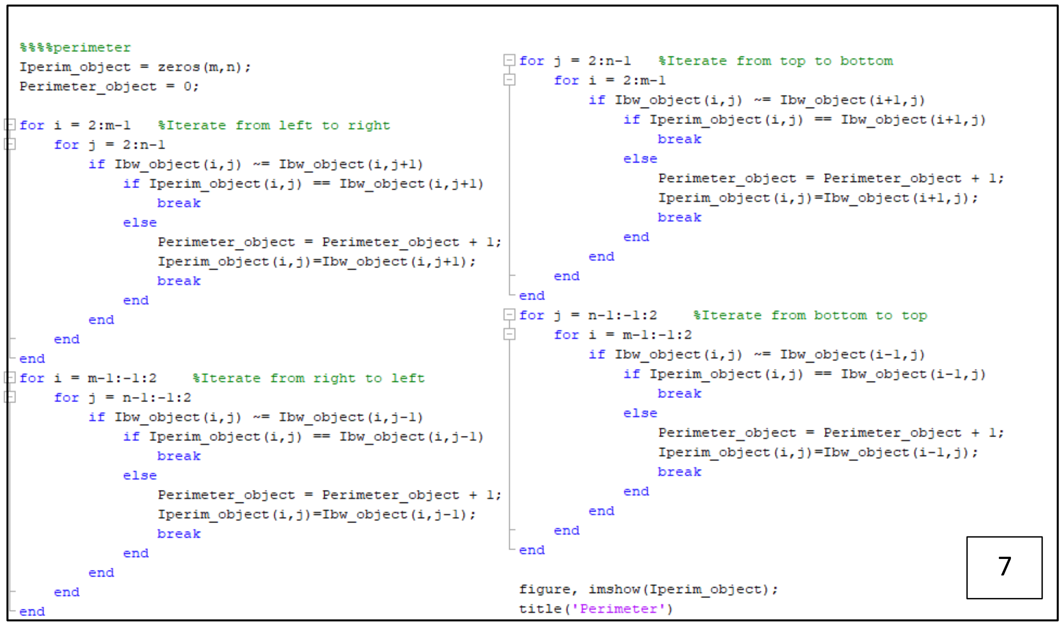
5. According to the formular of moment, we first calculate M00 to M02, and then multiplied by intensity of pixel (=1) at that x,y location (distance)



```
%%%%%Centroid
Xc_object=M10_object/M00_object;
Yc_object=M01_object/M00_object;
Centroid_object_XY=[Xc_object,Yc_object];

%%%%%inertia matrix
u00_object=M00_object;
u01=0;
u10=0;
u11_object=M11_object-(Xc_object*M01_object);
u20_object=M20_object-(Xc_object*M10_object);
u02_object=M02_object-(Yc_object*M01_object);
J_object=[u20_object u11_object;u11_object u02_object];

%eigenvalues and vectors
[v_object,d_object]=eig(J_object); %v is eigenvector [2x2]
                                   %d is eigenvalue [2x2]
lamda1_object=max(d_object(1,1),d_object(2,2)); %take first index of eigenvalue matrix
lamda2_object=min(d_object(1,1),d_object(2,2)); %take first index of eigenvalue matrix

%%%%Major minor axis calculation
Majoraxis_object=2*sqrt(lamda1_object/M00_object);
Minoraxis_object=2*sqrt(lamda2_object/M00_object);

if lamda1_object == d_object(2,2)    %choose left or right eigen vector corresponding to lamda1
    vx_object=v_object(1,2);
    vy_object=v_object(2,2);
else
    vx_object=v_object(1,1);
    vy_object=v_object(1,2);
end

%%%%Angle from horizontal
Anglefromhorizontal_object=atand(vy_object/vx_object);

%%%%perimeter
Iperim_object = zeros(m,n);
Perimeter_object = 0;
```

6

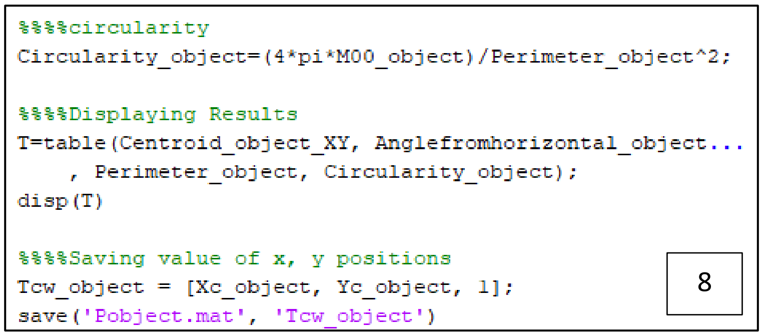
6. After we have the centroid of the target, we can implement the parameter we calculated above to derive major/minor axis of the target, the shape, the and also the orientation of the object according to the formular in the method part of task7



```
%%%%perimeter
Iperim_object = zeros(m,n);
Perimeter_object = 0;

for i = 2:m-1   %Iterate from left to right
    for j = 2:n-1
        if Ibw_object(i,j) ~= Ibw_object(i,j+1)
            if Iperim_object(i,j) == Ibw_object(i,j+1)
                break
            else
                Perimeter_object = Perimeter_object + 1;
                Iperim_object(i,j)=Ibw_object(i,j+1);
                break
            end
        end
    end
end
for i = m-1:-1:2    %Iterate from right to left
    for j = n-1:-1:2
        if Ibw_object(i,j) ~= Ibw_object(i,j-1)
            if Iperim_object(i,j) == Ibw_object(i,j-1)
                break
            else
                Perimeter_object = Perimeter_object + 1;
                Iperim_object(i,j)=Ibw_object(i,j-1);
                break
            end
        end
    end
end
for j = 2:n-1   %Iterate from top to bottom
    for i = 2:m-1
        if Ibw_object(i,j) ~= Ibw_object(i+1,j)
            if Iperim_object(i,j) == Ibw_object(i+1,j)
                break
            else
                Perimeter_object = Perimeter_object + 1;
                Iperim_object(i,j)=Ibw_object(i+1,j);
                break
            end
        end
    end
end
for j = n-1:-1:2    %Iterate from bottom to top
    for i = m-1:-1:2
        if Ibw_object(i,j) ~= Ibw_object(i-1,j)
            if Iperim_object(i,j) == Ibw_object(i-1,j)
                break
            else
                Perimeter_object = Perimeter_object + 1;
                Iperim_object(i,j)=Ibw_object(i-1,j);
                break
            end
        end
    end
end

figure, imshow(Iperim_object);
title('Perimeter')
```

7

7. Here we calculate the perimeter, which we write the code to scan the image from left to right; right to left; top to bottom; and bottom to top. Also, we introduce a counter to count the row and column that we are going to scan. If the current was different after scan, than we add one to the counter, which means scan the next row. Also, if the pixel found in a separate iteration, then the line is skipped.



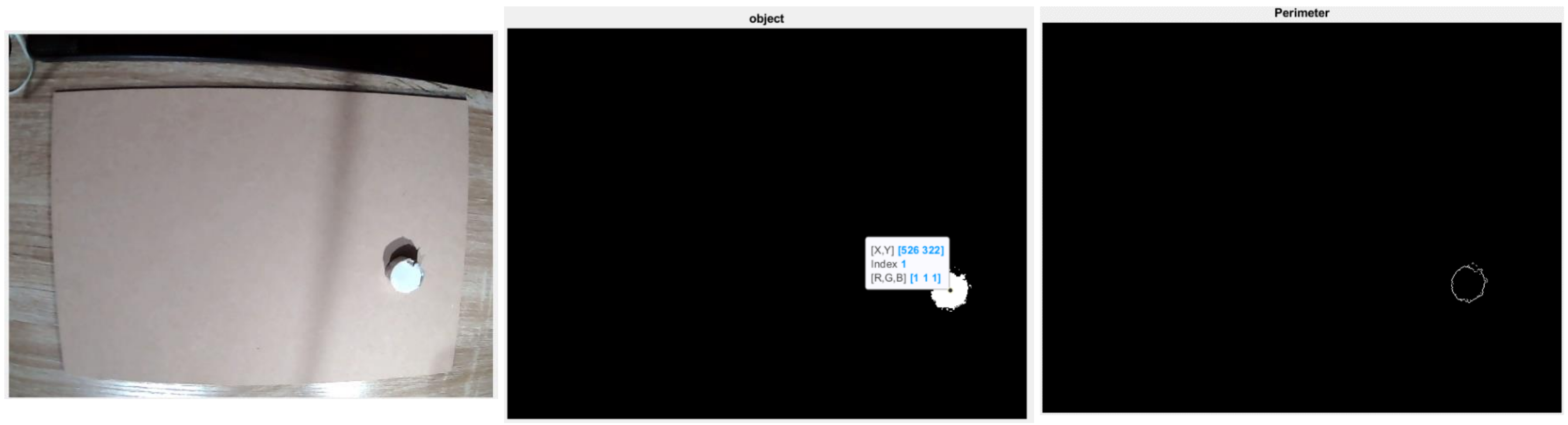
```
%%%%circularity
Circularity_object=(4*pi*M00_object)/Perimeter_object^2;

%%%%Displaying Results
T=table(Centroid_object_XY, Anglefromhorizontal_object...
    , Perimeter_object, Circularity_object);
disp(T)

%%%%Saving value of x, y positions
Tcw_object = [Xc_object, Yc_object, 1];
save('Pobject.mat', 'Tcw_object')
```

8

8. Finally, after getting all the values for the parameter, we calculate the circularity and shows together with the RGB value in a table.

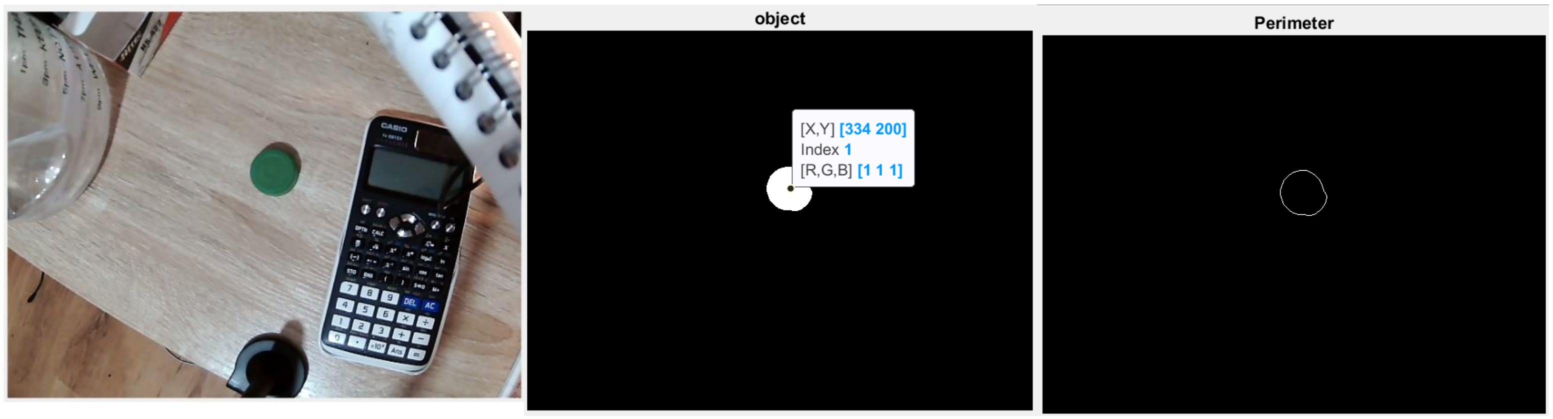
Results

After the image is taken, we run the image processing algorithm and the image can be detected successfully as shown below. We can also change the object and its position, and they can be detected accurately as well. As we can see from the result 'First', the Centroid_object_XY calculated by the MATLAB is 525.2, 321.98. As we click the centre of the processed image, it shows to be 526,322, which matches. After we rotated the target by 90, the angles from horizontal rotate 90 degrees as well. Similar result can be seen in the 'Second', where the centroid position and object are both detected. The centroid values are then saved in a .mat file to be used in task 8.



```
 Centroid_object_XY    Anglefromhorizontal_object    Perimeter_object    Circularity_object
 __________________    __________________________    ________________    __________________

  525.2     321.98               -53.851                    148                0.72516
```

First



```
Centroid_object_XY    Anglefromhorizontal_object    Perimeter_object    Circularity_object
__________________    __________________________    ________________    __________________

  332     201.18                43.257                     161                1.2028
```

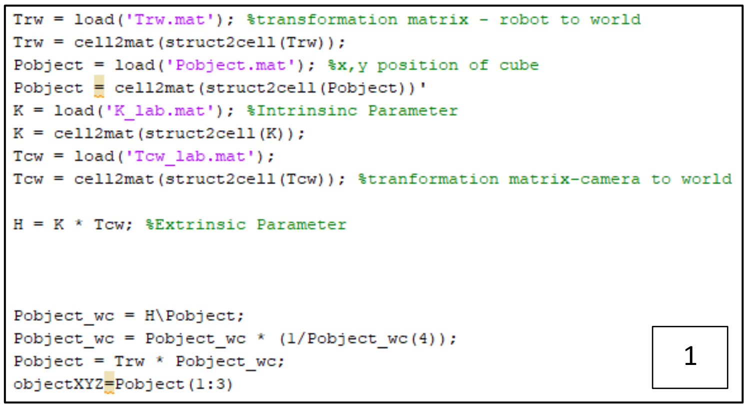
Second

Task 8 – Relating pixel position to robot position using experimental data

Method and Derivation

Until now, we have the transformation matrix of robot to world, the x,y position of the object, the intrinsic and extrinsic parameters, and the transformation matrix of camera to world. The pixel frame and the camera frame are connected by the intrinsic parameter K and the camera to world position and the robot to world position can be connected by the extrinsic parameter, which is H, and from this we can finally get the 2D pixel and the 3D robot frame.

MATLAB code



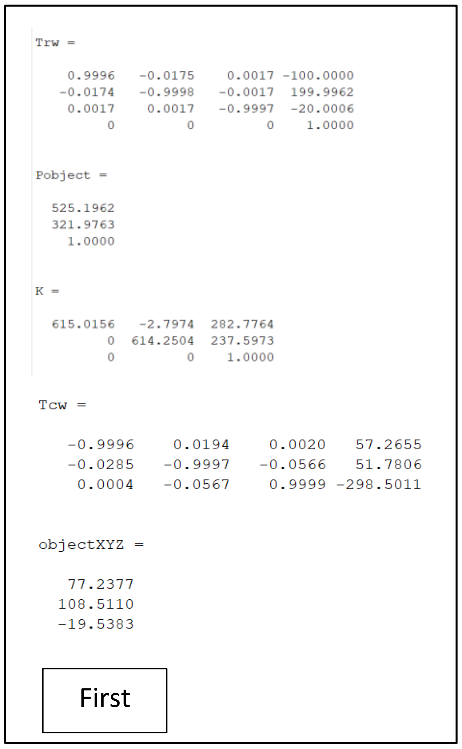
```
Trw = load('Trw.mat'); %transformation matrix - robot to world
Trw = cell2mat(struct2cell(Trw));
Pobject = load('Pobject.mat'); %x,y position of cube
Pobject = cell2mat(struct2cell(Pobject))'
K = load('K_lab.mat'); %Intrinsinc Parameter
K = cell2mat(struct2cell(K));
Tcw = load('Tcw_lab.mat');
Tcw = cell2mat(struct2cell(Tcw)); %tranformation matrix-camera to world

H = K * Tcw; %Extrinsic Parameter



Pobject_wc = H\Pobject;
Pobject_wc = Pobject_wc * (1/Pobject_wc(4));
Pobject = Trw * Pobject_wc;
objectXYZ=Pobject(1:3)
```

1

1. We load all the .mat files: Trw, K, Tcw, and Pobject in to the workspace, and use cell2mat function to convert the data in the processable format.
Then, we implement the calculations derived in task 4 to find the XYZ position of object.

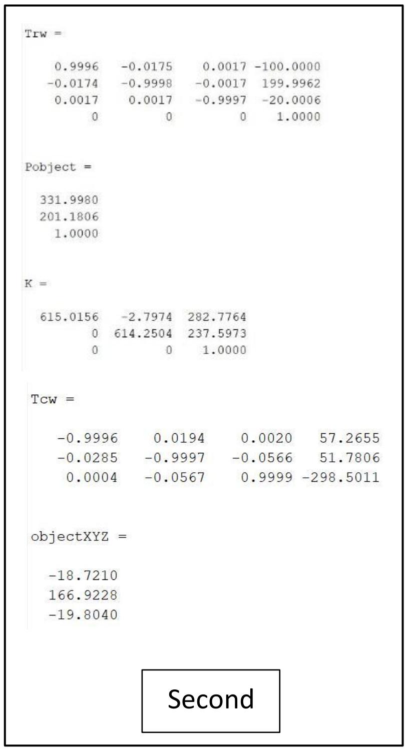
<u>Results</u>

The result of the code that we ran separately for the first and second image in task 7: ‘First’ and ‘Second’, gives the objects position. After we changed the position of the target twice, the XYZ position of the robot change as well, and doing a mental check, they make sense as the Z axis remains constant on the plane, but X,Y changes, acting similarly to task 4..



```
Trw =

    0.9996   -0.0175    0.0017 -100.0000
   -0.0174   -0.9998   -0.0017  199.9962
    0.0017    0.0017   -0.9997  -20.0006
         0         0         0    1.0000


Pobject =

  525.1962
  321.9763
    1.0000


K =

  615.0156   -2.7974  282.7764
         0  614.2504  237.5973
         0         0    1.0000


Tcw =

   -0.9996    0.0194    0.0020   57.2655
   -0.0285   -0.9997   -0.0566   51.7806
    0.0004   -0.0567    0.9999 -298.5011


objectXYZ =

   77.2377
  108.5110
  -19.5383
```

First



```
Trw =

    0.9996   -0.0175    0.0017 -100.0000
   -0.0174   -0.9998   -0.0017  199.9962
    0.0017    0.0017   -0.9997  -20.0006
         0         0         0    1.0000


Pobject =

  331.9980
  201.1806
    1.0000


K =

  615.0156   -2.7974  282.7764
         0  614.2504  237.5973
         0         0    1.0000


Tcw =

   -0.9996    0.0194    0.0020   57.2655
   -0.0285   -0.9997   -0.0566   51.7806
    0.0004   -0.0567    0.9999 -298.5011


objectXYZ =

  -18.7210
  166.9228
  -19.8040
```

Second

Trajectory planning

Task 9 –Trajectory Planning for Robot Motion

Method and Derivation

The trajectory planning is taking into consideration the time profile of the motion that the robot must execute. To control the velocity at the beginning and at the end of the motion to be zero we can use a cubic polynomial to manipulate the motion of the robot and avoid “rough and jerky” moves. When we want our robot to move from one point to another the cubic polynomial comes in very handy as we can control the time it takes for the robot to completely execute the movement while also controlling the number of intermediate movements the robot is going to take.

For our trajectory plan, we wanted the robot to move to the object over a period of 5 seconds, move back to the origin, place object at a new position over a period of 5 seconds and return back to the origin, which meant that multiple target points were included in the code for the robot to do this.

To ensure that the velocities at the start and end points are zero, we can use a cubic polynomial with the following equations:

$$u(t) = \begin{cases} a_0 + a_1 t + a_2 t^2 + a_3 t^3, t < t_f \\ u_f, t \geq t_f \end{cases}$$

$$\mathit{Then\ there\ are\ 4\ parameters\ } a_0, a_1, a_2, a_3 \mathit{\ which\ satisfy\ the\ 4\ constraints}$$

$$u(0) = u_0, \quad u(t_f) = u_f, \qquad \dot{u}(0) = 0, \qquad \dot{u}(t_f) = 0$$

$$\mathit{We\ can\ then\ find\ the\ parameters\ through\ solving\ 4\ simulateous\ equations}:$$

$$u(0) = a_0 + a_1 0 + a_2 0^2 + a_3 0^3 = a_0 = u_0$$

$$u(t_f) = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 = u_f$$

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

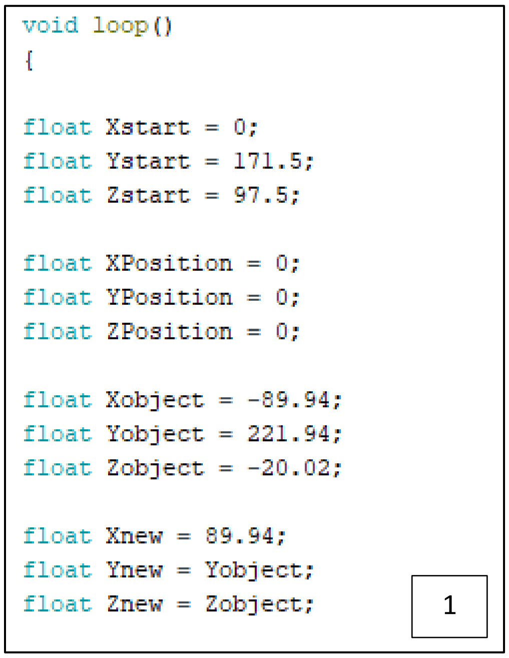
$$\dot{u}(t_f) = a_1 + 2a_2 t_f + 3a_3 t_f^2 = 0$$

$$\mathit{Thus\ we\ get\ the\ values}:$$

$$a_0 = u_0, \quad a_1 = 0, \quad a_2 = \frac{3}{t_f^2}(u_f - u_0), \quad a_3 = \frac{-2}{t_f^3}(u_f - u_0)$$

Where u0 is the initial position and uf is the final position. Thus with the initial and final position and the duration of the motion we can form a cubic polynomial for a smooth movement of the robot.

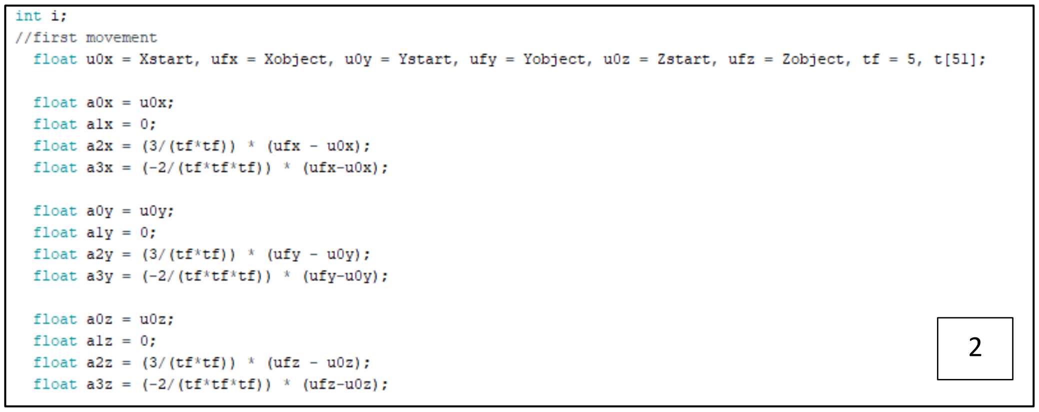
Arduino code



```
void loop()
{

float Xstart = 0;
float Ystart = 171.5;
float Zstart = 97.5;

float XPosition = 0;
float YPosition = 0;
float ZPosition = 0;

float Xobject = -89.94;
float Yobject = 221.94;
float Zobject = -20.02;

float Xnew = 89.94;
float Ynew = Yobject;
float Znew = Zobject;
```

1

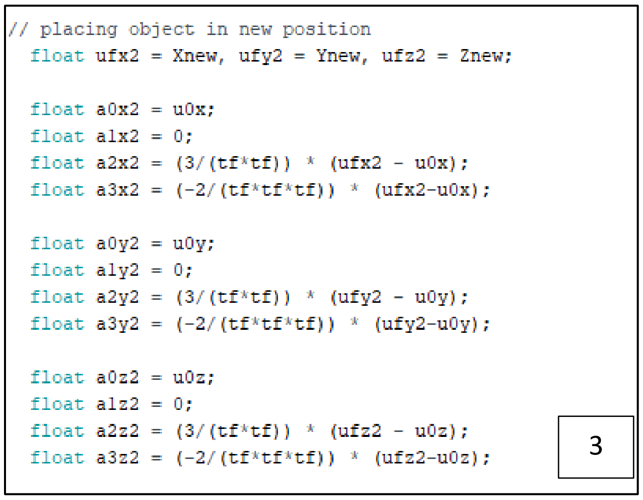
1. We will find the trajectory using cubic polynomial for each axis individually. For this Xstart, Ystart, Zstart are the starting position coordinates, XPosition , YPosition, ZPosition are the coordinates that are going to happen through the motion, Xobject, Yobject, Zobject are the coordinates at which we have the object, Xnew, Ynew, Znew are the coordinates of the place to where we are going to move the object.



```
int i;
//first movement
  float u0x = Xstart, ufx = Xobject, u0y = Ystart, ufy = Yobject, u0z = Zstart, ufz = Zobject, tf = 5, t[51];

  float a0x = u0x;
  float a1x = 0;
  float a2x = (3/(tf*tf)) * (ufx - u0x);
  float a3x = (-2/(tf*tf*tf)) * (ufx-u0x);

  float a0y = u0y;
  float a1y = 0;
  float a2y = (3/(tf*tf)) * (ufy - u0y);
  float a3y = (-2/(tf*tf*tf)) * (ufy-u0y);

  float a0z = u0z;
  float a1z = 0;
  float a2z = (3/(tf*tf)) * (ufz - u0z);
  float a3z = (-2/(tf*tf*tf)) * (ufz-u0z);
```

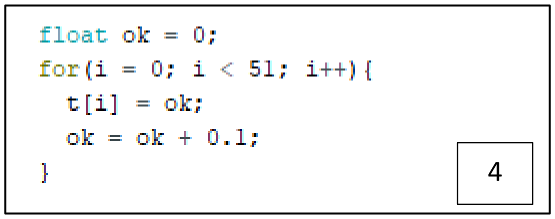
2

2. Here we declare the initial and final positions of each axis and we calculate the variables a0, a1, a2 and a3 for each axis as well.



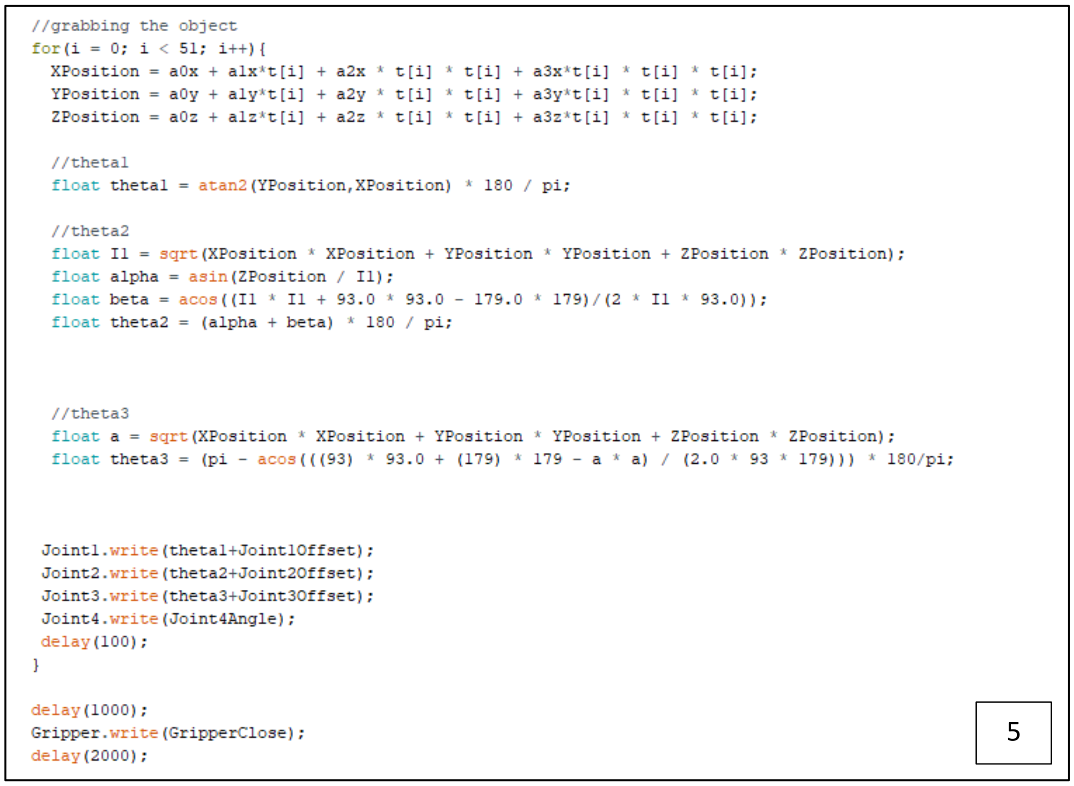
```
// placing object in new position
  float ufx2 = Xnew, ufy2 = Ynew, ufz2 = Znew;

  float a0x2 = u0x;
  float a1x2 = 0;
  float a2x2 = (3/(tf*tf)) * (ufx2 - u0x);
  float a3x2 = (-2/(tf*tf*tf)) * (ufx2-u0x);

  float a0y2 = u0y;
  float a1y2 = 0;
  float a2y2 = (3/(tf*tf)) * (ufy2 - u0y);
  float a3y2 = (-2/(tf*tf*tf)) * (ufy2-u0y);

  float a0z2 = u0z;
  float a1z2 = 0;
  float a2z2 = (3/(tf*tf)) * (ufz2 - u0z);
  float a3z2 = (-2/(tf*tf*tf)) * (ufz2-u0z);
```

3. We then declare the position of the place at which we are going to place the object after picking it up and then calculate the variables a0, a1, a2 and a3 for this trajectory.



```
float ok = 0;
for(i = 0; i < 51; i++){
  t[i] = ok;
  ok = ok + 0.1;
}
```

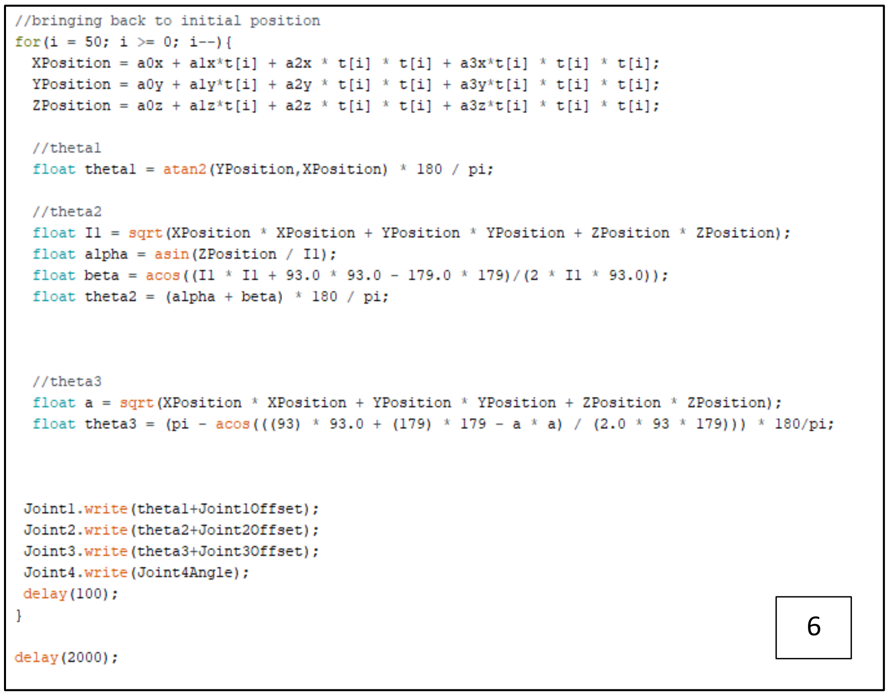
4. After that we form a time vector that has 50 increments of 0.1 and it's going to help us in calculating the positions and making the motion last for 5 second.



```
//grabbing the object
for(i = 0; i < 51; i++){
  XPosition = a0x + a1x*t[i] + a2x * t[i] * t[i] + a3x*t[i] * t[i] * t[i];
  YPosition = a0y + a1y*t[i] + a2y * t[i] * t[i] + a3y*t[i] * t[i] * t[i];
  ZPosition = a0z + a1z*t[i] + a2z * t[i] * t[i] + a3z*t[i] * t[i] * t[i];

  //theta1
  float theta1 = atan2(YPosition,XPosition) * 180 / pi;

  //theta2
  float I1 = sqrt(XPosition * XPosition + YPosition * YPosition + ZPosition * ZPosition);
  float alpha = asin(ZPosition / I1);
  float beta = acos((I1 * I1 + 93.0 * 93.0 - 179.0 * 179)/(2 * I1 * 93.0));
  float theta2 = (alpha + beta) * 180 / pi;



  //theta3
  float a = sqrt(XPosition * XPosition + YPosition * YPosition + ZPosition * ZPosition);
  float theta3 = (pi - acos(((93) * 93.0 + (179) * 179 - a * a) / (2.0 * 93 * 179))) * 180/pi;



 Joint1.write(theta1+Joint1Offset);
 Joint2.write(theta2+Joint2Offset);
 Joint3.write(theta3+Joint3Offset);
 Joint4.write(Joint4Angle);
 delay(100);
}

delay(1000);
Gripper.write(GripperClose);
delay(2000);
```

5. We then use for loop to make the robot move. To do so we calculate XPosition, YPostion, ZPosition using the equation for each t[i] of the array that we previously made, after which we calculate the angles theta1, theta2 and theta3 using inverse kinematics. After having the angles we use

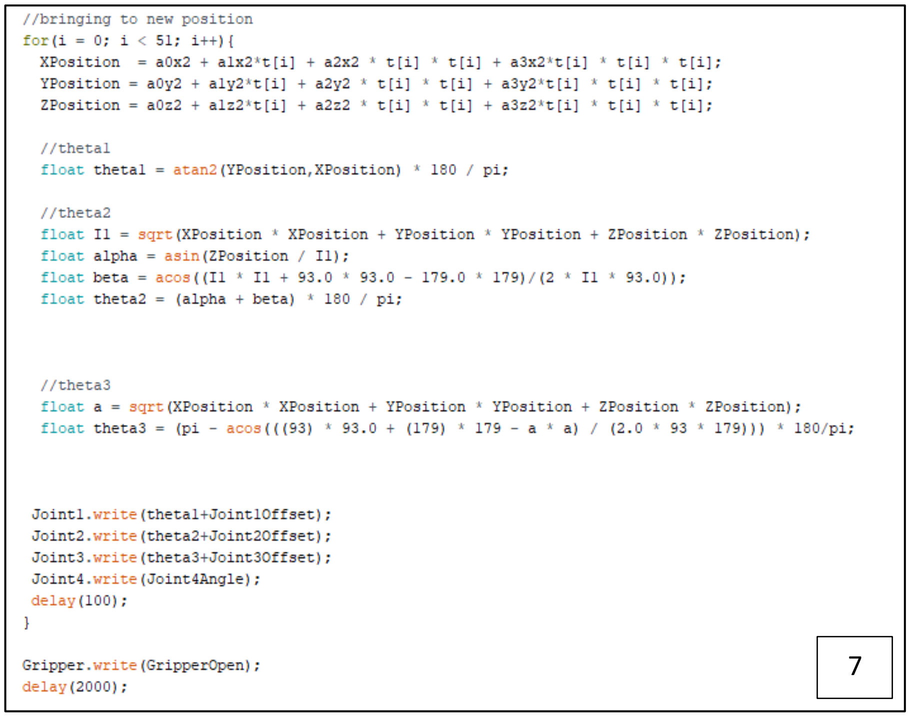
"Joint.write()" for each joint to move and very importantly, at the end of each loop we have a delay of 100 ms, as we want the whole motion to last for 5 seconds and there are 50 loops, thus 50 * 100 ms = 5 s. After the for loop we get to the desired position, we wait one second, we close the gripper to pick up the object and then we wait another 2 seconds.



```
//bringing back to initial position
for(i = 50; i >= 0; i--){
  XPosition = a0x + a1x*t[i] + a2x * t[i] * t[i] + a3x*t[i] * t[i] * t[i];
  YPosition = a0y + a1y*t[i] + a2y * t[i] * t[i] + a3y*t[i] * t[i] * t[i];
  ZPosition = a0z + a1z*t[i] + a2z * t[i] * t[i] + a3z*t[i] * t[i] * t[i];

  //theta1
  float theta1 = atan2(YPosition,XPosition) * 180 / pi;

  //theta2
  float I1 = sqrt(XPosition * XPosition + YPosition * YPosition + ZPosition * ZPosition);
  float alpha = asin(ZPosition / I1);
  float beta = acos((I1 * I1 + 93.0 * 93.0 - 179.0 * 179)/(2 * I1 * 93.0));
  float theta2 = (alpha + beta) * 180 / pi;



  //theta3
  float a = sqrt(XPosition * XPosition + YPosition * YPosition + ZPosition * ZPosition);
  float theta3 = (pi - acos(((93) * 93.0 + (179) * 179 - a * a) / (2.0 * 93 * 179))) * 180/pi;



 Joint1.write(theta1+Joint1Offset);
 Joint2.write(theta2+Joint2Offset);
 Joint3.write(theta3+Joint3Offset);
 Joint4.write(Joint4Angle);
 delay(100);
}

delay(2000);
```

6

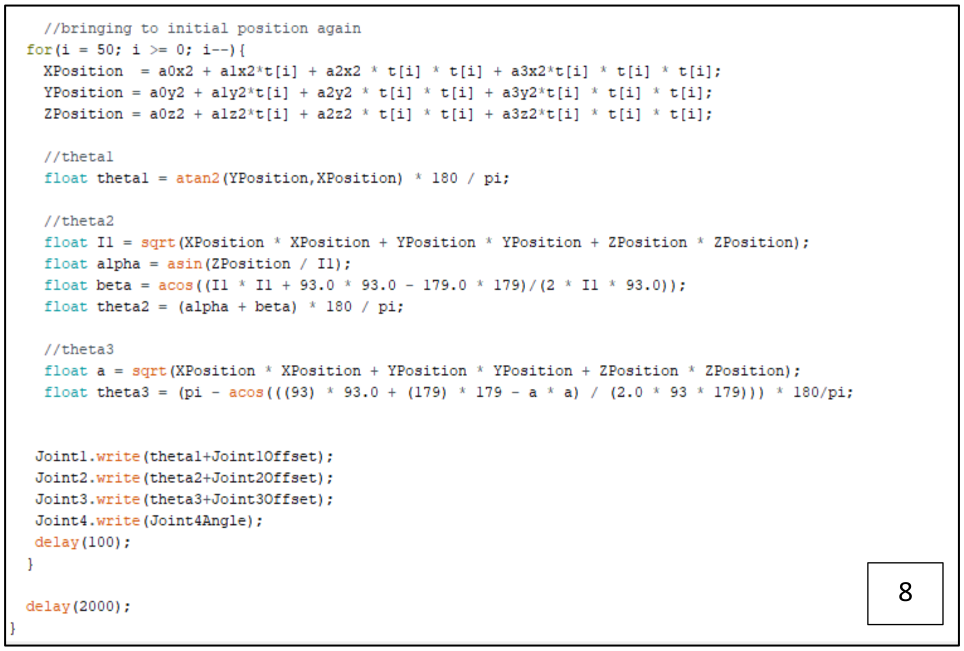
6. To go back to the initial position, we apply the same logic but this time the loop goes from 50 to 0 as we are going backwards to where we started. We then wait for two second until the next action.



```
//bringing to new position
for(i = 0; i < 51; i++){
  XPosition  = a0x2 + a1x2*t[i] + a2x2 * t[i] * t[i] + a3x2*t[i] * t[i] * t[i];
  YPosition = a0y2 + a1y2*t[i] + a2y2 * t[i] * t[i] + a3y2*t[i] * t[i] * t[i];
  ZPosition = a0z2 + a1z2*t[i] + a2z2 * t[i] * t[i] + a3z2*t[i] * t[i] * t[i];

  //theta1
  float theta1 = atan2(YPosition,XPosition) * 180 / pi;

  //theta2
  float I1 = sqrt(XPosition * XPosition + YPosition * YPosition + ZPosition * ZPosition);
  float alpha = asin(ZPosition / I1);
  float beta = acos((I1 * I1 + 93.0 * 93.0 - 179.0 * 179)/(2 * I1 * 93.0));
  float theta2 = (alpha + beta) * 180 / pi;



  //theta3
  float a = sqrt(XPosition * XPosition + YPosition * YPosition + ZPosition * ZPosition);
  float theta3 = (pi - acos(((93) * 93.0 + (179) * 179 - a * a) / (2.0 * 93 * 179))) * 180/pi;



 Joint1.write(theta1+Joint1Offset);
 Joint2.write(theta2+Joint2Offset);
 Joint3.write(theta3+Joint3Offset);
 Joint4.write(Joint4Angle);
 delay(100);
}

Gripper.write(GripperOpen);
delay(2000);
```

7

7. We then want to place the object to a new place so we use again the same logic and the same loop, but now the variables for the final position change to the coordinates of the new end position. Again for each XPosition, YPosition, ZPosition we find the angles and then move the joints, each loop having a delay of 100ms. After reaching the wanted destination the gripper opens to drop the object.



```
  //bringing to initial position again
for(i = 50; i >= 0; i--){
  XPosition  = a0x2 + a1x2*t[i] + a2x2 * t[i] * t[i] + a3x2*t[i] * t[i] * t[i];
  YPosition = a0y2 + a1y2*t[i] + a2y2 * t[i] * t[i] + a3y2*t[i] * t[i] * t[i];
  ZPosition = a0z2 + a1z2*t[i] + a2z2 * t[i] * t[i] + a3z2*t[i] * t[i] * t[i];

  //theta1
  float theta1 = atan2(YPosition,XPosition) * 180 / pi;

  //theta2
  float I1 = sqrt(XPosition * XPosition + YPosition * YPosition + ZPosition * ZPosition);
  float alpha = asin(ZPosition / I1);
  float beta = acos((I1 * I1 + 93.0 * 93.0 - 179.0 * 179)/(2 * I1 * 93.0));
  float theta2 = (alpha + beta) * 180 / pi;

  //theta3
  float a = sqrt(XPosition * XPosition + YPosition * YPosition + ZPosition * ZPosition);
  float theta3 = (pi - acos(((93) * 93.0 + (179) * 179 - a * a) / (2.0 * 93 * 179))) * 180/pi;


 Joint1.write(theta1+Joint1Offset);
 Joint2.write(theta2+Joint2Offset);
 Joint3.write(theta3+Joint3Offset);
 Joint4.write(Joint4Angle);
 delay(100);
}

delay(2000);
}
```

8

8. In the end we bring the robot to its initial position with a loop that goes from 50 to 0, calculating the positions and angles each loop.

Results

We chose the starting points to be:

Xstart = 0; Ystart = 171.5; Zstart = 97.5;

The objects calculated position:

Xobject = -89.94; Yobject = 221.94; Zobject = -20.02;

The position for where the robot was to place the object:

Xnew = 89.94; Ynew = Yobject; Znew = Zobject;

Firstly, we input these values into MATLAB, to simulate the cubic trajectory, as can be seen in figure 5.0, we can also see the change of position and speed on each axis.

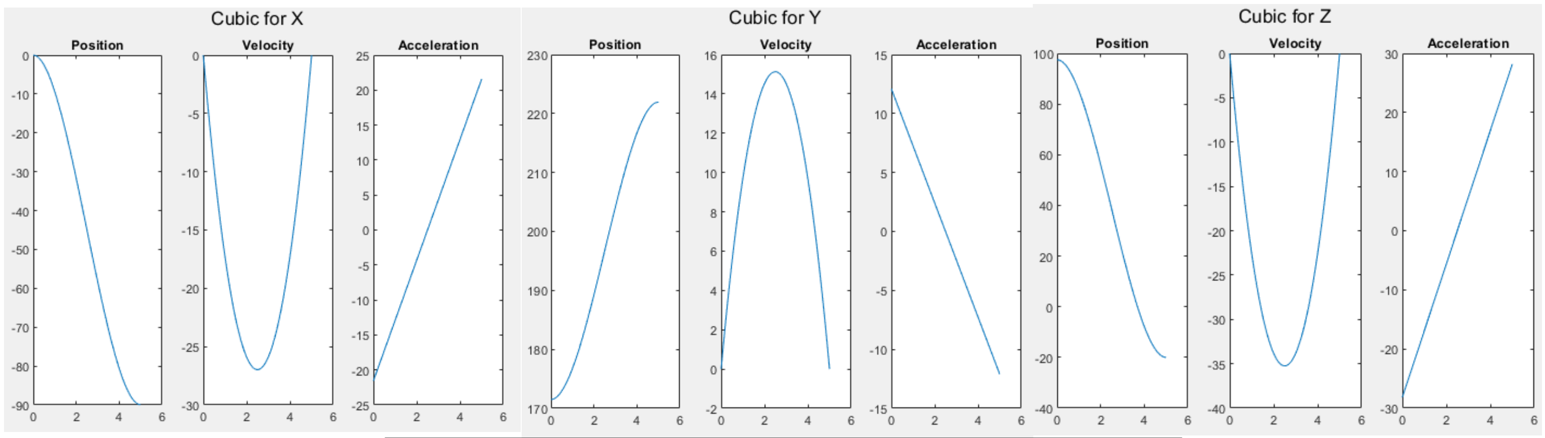
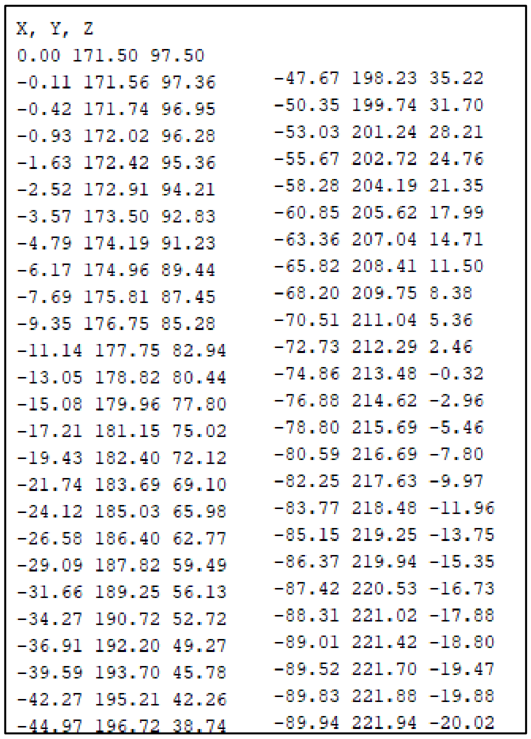


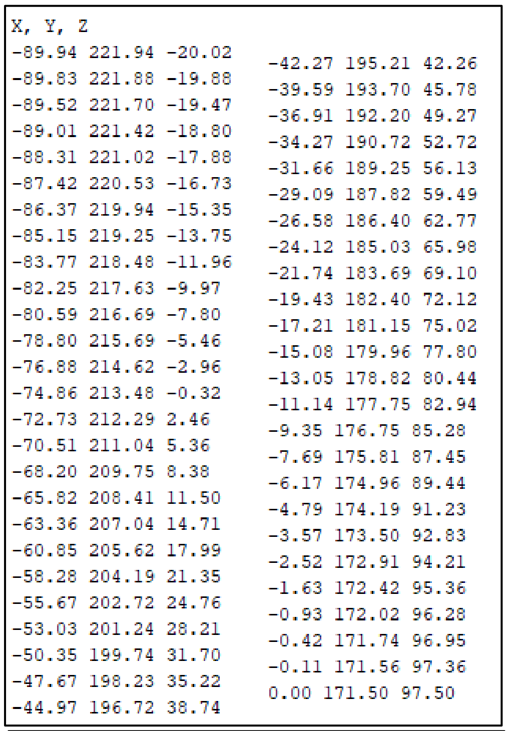
Figure 5.0 - MATLAB simulation of cubic trajectory for X, Y, Z

Then after uploading the Arduino code to the robot, we also serial print the values of the axis positions, X, Y, Z to observe the change for 50 increments from the initial position to the final position (figure 5.1), and then in another 50 moves back to the initial position (figure 5.2) and these value match the simulated values done in MATLAB.



```
X, Y, Z
0.00 171.50 97.50
-0.11 171.56 97.36      -47.67 198.23 35.22
-0.42 171.74 96.95      -50.35 199.74 31.70
-0.93 172.02 96.28      -53.03 201.24 28.21
-1.63 172.42 95.36      -55.67 202.72 24.76
-2.52 172.91 94.21      -58.28 204.19 21.35
-3.57 173.50 92.83      -60.85 205.62 17.99
-4.79 174.19 91.23      -63.36 207.04 14.71
-6.17 174.96 89.44      -65.82 208.41 11.50
-7.69 175.81 87.45      -68.20 209.75 8.38
-9.35 176.75 85.28      -70.51 211.04 5.36
-11.14 177.75 82.94     -72.73 212.29 2.46
-13.05 178.82 80.44     -74.86 213.48 -0.32
-15.08 179.96 77.80     -76.88 214.62 -2.96
-17.21 181.15 75.02     -78.80 215.69 -5.46
-19.43 182.40 72.12     -80.59 216.69 -7.80
-21.74 183.69 69.10     -82.25 217.63 -9.97
-24.12 185.03 65.98     -83.77 218.48 -11.96
-26.58 186.40 62.77     -85.15 219.25 -13.75
-29.09 187.82 59.49     -86.37 219.94 -15.35
-31.66 189.25 56.13     -87.42 220.53 -16.73
-34.27 190.72 52.72     -88.31 221.02 -17.88
-36.91 192.20 49.27     -89.01 221.42 -18.80
-39.59 193.70 45.78     -89.52 221.70 -19.47
-42.27 195.21 42.26     -89.83 221.88 -19.88
-44.97 196.72 38.74     -89.94 221.94 -20.02
```

Figure 5.1 - Cubic trajectory positions moving back to the start



```
X, Y, Z
-89.94 221.94 -20.02
-89.83 221.88 -19.88    -42.27 195.21 42.26
-89.52 221.70 -19.47    -39.59 193.70 45.78
-89.01 221.42 -18.80    -36.91 192.20 49.27
-88.31 221.02 -17.88    -34.27 190.72 52.72
-87.42 220.53 -16.73    -31.66 189.25 56.13
-86.37 219.94 -15.35    -29.09 187.82 59.49
-85.15 219.25 -13.75    -26.58 186.40 62.77
-83.77 218.48 -11.96    -24.12 185.03 65.98
-82.25 217.63 -9.97     -21.74 183.69 69.10
-80.59 216.69 -7.80     -19.43 182.40 72.12
-78.80 215.69 -5.46     -17.21 181.15 75.02
-76.88 214.62 -2.96     -15.08 179.96 77.80
-74.86 213.48 -0.32     -13.05 178.82 80.44
-72.73 212.29 2.46      -11.14 177.75 82.94
-70.51 211.04 5.36      -9.35 176.75 85.28
-68.20 209.75 8.38      -7.69 175.81 87.45
-65.82 208.41 11.50     -6.17 174.96 89.44
-63.36 207.04 14.71     -4.79 174.19 91.23
-60.85 205.62 17.99     -3.57 173.50 92.83
-58.28 204.19 21.35     -2.52 172.91 94.21
-55.67 202.72 24.76     -1.63 172.42 95.36
-53.03 201.24 28.21     -0.93 172.02 96.28
-50.35 199.74 31.70     -0.42 171.74 96.95
-47.67 198.23 35.22     -0.11 171.56 97.36
-44.97 196.72 38.74     0.00 171.50 97.50
```

Figure 5.2 Cubic trajectory positions moving back to the start

For these sets of values the robot performed very well, as the robot moved smoothly throughout the whole motion, and the object was picked and dropped to the desired place successfully.