AV Coursework

Part 1

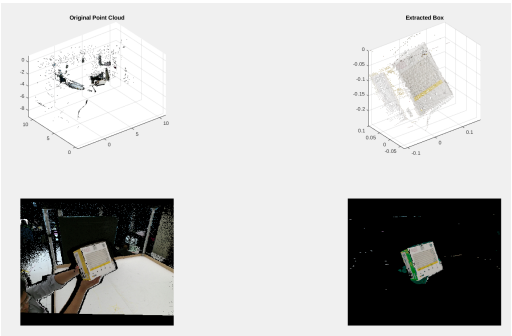
1) We first make a loop to go through each individual frame one by one - extracting the point cloud for each frame.

2) We then remove all the points that are far away from the box (which is centered near [-0.71,-0.30,0.81]). To do this we transpose the box to [0,0,0] and then remove all the points lying beyond 0.25 (our threshold distance for this box).

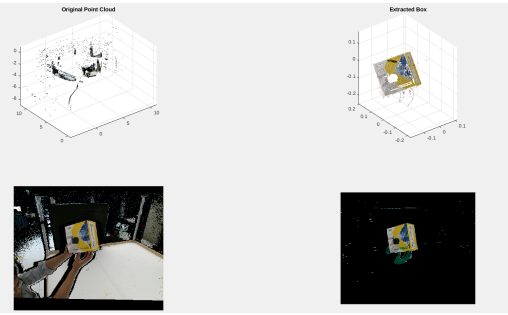
3) We then update the point cloud and proceed to remove the hand. We remove the hand by analyzing the color matrix of the point cloud and removing all the points which are within the color range of the hand. For simplicity, no hsv filter was applied and the color range chosen (hues of brown/skin color) was applied in RGB values itself.

4) Now all the unnecessary (NAN) points are removed and we need to remove noise. We create a binary image for the point cloud and use *bwareopen()* function on it - removing the small objects (noise). We then cast our point cloud matrices like the binary mask using *bsxfun()* function. With this we get our finalized extracted box.

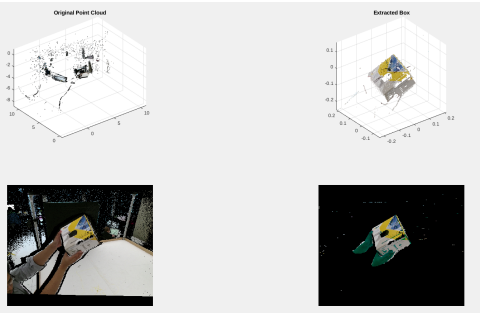
Some examples are shown in Fig1.



(a)



(b)



(c)

Figure 1: Box Extraction

Part 2

The planes are fitted using embedded matlab function *pcfitplane()*. Only planes with more than 1000 points are kept. We get 20 frames with two planes, and no frame with 3 planes because of the area restriction.

Some of the fitted plane normals point towards the inside of the box, to make sure all the normals point outside (Fig2), the following procedure is adopted:

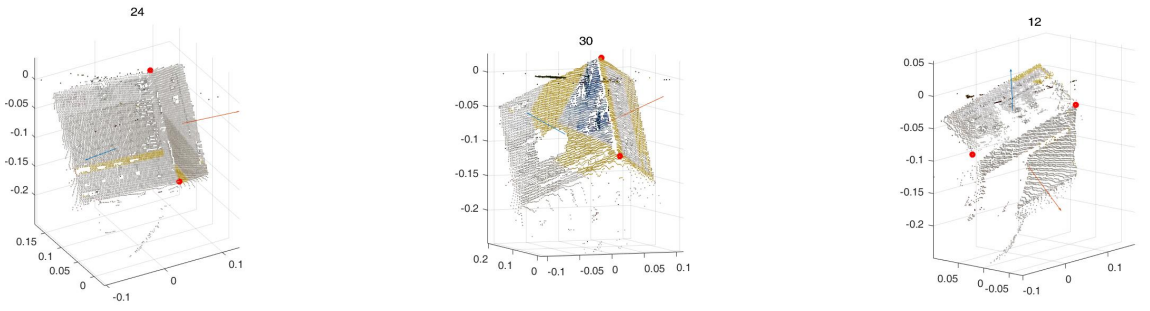
For two planes, estimate the centre point of each plane, make a vector pointing from one centre to another, and check the dot product of the vector and the calculated plane normals. Reverse the normal if the product is negative (or positive, depending on the direction of the vector).

Part 3

From *pcfitplane()* we can obtain the plane equation and maximum/minimum value of x, y, z in locations. The corners are calculated in the following way:

First, calculate the normal of the intersection line between two planes using the cross product of the plane normals. Second, set x to zeros and calculate the corresponding y, z by solving the equation of the two planes, to get a point on the intersection line. Combining these two gives us the full definition of the line. Third, using the limit value of x, y, z to get all possible corner points in the two planes (there may be more than 8 points because some points share the same limit value along certain axes). Plug these points into the equation of the intersection line, and find two points that are both close to the line and far apart from each other.

Plotting corners on the point clouds we can see that most corners are found correctly. In some frames the corners are off position due to the incompleteness of the plane, or due to the noise points outside the main body of the plane. (Fig2)



(a) (b) (c)

(d) (e) (f)

Figure 2: Corners and outward pointed normals

Part 4 and Part 5

The pose estimation is done in the following steps:

1) In the 20 frames with two planes, manually check which side (out of 6 box sides) the planes belong to, and add labels.

2) Using matlab function *pcregrigid()* to match a certain side across the frames where the side can be seen, to obtain the transformation matrices; then apply the matrices to whole frames to get the whole scene. Transformation is done by *pctransform()* and cloud point fusing is done by *pcmerge()*. It is possible that the sides are matched between two frames but the plane normals are pointed towards opposite direction, so the whole scene can't be matched right (Fig3), we check these cases by computing the dot product of transformed plane normals (which should always pointed toward the outside), and discard this match if the product is negative.

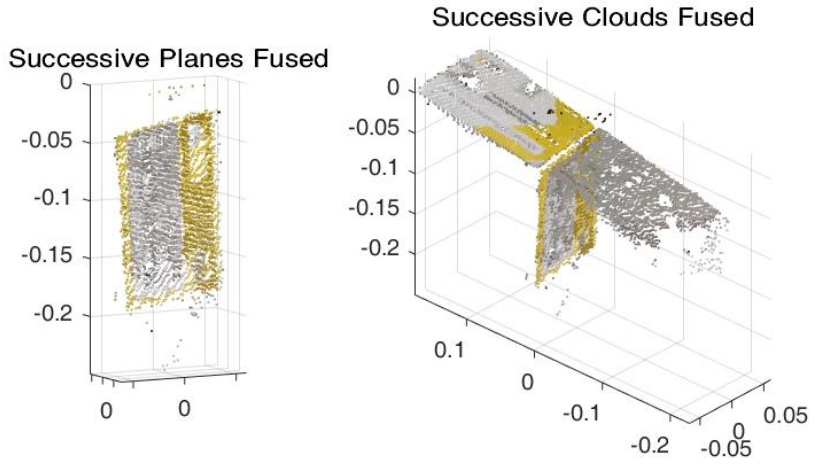


Figure 3: Plane normals get matched in opposite direction

3) If we don't get all six sides in the final scene, chose another side, and repeat step 2.

4) Do pose estimation between the result of step 2&3 and merge them together, we can get the whole box.

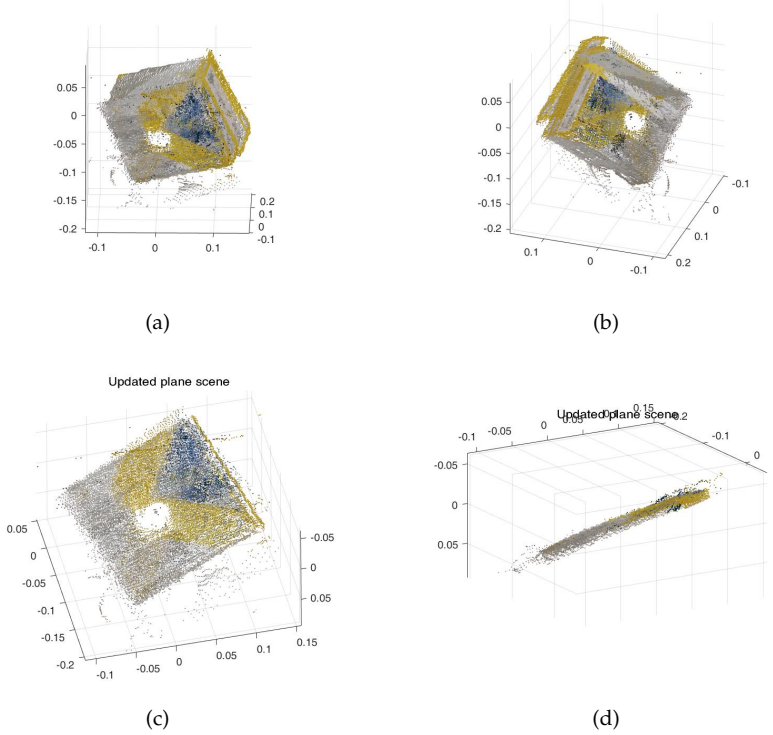
By experiment using the front side and the right side works best. This is due to several reasons:

1) They are large and clearly seen in many frames. Front side can be clearly seen in 6 frames and right side can be clearly seen in 7 frames.

2) Because the ICP algorithm used by *pcregrigid()* doesn't consider colour, using a rectangle will work better than a square. But due to the colour and image preprocessing, the front side is slightly incomplete, which gives it a more unique shape that can be taken use of.

3) Whether the side is extracted whole or not, it need to remain consistent among frames. The up side can get a unique shape due to the lightning-shaped yellow patch, but the wholeness of the side is very inconsistent.

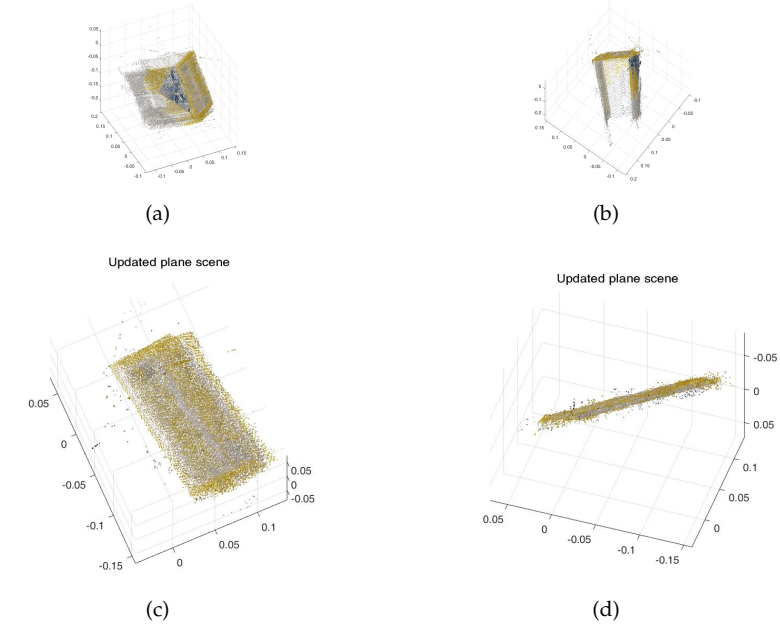
After fusing on each side we get a 4-sided box (Fig4, Fig5), and luckily these two 4-sided boxes cover all the six different sides. Fusing them together gives us the whole box in (Fig6).



(a) (b)

(c) (d)

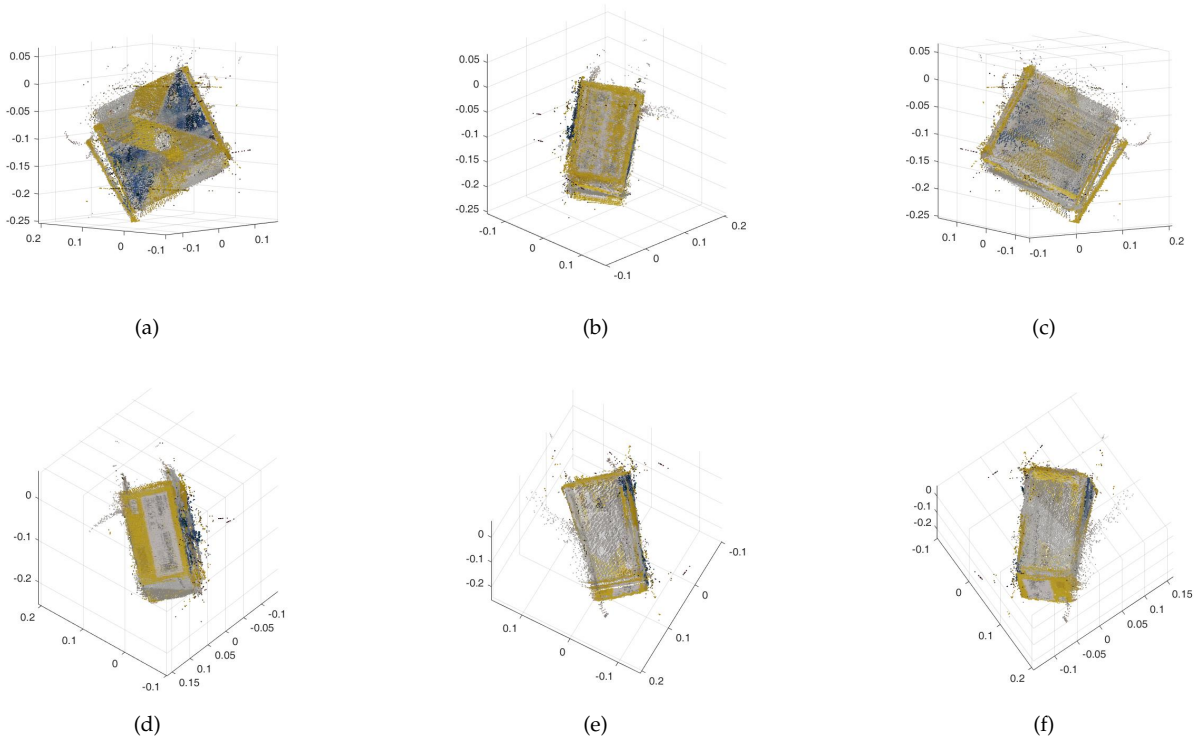
Figure 4: Fused 4-sided box and large plane (using front side)



(a) (b)

(c) (d)

Figure 5: Fused 4-sided box and large plane (using right side)



(a) (b) (c)

(d) (e) (f)

Figure 6: Fused Whole box - looking from 6 angles

The box is not entirely right:

1) The front side is rotated 180 degree between two 4-side boxes, and opposite side can lay on top of each other. This is because in our case the right side is matched with a rotation of 180 degrees in one of the consecutive frames (Fig7), causing the opposite sides of the box to switch position. A possible remedy is to deliberately remove the small white patch on this side during preprocessing and use it as an anchor.

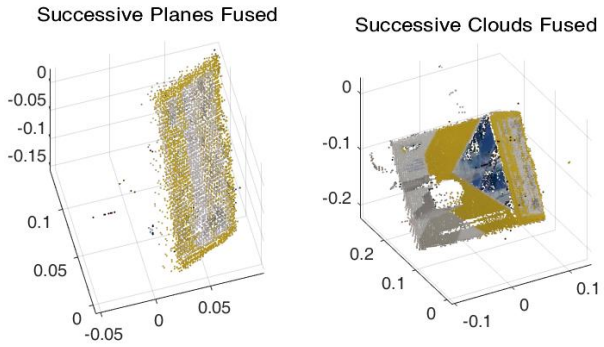


Figure 7: The side is matched with a rotation of 180 degree

2) The normals are not entirely perpendicular. This is because sometimes the rotation is slightly off because the sides in two frames have different areas. This can be due to the quality of the extracted box

cloud, or when the plane is the second largest in a frame, a slim edge line of it is included in the first plane. A possible remedy is to set the distance restriction when fitting plane smaller.

Part 6

1) To calculate the average angular error, we first calculate the unit normal by taking the dot product of the normals and dividing by their magnitude. This gives us the unit normal error for a pair of angles. We then use two loops to calculate the error for every normal with every normal of a frame number greater than it - as mentioned in the assignment description. This result recalculated every time the large plane is fused on to the final image. The result is plotted for all 50 frames.

2) Similarly, to find the average corner displacement error, we first find the closest fused corner of the large frame just added and take that to be the same corner. We verified this by viewing the successive frames manually in Q3.5. We then find the difference in the positions (error) and calculate the average using two loops in a similar pattern to the previously plotted normals.

3) Finally, to find the RMS error, we run *pcfitplane()* to refit a plane onto our fused large plane (fused image) every time we add a new frame where that large plane exists. The MATLAB function has an inbuilt feature that returns the mean square error of the plane fitting as per the formula specified in the question. Hence, we take square root of the MSE value to get RMSE and then plot this for different frame numbers.

The results are shown in Fig8.

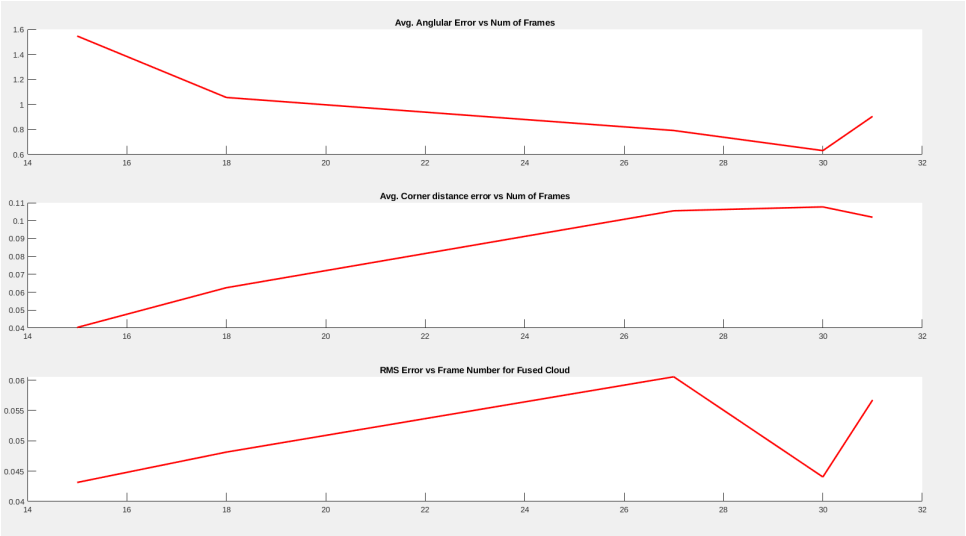


Figure 8: Angular error, Corner distance and RMS error. (using front side)

Appendix

Fig9 shows the image index, number of 3D box pixels found, number of planes found, and average plane fit RMS of the planes found in that frame.

ImageIdx	pixelNum	planeNum	Average RMS	ImageIdx	pixelNum	planeNum	Average RMS
1	0	0	NaN	26	4951	1	0.00276
2	0	0	NaN	27	9193	2	0.00141
3	41	0	NaN	28	8944	1	0.00168
4	3348	1	0.00237	29	7263	1	0.00306
5	6011	1	0.00186	30	9644	2	0.00176
6	7255	2	0.00208	31	7017	2	0.00183
7	1203	0	NaN	32	4459	1	0.00190
8	1373	0	NaN	33	6254	2	0.00147
9	9298	2	0.00167	34	4126	1	0.00348
10	8330	1	0.00178	35	2424	0	NaN
11	8247	2	0.00175	36	5966	2	0.00209
12	6573	2	0.00199	37	3779	1	0.00210
13	2388	0	NaN	38	4659	1	0.00183
14	1133	0	NaN	39	7151	2	0.00139
15	10562	2	0.00220	40	7780	2	0.00139
16	5645	1	0.00289	41	1376	0	NaN
17	6901	1	0.00203	42	3918	1	0.00163
18	7897	2	0.00257	43	7150	2	0.00162
19	6025	2	0.00269	44	3404	1	0.00206
20	1400	0	NaN	45	1009	0	NaN
21	9708	2	0.00154	46	9993	2	0.00173
22	9589	2	0.00226	47	10458	2	0.00153
23	7783	1	0.00215	48	9667	1	0.00252
24	10626	2	0.00142	49	9168	1	0.00241
25	3857	1	0.00225	50	0	0	NaN

Figure 9: Information of extraction

Code

Extracting box and finding planes.

```
%% Load the training data
clear all
close all
box = load('assignment_1_box.mat');
box = box.pcl_train;
flag=1;
checkframe=1;
%% Uncomment to load the test file
%box = load('assignment_1_test.mat');
%box = box.pcl_test;
totalcount=0;
% count the number of recognised planes in one frame
successFrames = {};
allFrames = {};
```

```
numSuccessFrames = 0;
% display the points as a point cloud and as an image
count_pixels = [];
count_planes = [];
avrg_rms = [];
for frameNum = 1:length(box) % Reading the 50 point-clouds
    frameNum
        % extract a frame
        rgb = box{frameNum}.Color; % Extracting the colour data
        point = box{frameNum}.Location; % Extracting the xyz data

    pc = pointCloud(point, 'Color', rgb);
    figure(1);
    subplot(2,1,1);
    pcshow(pc);
    title('Original Point Cloud');

        %Extract points close to box%%%%%%%%%%%%%%%%%%%%%%
        BoxPos = [-0.71,-0.30,0.81];
    for i=1:length(point)
        point(i,:)=((point(i,:)-BoxPos));
        if(abs(point(i,1))>0.25|| abs(point(i,2))>0.25|| abs(point(i,3))>0.25)
            point(i,:)=NaN;
            rgb(i,:)=NaN;
        end
    end
        pc = pointCloud(point, 'Color', rgb); % Creating a point-cloud variable

        %Remove hand%%%%%%%%%%%%%%%%%%%% hand is 100=-20, 80+-20, 40 +-10
        r = rgb(:,1);
        g = rgb(:,2);
        b = rgb(:,3);
        for i=1:length(r)
                if((r(i)<150 && r(i)>40)&&(g(i)<130 && g(i)>35)&&(b(i)<100 ...
        && b(i)> 0))%rgb color range of values for hand
                        % r(i)=NaN;
                        % g(i)=NaN;
                        % b(i)=NaN;
                        point(i,:)=NaN;
                        rgb(i) = NaN;
                end
        end
        %Remove points that don't belong to box
        idx = find(isnan(point(:,1)));
        point(idx,:) = [];
        rgb(idx,:) = [];

        %Remove excess noise%%%%%%%%%%%%%%%%%%%%%%%%
        BW = im2bw(rgb,0.1);
        BW2 = bwareaopen(BW, 10);
        new_rgb_box= bsxfun( @times , rgb , cast( BW2, 'like' , rgb) ) ;
        new_point_box= bsxfun( @times , point , cast( BW2, 'like' , point) ) ;
    %new_rgb_box=rgb; new_point_box=point;

        %Remove points that don't belong to box
        idx = find(sum(new_rgb_box,2)==0);
        new_point_box(idx,:) = [];
        new_rgb_box(idx,:) = [];
```

```
%     imshow(BW2);
%     imshow(new_rgb_box);
        pc1 = pointCloud(new_point_box, 'Color', new_rgb_box);

          figure(1);
    subplot(2,1,2);
         pcshow(pc1);
         title('Extracted Box');

    figure(3);
    clf
         pcshow(pc1);
         title('Plane Data with Normals and Corners');
    hold on
    allFrames{frameNum}.box = pc1;
    %check if there is enough points left

    count_pixels = [count_pixels;pc1.Count];
        if(length(new_point_box) < 3000)
        count_planes = [count_planes;0];
        avrg_rms = [avrg_rms;nan];
                continue
        end
%%
        % fit three planes, and only store when it's large enough
        maxDistance = 0.01;
    count = 0;
    %% Find Plane 1

        [model1,inlierIndices,outlierIndices,rmse1] = pcfitplane(pc1,...
                        maxDistance);
        plane1 = select(pc1,inlierIndices);
        remainPtCloud = select(pc1,outlierIndices);
    if plane1.Count > 1000
        count = count + 1; totalcount=totalcount+1;
        boxPlanes{count} = plane1;
        boxNormals{count} = model1.Parameters;

      origin1 = [mean(plane1.XLimits),mean(plane1.YLimits),mean(plane1.ZLimits)];
      direction1 = model1.Parameters(1:3);
    end

    thresh=0.01;
    mergeSize=0.001;


    %% Find Plane 2

        [model2,inlierIndices,outlierIndices,rmse2] = pcfitplane(remainPtCloud,...
                                maxDistance);
        plane2 = select(remainPtCloud,inlierIndices);
        remainPtCloud = select(remainPtCloud,outlierIndices);



    if plane2.Count > 1000
        count = count + 1; totalcount=totalcount+1;
```

```
        boxPlanes{count} = plane2;
        boxNormals{count} = model2.Parameters;
        origin2 = [mean(plane2.XLimits),mean(plane2.YLimits),mean(plane2.ZLimits)];
        direction2 = model2.Parameters(1:3);
    end

    %% Find Plane 3

        [model3,inlierIndices,outlierIndices] = pcfitplane(remainPtCloud,...
                                maxDistance);
        plane3 = select(remainPtCloud,inlierIndices);
        remainPtCloud = select(remainPtCloud,outlierIndices);
    if plane3.Count > 1000
        count = count + 1; totalcount=totalcount+1;
        boxPlanes{count} = plane3;
        boxNormals{count} = model3.Parameters;
        origin = [mean(plane3.XLimits),mean(plane3.YLimits),mean(plane3.ZLimits)];
        direction = model3.Parameters(1:3);
        if dot(origin - BoxPos, direction) < 0
            direction = -direction;
        end
        quiver3(origin(1),origin(2),origin(3),...
            direction(1),direction(2),direction(3),...
            0.1);
    end

    %% Count the Number of Planes

    count_planes = [count_planes;count];
    if count == 0
        avrg_rms = [avrg_rms;nan];
    elseif count == 1
        avrg_rms = [avrg_rms;rmse1];
    else
        avrg_rms = [avrg_rms;mean([rmse1,rmse2])];
    end

    if (count <= 1)
        continue;
    end

%%  check fit results and view extracted planes seperately
%           figure(frameNum);
%     subplot(1,2,1)
%           pcshow(plane1)
%     title([num2str(frameNum),'-plane1'])
%            subplot(1,2,2)
%           pcshow(plane2)
%     title([num2str(frameNum),'-plane2'])
%     drawnow
%           figure
%           pcshow(plane3)
%           title('Third Plane')
%           figure
%           pcshow(remainPtCloud)
%           title('Remaining Point Cloud')

%% Reverse Normals towards outer faces and display results
```

```
   if (dot(origin1 - origin2, direction1) < 0)
        direction1 = -direction1;
    end
    if (dot(origin2 - origin1, direction2) < 0)
        direction2 = -direction2;
    end
%
    figure(3);
         pcshow(pc1);
    hold on

     quiver3(origin1(1),origin1(2),origin1(3),...
        direction1(1),direction1(2),direction1(3),...
        0.1);
    quiver3(origin2(1),origin2(2),origin2(3),...
        direction2(1),direction2(2),direction2(3),...
        0.1);

    N1 = model1.Parameters;
    N2 = model2.Parameters;
    N = cross(N1(1:3),N2(1:3));
    N = N/norm(N);
    xx = 0;
    yy = (N2(4)*N1(3) - N1(4)*N2(3))/ N(1);
    zz = (N1(4)*N2(2) - N2(4)*N1(2))/ N(1);
    corners = checkCorner(N,xx,yy,zz,plane1,plane2);
     scatter3(corners(:,1),corners(:,2),corners(:,3),'filled','r');
     hold off;

     %% Save Data for Q4-6 Implementation to work with

    oneFrame.box = pc1;
    oneFrame.numPlanes = count;
    oneFrame.planes = boxPlanes;
    oneFrame.planeParameters = boxNormals;
    oneFrame.checkframe = checkframe;
    oneFrame.frameNum=frameNum;

    numSuccessFrames = numSuccessFrames + 1;
    successFrames{numSuccessFrames} = oneFrame;


    %planetmp=pcmerge(plane1, plane2,0.001);
    Planespc{numSuccessFrames} = oneFrame;
    %Planespc{numSuccessFrames}.box=pcmerge(planetmp, plane3,0.001);
    Planespc{numSuccessFrames}.box=pcmerge(plane1, plane2,0.001);
    Planespc{numSuccessFrames}.corners = corners;
    Planespc{numSuccessFrames}.plane1=plane1;
    Planespc{numSuccessFrames}.plane2=plane2;
    Planespc{numSuccessFrames}.frameNum = frameNum;
    Planespc{numSuccessFrames}.norms{1} = direction1;
    Planespc{numSuccessFrames}.norms{2} = direction2;
    %pause();
end
% save('successFrames.mat','successFrames');
% save('allFrames.mat','allFrames');
```

```
save('Planespc.mat','Planespc');
```

Find corners

```
function corners = checkCorner(N,xx,yy,zz,plane1,plane2)
possibleCorners = [];
plane = plane1;
p = plane.Location;
pX = plane.XLimits;
pY = plane.YLimits;
pZ = plane.ZLimits;
idx = find((p(:,1) == pX(1)) | (p(:,1) == pX(2)));
possibleCorners = [possibleCorners;p(idx,:)];
idx = find((p(:,2) == pY(1)) | (p(:,2) == pY(2)));
possibleCorners = [possibleCorners;p(idx,:)];
idx = find((p(:,3) == pZ(1)) | (p(:,3) == pZ(2)));
possibleCorners = [possibleCorners;p(idx,:)];

plane = plane2;
p = plane.Location;
pX = plane.XLimits;
pY = plane.YLimits;
pZ = plane.ZLimits;
idx = find((p(:,1) == pX(1)) | (p(:,1) == pX(2)));
possibleCorners = [possibleCorners;p(idx,:)];
idx = find((p(:,2) == pY(1)) | (p(:,2) == pY(2)));
possibleCorners = [possibleCorners;p(idx,:)];
idx = find((p(:,3) == pZ(1)) | (p(:,3) == pZ(2)));
possibleCorners = [possibleCorners;p(idx,:)];

error = [];
for i = 1:length(possibleCorners)
    pnt = possibleCorners(i,:);
    error_xy = abs((pnt(1) - xx) / N(1) - (pnt(2) - yy) / N(2));
    error_xz = abs((pnt(1) - xx) / N(1) - (pnt(3) - zz) / N(3));
    error = [error; max(error_xy,error_xz)];
end

idx_min = find(error == min(error));
idx_min = idx_min(1);
error(idx_min) = max(error);
corner1 = possibleCorners(idx_min,:);
for i = 1:length(error)
    idx = find(error == min(error));
    idx = idx(1);
    corner2 = possibleCorners(idx,:);
    if sqrt(sum((corner2 - corner1) .^2)) > 0.05
        break
    else
        error(idx) = max(error);
    end
end
corners = [corner1;corner2];
```

Add sides information manually

```
clear all
close all
load('Planespc.mat')
```

```
% plot and check sides
% for i = 1:length(Planespc)
%     oneFrame = Planespc{i};
%     figure(i);
%     subplot(1,2,1)
%     pcshow(oneFrame.plane1)
%     title([num2str(oneFrame.frameNum),'-plane1'])
%      subplot(1,2,2)
%     pcshow(oneFrame.plane2)
%     title([num2str(oneFrame.frameNum),'-plane2'])
%     drawnow
% end

% set sp1 and sp2 manually
%  fig 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
sp1 = [1,3,3,0,4,4,2,3,3,3,4,4,2,1,0,0,0,0,3,3];
sp2 = [4,0,0,0,0,2,0,2,0,0,0,2,4,2,0,0,0,2,2,0];
for i = 1:length(Planespc)
    Planespc{i}.side_plane1 = sp1(i);
    Planespc{i}.side_plane2 = sp2(i);
end
save('Planespc.mat','Planespc');

%     if ismember(frameNum, [])
%         checkframe = 1;
%     elseif ismember(frameNum, [])
%         checkframe = 2;
%     else
%         checkframe = 0;
%     end
```

Estimate pose and fuse scene

```
clear all
close all

load('Planespc.mat');
successFrames=Planespc;

side_used = 4; %%The side which is to be used as the large plane for reference

gridSize = 0.0015;
gridStep = 0.02;
mergeSize = 0.001;

ctr = 1;
n_corner = 1;
xprevrms=0;
yprevrms=0;
xprevcorner=0;
yprevcorner=0;
xprevangle=0;
yprevangle=0;
accumTform = [];
rmse=0;

found_ref = 0;
for i = 1:length(successFrames)%% Go through all the verified frames
```

```
% find first frame with large side_used and initialize values with it
if found_ref == 0
    if successFrames{i}.side_plane1 == side_used
        planeScene = successFrames{i}.plane1;
        planeNorm = successFrames{i}.norms{1};
    elseif successFrames{i}.side_plane2 == side_used
        planeScene = successFrames{i}.plane2;
        planeNorm = successFrames{i}.norms{2};
    else
        continue
    end
    ptCloudScene = successFrames{i}.box;
    found_ref = 1;

    % initialise variables
    normals{ctr}=successFrames{i}.planeParameters{1};
    corners{n_corner} = successFrames{i}.corners(1,:);
    moving = pcdownsample(ptCloudScene, 'gridAverage', gridSize);
    moving_plane = pcdownsample(planeScene, 'gridAverage', gridSize);
    moving_planeNorm = planeNorm;

    figure
    hAxes = pcshow(ptCloudScene, 'VerticalAxis','Y', 'VerticalAxisDir', 'Down');
    title('Updated world scene')
    % Set the axes property for faster rendering
    hAxes.CameraViewAngleMode = 'auto';
    hScatter = hAxes.Children;

    % Visualize the plane scene.
    figure
    hAxes2 = pcshow(ptCloudScene, 'VerticalAxis','Y', 'VerticalAxisDir', 'Down');
    title('Updated plane scene')
    % Set the axes property for faster rendering
    hAxes2.CameraViewAngleMode = 'auto';
    hScatter2 = hAxes2.Children;
    continue
end
if successFrames{i}.side_plane1 == side_used
    planeCurrent = successFrames{i}.plane1;
    ptCloudCurrent = successFrames{i}.box;
    planeNorm = successFrames{i}.norms{1};
elseif successFrames{i}.side_plane2 == side_used
    planeCurrent = successFrames{i}.plane2;
    ptCloudCurrent = successFrames{i}.box;
    planeNorm = successFrames{i}.norms{2};
else
    continue
end
% Use previous moving point cloud as reference.
fixed = moving;
fixed_plane = moving_plane;
fixed_planeNorm = moving_planeNorm;

moving = pcdownsample(ptCloudCurrent, 'gridAverage', gridSize);
moving_plane = pcdownsample(planeCurrent, 'gridAverage', gridSize);
moving_planeNorm = planeNorm;
```

```
% Apply ICP registration.
tform = pcregrigid(moving_plane, fixed_plane, 'Metric','pointToPlane',...
    'Extrapolate', true,...
    'Tolerance',[0.00001, 0.00005],...
    'MaxIteration',300);

moving_planeNorm_new = [moving_planeNorm,1] * tform.T;% For the fused plane
moving_planeNorm_new = moving_planeNorm_new(1:3);

if dot(moving_planeNorm_new, fixed_planeNorm) < 0
% Ignore the frame if fused in reverse (normals in opp direction)
    moving = fixed;
    moving_plane = fixed_plane;
    moving_planeNorm = fixed_planeNorm;
    continue;
end

planePrevNext=pctransform(planeCurrent, tform);
planePrevNext=pcmerge(fixed_plane, planePrevNext, mergeSize);
%Merged Successive Frames

ptCloudPrevNext=pctransform(ptCloudCurrent, tform);
ptCloudPrevNext=pcmerge(fixed, ptCloudPrevNext, mergeSize);
%Merged Successive Large Planes

if isempty(accumTform)
    accumTform = tform;
else
    accumTform = affine3d(tform.T * accumTform.T);
    %Update the accumulated transform
end
planeAligned = pctransform(planeCurrent, accumTform);
ptCloudAligned = pctransform(ptCloudCurrent, accumTform);

% Update the world scene.
planeScene = pcmerge(planeScene, planeAligned, mergeSize);
ptCloudScene = pcmerge(ptCloudScene, ptCloudAligned, mergeSize);

if(1)
    %%Caluclate errors in normals, corners and RMSE of fitted plane

    %% NORMALS ERROR CALCULATION
     ctr=ctr+1;
     normals{ctr}=successFrames{i}.planeParameters{1}*accumTform.T;
     tmpctr=0;
     totalangle=0;
     for m=1:length(normals)-1
        for n=(m+1):(length(normals))
        angle=acos( dot(normals{m} , normals{n})) ;%%IN Radians
        totalangle=totalangle+angle; tmpctr=tmpctr+1;
        end
     end
     avgangle=totalangle/tmpctr;

    figure(300);
    subplot(3,1,1);hold on;
    if(xprevangle~=0 &&yprevangle~=0)
    plot([successFrames{i}.frameNum,xprevangle], ...
```

```
    [avgangle,yprevangle], 'r-', 'LineWidth', 2);
    end
    xprevangle=successFrames{i}.frameNum; yprevangle=avgangle;
    title('Avg. Anglular Error vs Num of Frames');


 %% Corner Error Average Calculation
    n_corner = n_corner + 1;
    corner1 = successFrames{i}.corners(1,:);
    corner2 = successFrames{i}.corners(2,:);
    dis1 = sum((corner1 - corners{1}).^2);
    dis2 = sum((corner2 - corners{1}).^2);
    if dis1 < dis2
        corners{n_corner}=corner1;
    else
        corners{n_corner}=corner2;
    end
     tmpctr=0;
     totaldist=0;
     for m=1:length(corners)-1
        for n=(m+1):(length(corners))
            totaldist = totaldist + sqrt(sum((corners{m}-corners{n}).^2));
            tmpctr = tmpctr + 1;
        end
     end
     avgdist=totaldist/tmpctr;
 subplot(3,1,2);hold on;
    if(xprevcorner~=0 &&yprevcorner~=0)
    plot([successFrames{i}.frameNum,xprevcorner], ...
    [avgdist,yprevcorner], 'r-', 'LineWidth', 2);
    end
    xprevcorner=successFrames{i}.frameNum; yprevcorner=avgdist;
    title('Avg. Corner distance error vs Num of Frames')

%% Plane Fitting and RMS

    [model1,inlierIndices,outlierIndices,mse] = pcfitplane(planeScene,0.01);
    rmse=sqrt(mse);
    subplot(3,1,3);hold on;
    if(xprevrms~=0 &&yprevrms~=0)
    plot([successFrames{i}.frameNum,xprevrms], ...
    [rmse,yprevrms], 'r-', 'LineWidth', 2);
    end
    xprevrms=successFrames{i}.frameNum; yprevrms=rmse;
    title('RMS Error vs Frame Number for Fused Cloud');
end

%Visualize the successive frame stitching
figure;
subplot(1,2,1)
pcshow(planePrevNext);
title('Successive Planes Fused');
subplot(1,2,2)
pcshow(ptCloudPrevNext);
title('Successive Clouds Fused');


% Visualize the world scene.
```

```
    hScatter.XData = ptCloudScene.Location(:,1);
    hScatter.YData = ptCloudScene.Location(:,2);
    hScatter.ZData = ptCloudScene.Location(:,3);
    hScatter.CData = ptCloudScene.Color;
    drawnow('update')
     % Visualize the plane scene.
    hScatter2.XData = planeScene.Location(:,1);
    hScatter2.YData = planeScene.Location(:,2);
    hScatter2.ZData = planeScene.Location(:,3);
    hScatter2.CData = planeScene.Color;
    drawnow('update')
end
save('box_side2.mat','ptCloudScene');
```

Fuse two 4-sided box into one

```
close all;
clear all;
load('box_side4.mat')
s4 = ptCloudScene;
load('box_side2.mat')
s2 = ptCloudScene;
% figure
% pcshow(s4)
% figure
% pcshow(s2)
mergeSize=0.001;
tf=pcregrigid(s4,s2,'Metric','pointToPlane',...
        'Extrapolate', true,...
        'Tolerance',[0.00001, 0.00005],...
        'MaxIteration',3000);
    s4=pctransform(s4,tf);
pcshow(pcmerge(s2,s4,mergeSize));
```