Homework – 3

Name: **Sridhar Mocherla**

Part A:

For the given set of coordinates in line_data.txt, we fit them to lines using non-homogeneous linear least square and homogeneous linear least squares (also called Total Least Squares). First, we determine and show the results for Non homogeneous linear least squares below.

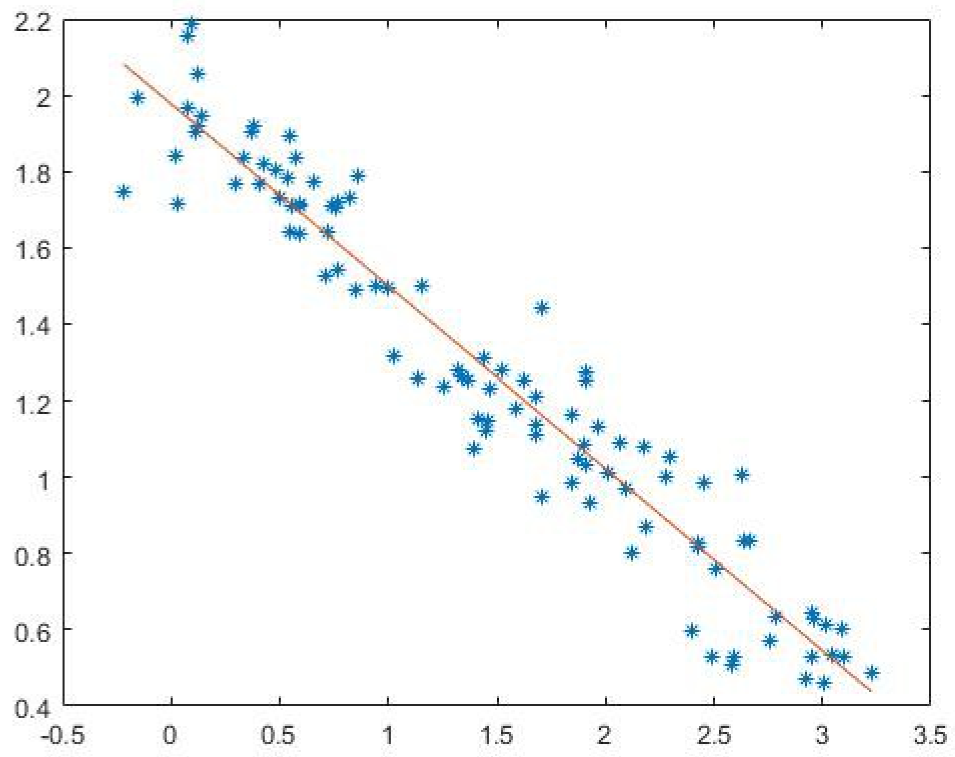

Figure 1: Using non-homogeneous squares to fit points in line_data.txt

Figure 1 shows that this technique provides a very good fit to the points. The slope and intercept of the line under Non Homogeneous Linear Least Squares (NHLLS) is as follows:

$$\begin{pmatrix} m \\ c \\ RMS_y \\ RMS_{dist} \end{pmatrix} = \begin{pmatrix} -0.4772 \\ 1.9769 \\ 0.1250 \\ 0.1128 \end{pmatrix}$$

Figure 2: Slope(m), Y-intercept (c) and RMS errors for Y and perpendicular distances.

Figure 2 shows the shows values obtained for slope, intercept, the RMS errors using the Non Homogeneous Linear Least Squares method. The code snippet to obtain these values is below:

```
U=[];
for i=1:100
    U=[U;x(i),1];
end
result = pinv(U)*y';
slope = result(1,:);
intercept = result(2,:);
```

Now we obtain the plot for the line using Total Least Squares method.

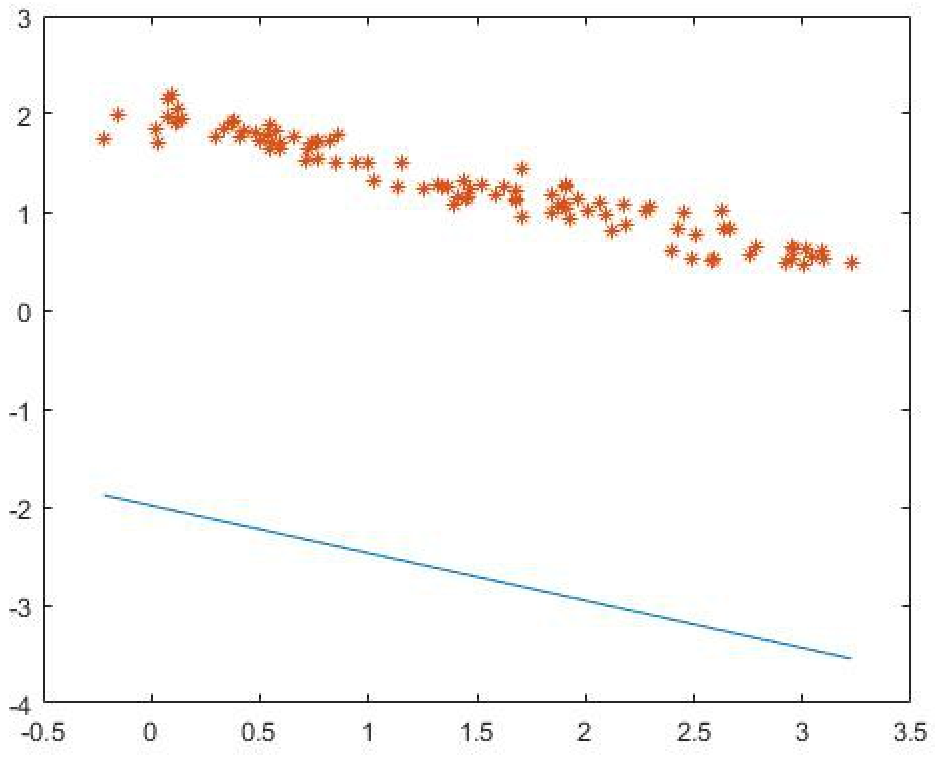

Figure 3: Plot for line fitting using Total Least Squares

Figure 3 shows the plot for the line fitted using Total Least Squares. As we, it is not very accurate and this is reflected in the RMS error we obtain.

$$\begin{pmatrix} m \\ c \\ RMS_y \\ RMS_{dist} \end{pmatrix} = \begin{pmatrix} -0.4840 \\ -1.9868 \\ 3.9756 \\ 3.5784 \end{pmatrix}$$

Figure 4: Slope, Y-intercept, RMS errors under TLA

The code snippet to obtain these values using TLA method is below:

```
A=[];
for i=1:size(x,2)
    A = [A;x(i) - xMean,y(i) - yMean];
end

[V,D] = eig(A'*A);
eigVec = V(:,1)
a = eigVec(1);
b = eigVec(2);
slope_tla = -a/b;
intercept_tla = d/b;
```

Part B

We consider 15 points distributed equally in each of the planes (XY, YZ and XZ). Figure 5 shows the position of these points. The values of these coordinates according to the world coordinate system are listed in the world_coords.txt. The MATLAB image coordinates for these points is listed in image_coords.txt.

The code snippet to obtain the image coordinates of these 15 points is below:

```
[u,v] = ginput(15);
```

From figure 5, we have labelled the points P_1, P_2,…….P_{15} and we also observe that there are 5 points in each plane so that calibration is more accurate across all the planes.

To determine the camera matrix (M), we use the homogeneous linear least squares method and we obtain the 2nx12 matrix P by below MATLAB code(in partb.m):

```
zeroelem = repelem(0,4); % 1x4 0 vector for building P

for i=1:15
    world_row = world_coords(i,:);
    image_coord = image_coords(i,:);
    img_u = image_coord(1);
    temp1 = -img_u*world_row;
    row1 = [world_row,zeroelem,temp1]; %build row 1 for point
    img_v = image_coord(2);
    temp2 = -img_v*world_row;
    row2 = [zeroelem,world_row,temp2]; %build row 2 for point

    P=[P;row1;row2];
end
```

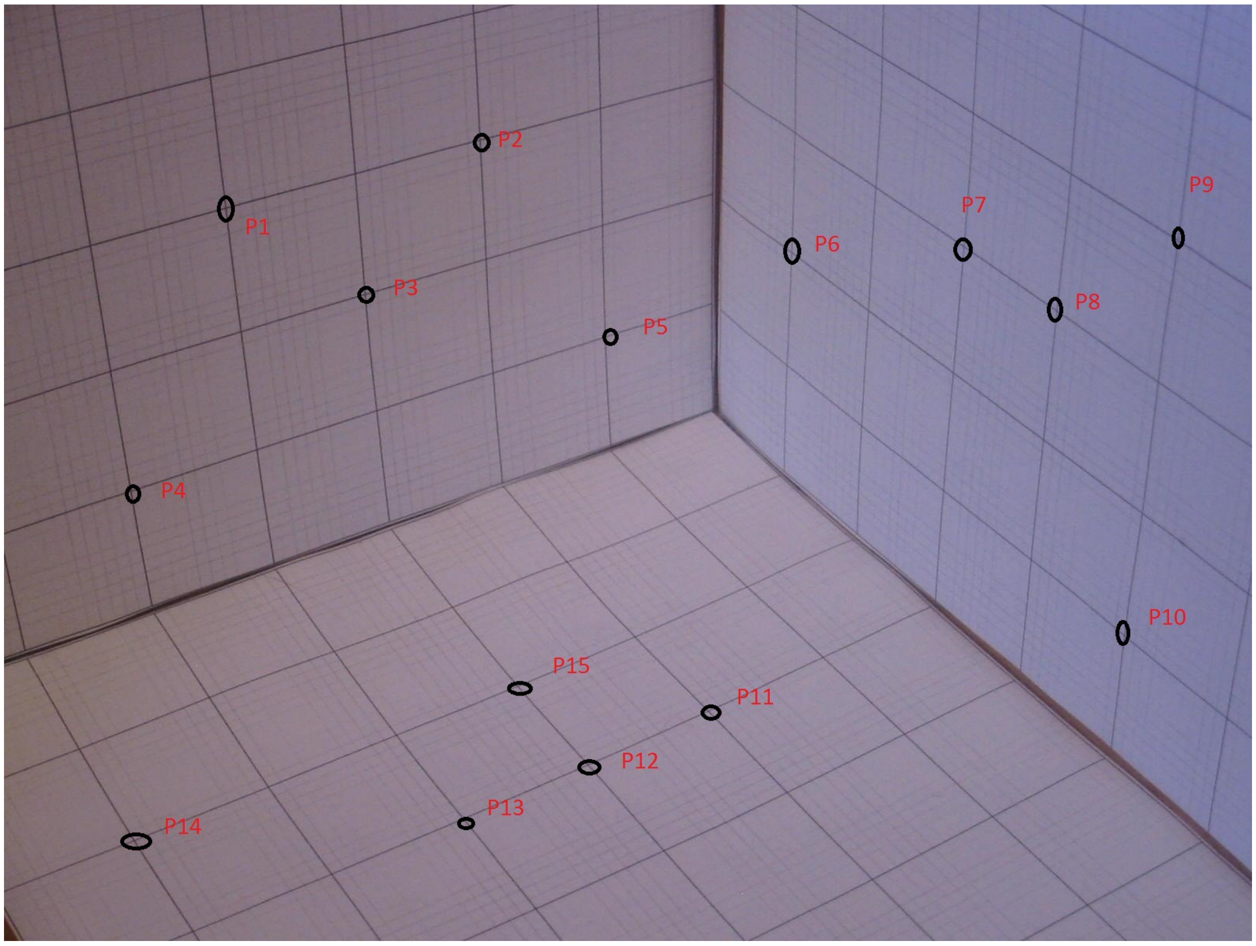

Figure 5: 15 points selected to calibrate the camera using Total Least Squares.

After obtaining P, we solve for PM = 0 and use compute the minimum eigen value of P'*P using eig() function. The code for obtaining M is below:

```
transp_prod = P'*P;
[V,D] = eig(transp_prod);
if ~issorted(diag(D))
    [V,D] = eig(transp_prod);
    [D,I] = sort(diag(D));
    V = V(:, I);
```

```
end
%[U, S, V] = svd(P,0);
%[U, S, V] = svd(P,0);
m = V(:,1);%12x1 unit vector with norm(m)=1

M = reshape(m,4,3)';%camera_matrix
m1 = m(1:4)';
m2 = m(5:8)';
m3 = m(9:12)';
```

The camera matrix (M) obtained by this method is below:

$$M = \begin{bmatrix} -0.1391 & 0.0541 & -0.0224 & 0.8443 \\ 0.0319 & 0.0548 & -0.1401 & 0.4906 \\ -0.0000 & -0.0000 & -0.0000 & 0.0009 \end{bmatrix}$$

Figure 6: 3x4 Camera matrix (M)

Now we project the 3D coordinates (after making them homogeneous) onto the 2D image to get the estimate of where the camera would project them using M. The equation to obtain these image coordinates is:

$$\begin{bmatrix} U \\ V \\ W \end{bmatrix} = M * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Figure 7: Equation to obtain image coordinates based on world coordinates using M

Now that we've obtained the (u,v) = (U/W,V/W) values we project them onto the image and see where they land up compared to the points that we actually chose. Figure 8 illustrates this clearly.

We also find the RMS error for where the points ended up and where the points actually where. The RMS error obtained is **7.3003**. The Sum of Squared Error (SSE) is **799.4190**. The RMS error is acceptable and shows that the calibration was accurate.

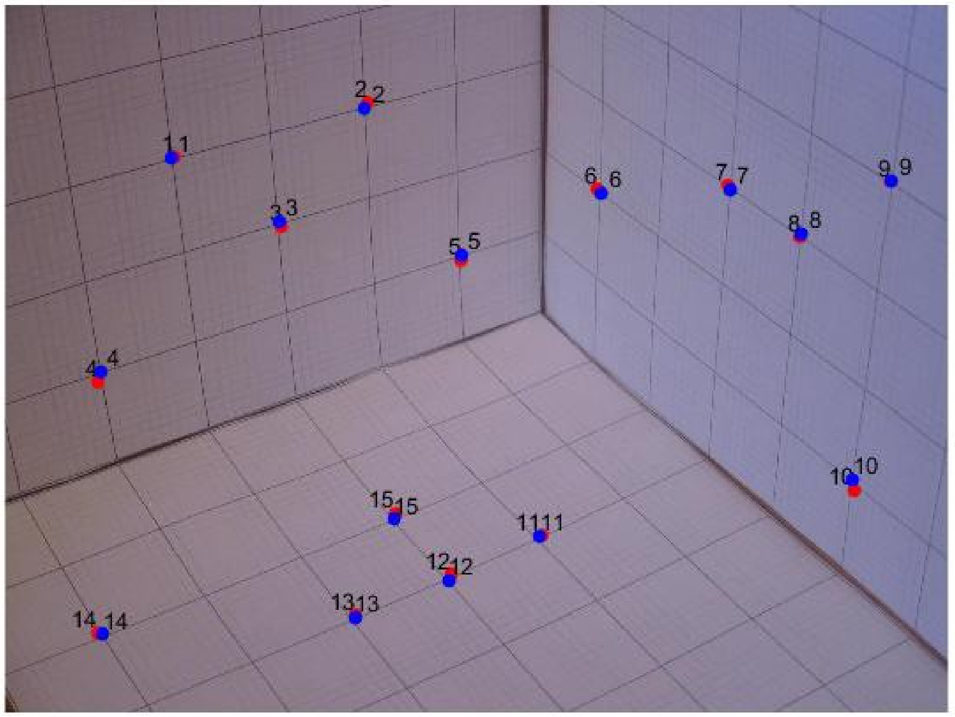

Figure 8: Comparison of actual clicked points and the points obtained by mapping using camera matrix. Red points are the projections and Blue points are the clicked locations.

Part C:

Generating a sphere:

We have the equations to generate the 3D coordinates as functions of Θ and Φ:

```
x = x0 + cos(Φ)*cos(Θ)*R
y = y0 + cos(Φ)*sin(Θ)*R
z = z0 + sin(Φ)*R
```

Here (x0,y0,z0) = (3,2,3) which is the center of the sphere.

Θ ranges from 0 to 2∏ and Φ from -∏/2 to ∏/2. Since x, y and z are functions of 2 variables we can generate data points using meshgrid() function in MATLAB. It generates grid coordinates for the specified variables. The process for rendering the sphere consists of a series of steps:

1. Generate the (x,y,z) coordinates based on the angles.
2. Find the visible points based on the camera position.

3. Find the areas to shade based on light direction.
4. Render the sphere in the image 2.

We explore each of these steps in detail with code snippets for each of them to explain the implementation.

1. Generate the (x,y,z) coordinates based on the angles.
   We use meshgrid() function to generate values of phi and theta as x,y and z are functions of two variables. Then, we use the equations specified to generate (x,y,z) triplets for our sphere. Below MATLAB code snippet illustrates this:

```
phi = linspace(-pi/2,pi/2,100); %generate input values for phi
theta = linspace(0,2*pi,100);%generate input values for theta

[phi,theta] = meshgrid(phi,theta);%obtain grid points as x,y,z are functions
of two variables

R=0.5; %radius
x = 3 + cos(phi).*cos(theta)*R;
y = 2 + cos(phi).*sin(theta)*R;
z = 3 + sin(phi)*R;
```

2. Find the visible points with respect to camera.
   To find the visible points with respect to camera, we need to find the surface normal for each point on the sphere. Surfnorm() returns the directions of the normals for a set of coordinates. Then, using the equation specified in the question (P-X).N(X) and the resulting value, we filter the visible points from the total set of points. Below MATLAB code snippet illustrates this:

```
P=[9,14,11] % camera position

[nx,ny,nz] = surfnorm(x,y,z); %generate surface normal coordinates


X = x(:);
Y = y(:);
Z = z(:);
Nx = nx(:);
Ny = ny(:);
Nz = nz(:);
visible_points = [];%matrix to store visible points
points = [X,Y,Z];
N = [Nx,Ny,Nz];
for i=1:size(X,1)
    temp = points(i,:,:);
    diff = P -temp;

    if dot(diff,N(i,:,:))> 0 %condition to find point is visible
        visible_points = [visible_points;temp];
    end
end
```

3. Find areas to shade based on light direction
   Based on the light direction and surface normal for each point on the sphere, we use Lambertian's cosine law to compute the dot product and thus the intensity of light at that point. If intensity is less than zero, then we can assume that light doesn't reach that point and thus it can be shaded. We find all such points from the set of visible points.

```
light_dir = [33,29,44];
 shading_points = [];
for i=1:size(visible_points,1)
    intensity = dot(N(i,:,:),light_dir); %Lambertian dot product of normal and
light direction gives intensity at that point
    if intensity <=0
        point = visible_points(i,:,:);
        img_point = camera_matrix*point';
        img_point(1:2) = img_point(1:2)/img_point(3);
        shading_points = [shading_points;img_point(1),img_point(2)];
    end
end

plot(shading_points(:,1),shading_points(:,2),'k.'); %shade the sphere at
points based on intensity determined by Lambertian reflectance
hold off;
```

Finally we can see the rendered sphere onto the image in Figure 9. The shaded part is black.

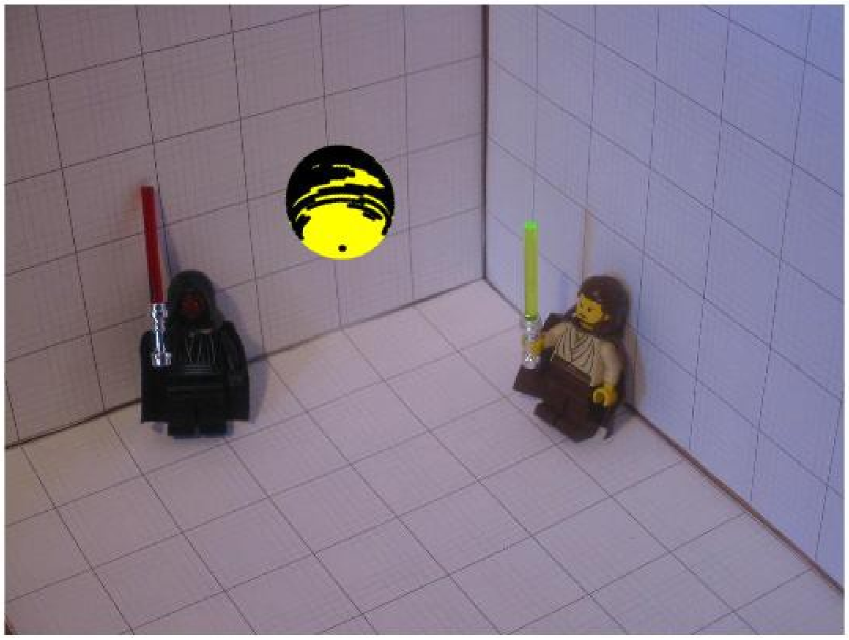

Figure 9: Rendered sphere with shading based on Lambertian reflectance

Part D: (Note D: the mentioned code snippets for Part D is in partd.m)

1. In figure 10, we have the 3x4 camera matrix obtained from Part B

$$M = \begin{bmatrix} -0.1391 & 0.0541 & -0.0224 & 0.8443 \\ 0.0319 & 0.0548 & -0.1401 & 0.4906 \\ -0.0000 & -0.0000 & -0.0000 & 0.0009 \end{bmatrix}$$

Figure 10: Camera matrix 3x4

Our goal is to determine the intrinsic camera matrix (3x3) K. It is of the form

$$K = \begin{bmatrix} \alpha_x & \gamma & u_0 \\ 0 & \alpha_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 11: Intrinsic camera parameters listed as part of the matrix

In this case, ϒ which denotes skew is assumed to be zero as per the question. We need to determine the remaining 4 parameters.

First, we need to determine a scaling constant for M as we assumed a unit norm for M but in actual, it is different. This constant can be derived from below equation

$$const = \sqrt{M(3,1)^2 + M(3,2)^2 + M(3,3)^2)}$$

$$M = M/const$$

Figure 12: Determining a scaling constant for M.

Then, for computing α_x and α_y we extract vectors m_1, m_2 and m_3 such that

```
m1 = M(1,1:3)';
m2 = M(2,1:3)';
m3 = M(3,1:3)';
```

We can then use vectors to determine α_x, α_y, u_o and v_o from equations in Figure 13.

$$u_o = m_1' * m_3$$

$$v_o = m_2' * m_3$$

$$\alpha_x = \sqrt{m_1' * m_1 - u_0^2}$$

$$\alpha_y = \sqrt{m_2' * m_2 - v_0^2}$$

Figure 13: Equations to determine focal lengths and principal point coordinates.

The resultant matrix K (3x3) is obtained after determining these values and is as showing in Figure 14.

$$K = \begin{bmatrix} 3278.8 & 0 & 1009.6 \\ 0 & 3432.7 & 653.1 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 14: Intrinsic camera matrix (K)

2. Finding the extrinsic camera matrix (X)

To find the 4x4 extrinsic matrix, we need to find the Rotation Matrix (R) and Translation vector (T) and combine them. Using the m_1, m_2 and m_3 vectors obtained in #1, we can find the 3 orthonormal rotation vectors using equations in Figure 15.

$$R(3,:) = s * M(3,1:3)$$
$$s = sign(M(3,4))$$
$$R(1,:) = s * (u_0 * M(3,1:3) - M(1,1:3))/\alpha_x;$$
$$R(2,:) = s * (v_0 * M(3,1:3) - M(2,1:3))/\alpha_y;$$
$$R(3,:) = s * M(3,1:3)$$

Figure 15: equations to obtain rotational vectors, s denotes whether camera is in the front/back

We also determine the Translation vector (t) similarly from equations in Figure 16:

$$T(3) = s * M(3,4)$$
$$T(1) = s * (u_0 * M(3,4) - M(1,4))/\alpha_x$$
$$T(2) = s * (v_0 * M(3,4) - M(2,4))/\alpha_y$$

Figure 16: Equations to obtain 3x1 Translation vector

We check det(R) is 1 to verify rotation is according to a right handed coordinate system. Thus, the extrinsic matrix (4x4) is shown in Figure 17 after obtaining the rotational and translation values.

$$X = \begin{bmatrix} 0.8182 & -0.5742 & -0.0289 & 0.6466 \\ -0.3104 & -0.4836 & 0.8184 & 0.7669 \\ -0.4839 & -0.6606 & -0.5739 & 21.1046 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 17: Extrinsic matrix (X) as a combination of R and T

3. Finding M again

   We use K, X and the 3x4 projection matrix to determine M again by multiplying them as shown in Figure 18.

$$M_{est} = K * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} * X$$

Figure 18: Estimating M again using the obtained matrices in #1 and #2

   We also reduce the M obtained here by the scaling constant used in #1 as the matrix obtained in Part B was of norm 1.

   Thus we obtain M_{est} as shown in Figure 19.

$$M_{est} = \begin{bmatrix} 0.4986 & -0.5795 & -0.1532 & 5.3243 \\ -0.3140 & -0.4753 & 0.5533 & 3.7306 \\ -0.0001 & -0.0002 & -0.0001 & 0.0048 \end{bmatrix}$$

Figure 19: Estimated camera matrix using Figure #18

4. Finding all the unknown parameters

We find the following parameters (intrinsic) from K:

- Focal length in horizontal units (α_x)
- Focal length in vertical units (α_y)
- Principal coordinates (centre of image)(u0,v0)

We find the following the parameters (extrinsic) from X:

- Position of camera (C)
- Orientation of camera (O)

From the K matrix obtained in #1 and using the equation in Figure, we can infer that

- Focal length in horizontal units = **3278.8**
- Focal length in vertical units = **3432.7**
- Principal point (u_0,v_o) = (1009.6,653.1)

From the X matrix obtained in #2, we can obtain the position and orientation of camera as follows:

Given a 3x3 Rotation matrix (R), the position of camera C is given by equation in Figure 20.

$$C = -R^T t$$

Figure 20: Equation to obtain camera position

Using this, we get the camera coordinates as per Figure 21.

$$C = \begin{bmatrix} 9.9218 \\ 14.6844 \\ 11.5036 \end{bmatrix}$$

Figure 21: Estimated camera position

The orientation(O) of a camera is given by Figure 22

$$O = R^T \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Figure 22: Equation to obtain orientation of camera

Using above equation we obtain the orientation as in Figure 23.

$$O = \begin{bmatrix} -0.4839 \\ -0.6606 \\ -0.5739 \end{bmatrix}$$

Figure 23: Estimate of the orientation of the camera

For reference, the code to compute these parameters is below:

```
M=camera_matrix;
% Since the norm of camera matrix M is equal to 1, we can calculate the
% scale factor
scaling_const=sqrt(M(3,1)^2 + M(3,2)^2 + M(3,3)^2);
% Scale the Matrix with the scale factor - as actual norm of M will vary
M = M / scaling_const;

%Assuming the camera is in front
s = sign(M(3,4));
% Translation along the Z direction
T(3) = s*M(3,4);

% Create a 3x3 Rotation matrix and fill it with zeros
R = zeros(3,3);

% Last row of the rotation matrix is the same as the first
% three elements of the last row of the calibration matrix
R(3,:)=s*M(3,1:3);

% Matrix M can be written as:
% M=( m1'    )
%   ( m2' m4)
%   ( m3'    )
% We can now calculate mi, where mi is a 3 element vector
m1 = M(1,1:3)';
m2 = M(2,1:3)';
m3 = M(3,1:3)';
m4 = M(1:3,4);

% Now, we can calculate the centres of projection u0 and v0 which will be
% centre of image
u0 = m1'*m3;
v0 = m2'*m3;
% Calculating the focal length in X and Y directions,
alpha=sqrt( m1'*m1 - u0^2 );
beta=sqrt( m2'*m2 - v0^2 );
% We can now calculate the first and second rows of the rotation matrix
R(1,:) = s*(u0*M(3,1:3) - M(1,1:3) ) / alpha;
R(2,:) = s*(v0*M(3,1:3) - M(2,1:3) ) / beta;

% We can also calculate the first and second elements of the Translation
% vector
T(1) = s*(u0*M(3,4) - M(1,4) ) / alpha;
T(2) = s*(v0*M(3,4) - M(2,4) ) / beta;
T = T';
translationVector=T;

%The rotation matrix R obtained so far may not be guaranteed to be
%orthonormal so using SVD to recompute an accurate estimate

[U,D,V] = svd(R);
R = U*V';
rotationMatrix=R;
```

```
temp = [R,T];
temp = [temp;0,0,0,1];
projection_mat = [1,0,0,0;0,1,0,0;0,0,1,0];
K = [alpha,0,u0;0,beta,v0;0,0,1];
M_est = K*projection_mat*temp;

C= -R'*T;%equation to obtain camera postion
O = R'*[0;0;1];%equation to obtain camera orientation
```