Aynur Süne - 21041

EE417 Post-Lab #9

```
%% Lab#9 Assignment starts here.

M = zeros(57,20);

for i= 1:1:19

        x2skew = skew(p2(:,i));
        element = x2skew*R*p1(:,i);

            M(3*i-2,i) = element(1);
            M(3*i-1,i) = element(2);
            M(3*i,i) = element(3);

end

for a = 1:1:19
   x2skew = skew(p2(:,a));
   element2 = x2skew*T;
   M(3*a-2,20) = element2(1);
   M(3*a-1,20) = element2(2);
   M(3*a,20) = element2(3);
end

[U,S,V] = svd(M'*M);
lambda = V(:,end);

points = zeros(3,19);

for b=1:1:19
    points(:,b)=lambda(b)*p1(:,b);
end

points = [points; ones(1,19)];

e = inv(Hc1)*points;

%% Plot the 3D points
figure
subplot(1,2,1)
plot3(P_W(1,:),P_W(2,:),P_W(3,:),'b.','MarkerSize',36)
axis equal
grid on
axis vis3d
xlabel('X')
ylabel('Y')
zlabel('Z')
title('Original World Points')


 subplot(1,2,2)
% Plot your reconstructed world points here.

plot3(e(1,:),e(2,:),e(3,:),'r.','MarkerSize',36)
axis equal
grid on
axis vis3d
xlabel('X')
ylabel('Y')
zlabel('Z')
title('Reconstructed World Points (Up to scale)')
```

We know that if the configuration is non-critical, the Euclidean structure of points and motion of the camera can be reconstructed up to a scale. Euclidean transformation of a point in one view to the second view is as follows:

$$\lambda_2 x_2 = R\lambda_1 x_1 + \gamma T$$

We can eliminate one of the scales to reduce number of unknowns, by multiplying both sides with its skew-symmetric form. In this case, λ_2 is eliminated with the multiplication of skew-symmetric form of x_2. We get the following equation:

$$\lambda_1^j \hat{x}_2^j R x_1^j + \gamma \hat{x}_2^j T = 0, \qquad j = 1, 2, ..., n$$

In order to find the unknown parameters, we can write this equation as follows:

$$\mathbf{M}\Lambda = \begin{bmatrix} (\hat{x}_2^1 R x_1^1)_{3\times 1} & 0_{3\times 1} & \cdots & 0_{3\times 1} & (\hat{x}_2^1 T)_{3\times 1} \\ 0_{3\times 1} & (\hat{x}_2^2 R x_1^2)_{3\times 1} & & \vdots & (\hat{x}_2^2 T)_{3\times 1} \\ \vdots & & \ddots & 0_{3\times 1} & \vdots \\ 0_{3\times 1} & \cdots & 0_{3\times 1} & (\hat{x}_2^n R x_1^n)_{3\times 1} & (\hat{x}_2^n T)_{3\times 1} \end{bmatrix} \begin{bmatrix} \lambda_1^1 \\ \lambda_1^2 \\ \vdots \\ \lambda_1^n \\ \gamma \end{bmatrix} = 0$$

Implementation in code starts with constructing this M matrix. Since elements on diagonal are 3x1 and we need n elements where n is the number of points, we need 19x3=57 rows because we have 19 points. We also need 19 columns for each point and an extra column for translation. M matrix's size becomes 57x20. We initialize the M matrix and fill it with the necessary elements in a for loop. One element of the diagonal is 3x1 so resulting M matrix will look different. This 3x1 element is the multiplication of skew-symmetric form of x_2 (points of view 2) which is 3x3, rotation matrix which is 3x3 and x_1 (points of view one) which is 3x1; gives the diagonal element. Last column is filled with another for-loop, consists of skew-symmetric form of a point multiplied by translation, gives 3x1 result for each of 19 points. Part of the resulting M matrix is as follows:

M

57x20 double

	1	2	3	4	5	6	7	8	9	10	11	12
1	0.0263	0	0	0	0	0	0	0	0	0	0	0
2	-0.5467	0	0	0	0	0	0	0	0	0	0	0
3	-0.0768	0	0	0	0	0	0	0	0	0	0	0
4	0	0.0211	0	0	0	0	0	0	0	0	0	0
5	0	-0.5523	0	0	0	0	0	0	0	0	0	0
6	0	-0.0614	0	0	0	0	0	0	0	0	0	0
7	0	0	0.0137	0	0	0	0	0	0	0	0	0
8	0	0	-0.5595	0	0	0	0	0	0	0	0	0
9	0	0	-0.0440	0	0	0	0	0	0	0	0	0
10	0	0	0	1.6920e-...	0	0	0	0	0	0	0	0
11	0	0	0	-0.5132	0	0	0	0	0	0	0	0
12	0	0	0	-0.0097	0	0	0	0	0	0	0	0
13	0	0	0	0	-0.0066	0	0	0	0	0	0	0
14	0	0	0	0	-0.5183	0	0	0	0	0	0	0
15	0	0	0	0	0.0081	0	0	0	0	0	0	0

As it can be seen from here, It is different from a diagonal matrix in results. In order to find the resulting column vector (lambdas) Λ that equates $M\Lambda=0$, we apply SVD decomposition to M^TM. We get [U,S,V] matrices in result and last column of the V matrix gives Λ. Last element of this vector gives us the Gamma value, which is the scale of the translation T. Lambdas are the depth value of the points, we multiply the pixel points of the first view by their corresponding depth values, adding ones to last row instead of gamma and convert them to

the world points by multiplying the results with inverse of Hc1 where Hc1 is composed of extrinsic parameters R (rotation) and T (translation) of first view. Result is as follows:

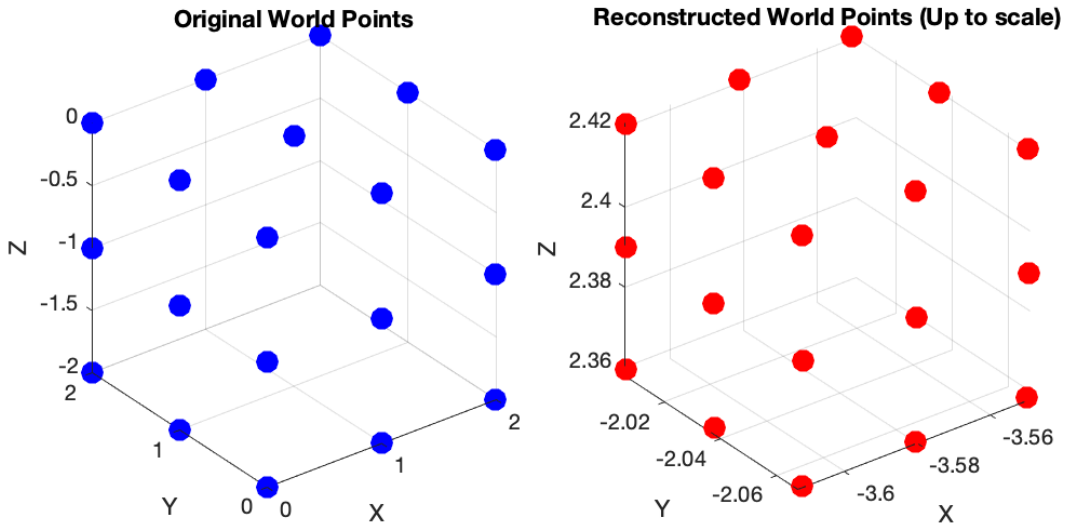


Figure below shows the lambda vector that consist of depth values of each point and gamma value at last row. When we compute Sum of Squares Error (SSD) of each point, result is 746.2319 which is a really high number.

lambda

20x1 double

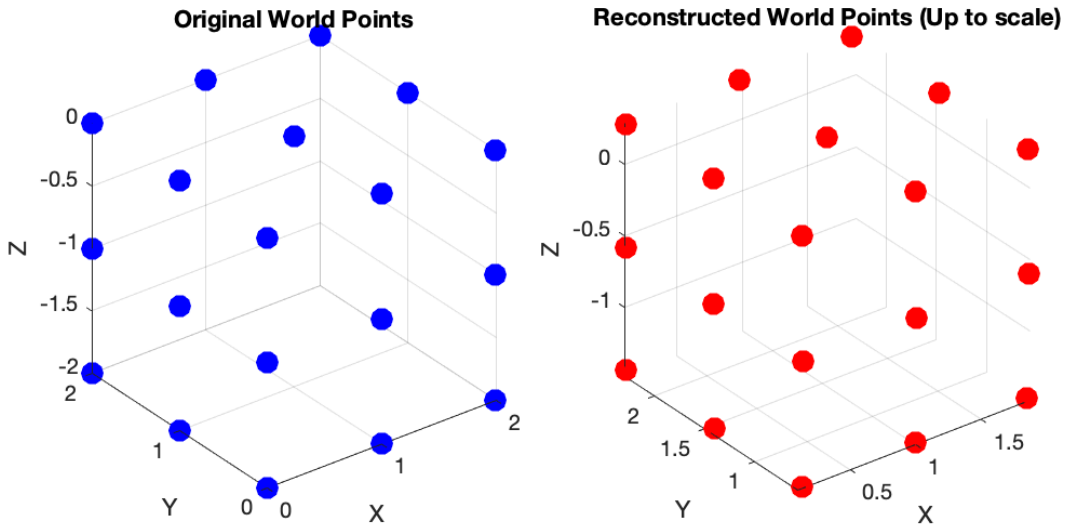
	1
1	0.2121
2	0.2000
3	0.1875
4	0.2270
5	0.2148
6	0.2030
7	0.2420
8	0.2301
9	0.2179
10	0.2103
11	0.2326
12	0.2258
13	0.2483
14	0.2404
15	0.2630
16	0.2225
17	0.2452
18	0.2348
19	0.2578
20	0.1135

This gamma, however, is not the true scale of the translation matrix, it is up to scale, means that real gamma (scale) value is some multiple of the estimated lambda and lets' say this multiple is k. In order to find the true scale, we can divide true translation matrix with our estimated translation (Tc2c1 ./ T) to approximately see in what scale our estimated translation matrix values differ from true translation. This operation is not entirely true since we cannot know the true translation of a camera in real life, but it is given in codes for proper use. The result gives us two different gamma values since second element of the true translation is 0 and will not give us any scale value, but first and third elements will give. I tried for both and also with the average of these values. As a result, I obtained three different k values and multiplied each with lambda vector separately. These lambda values are multiplied by pixel points and converted to world points as in lab. Implementation is as follows:

```
points = zeros(3,19);

EstimatedGamma = lambda(end);
Gamma = Tc2c1./T;

Gamma1 = Gamma(1);
Gamma3 = Gamma(3);

k1 = Gamma1/EstimatedGamma;
k3 = Gamma3/EstimatedGamma;
k2 = (Gamma1+Gamma3)/(2*EstimatedGamma);
disp(k1);
disp(k3);
disp(k2);

lambda1 = lambda*k1;
lambda3 = lambda*k3;
lambda2 = lambda*k2;

for b=1:1:19
    points(:,b)=lambda1(b)*p1(:,b);
end

points = [points; ones(1,19)];

e = inv(Hc1)*points;

err = zeros(3,19);
for i=1:1:3
    for j=1:1:19
        err(i,j) = (P_W(i,j) - e(i,j))^2;
    end
end

ssd = sum(sum(err));
disp(ssd);
```

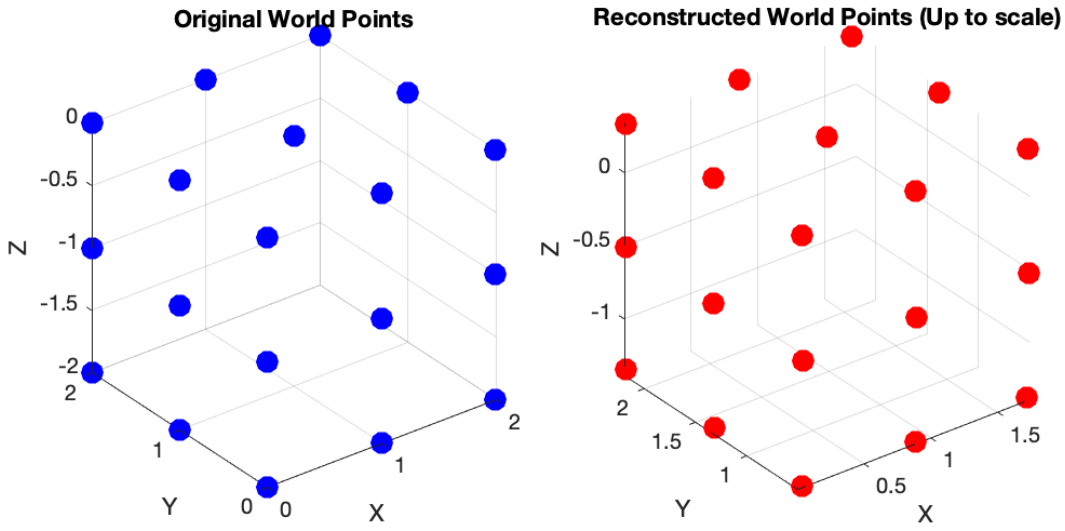
I computed sum of squares error to compare which gamma (or k) value will give a better result.

<u>For k1 = 28.0136</u> (Obtained from first gamma value)



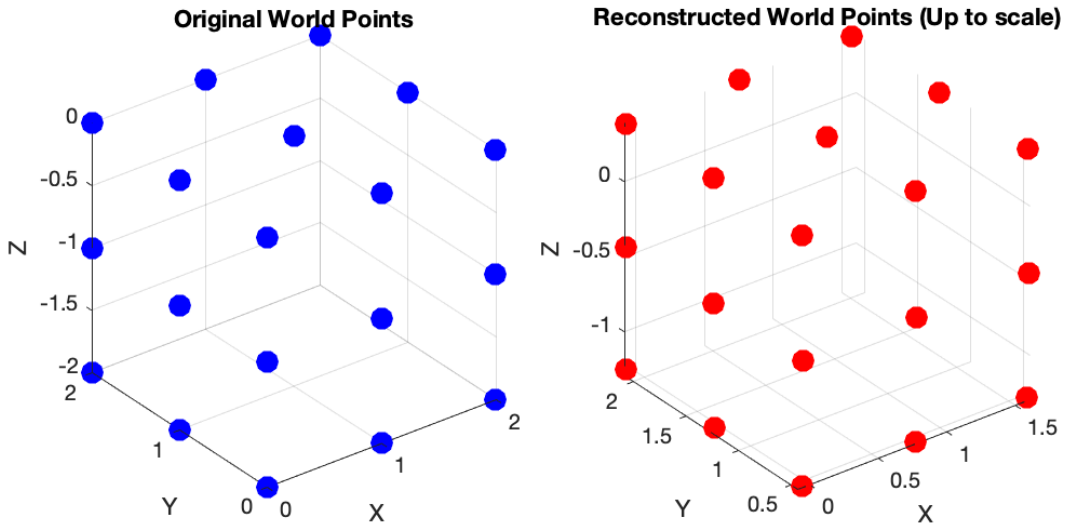
SSD = 7.6426

<u>For k2 = 27.3438</u> (Obtained from average of first and second gamma value)



SSD = 7.7660

<u>For k3 = 26.6740</u> (Obtained from second gamma value)



SSD = 8.8143

All three values are very similar to each other and approximately a hundred times better than the results we obtained from our estimated gamma during lab. If we must select the best, first k value we calculated with first gamma value gave the best result with an SSD value equals to 7.6426 which is smallest.

<u>Addition:</u> New function called skew is written to convert point to skew-symmetric form. Implementation is as follows:

```
function  [skewed] = skew(mat)

skewed = [0 -mat(3) mat(2);mat(3) 0 -mat(1);-mat(2) mat(1) 0];

end
```