Advanced Vision Assignment 2

Edward Stevinson
Hoi Hei Ma
University of Edinburgh
s1567581@sms.ed.ac.uk
s1578393@sms.ed.ac.uk

March 17, 2016

1 Introduction

The assignment involved registering and fusing range data from 16 views captured by a Kinect depth sensor. From this fused data the structures in the views can be extracted. For code mentioned in the report please see the appendix.

2 Background Plane Extraction

The first stage is to extract the background plane so it can be discarded, leaving the item points behind (belonging to 3 balls and an object). The script that achieves this is `main.m`. The code iterates over each of the views, taking the data from `pcl_cell` (the RGB and XYZ values) and creates a point cloud by reshaping the matrix and removing all points with $x = y = z = 0$. The function `backgroundsub()` then removes the data points that belong to the background from the point cloud. It does this by using `select_starting_patch()` to randomly select a point from the point cloud and iterates over all the remaining points, building a matrix of all the points within a `threshold_distance` of the selected point. If there are more than 10 points within this threshold distance then `fitplane()` (provided code) fits a plane to these points, returning the equation of this infinite plane and the least square fitting error. When a plane has been found with a suitable fitting error the equation of the plane is returned to `backgroundsub()` which uses `getallpoints()` to grow the background plane in each of the views. If a point both:

- lies within a threshold distance of the plane, (`abs(pnt'*plane) < DISTTOL`) ; and
- lies within a threshold distance of the centre of the points that have already been categorised as on the plane, (`oldlist_center - pointlist(i,1:3) < max_dist`)

then the point is declared on the background plane and removed from the point cloud. This leaves is with `fg_cloud`, a point cloud of the non-background points.

The forground data are further cleaned by throwing away any outliers, before k-means clustering is then implemented to split the data into 4 clusters. If the standard deviation of the mean distances between items is found to be within a range (recorded from observation) then this plane is accepted as the background plane. Finally the point cloud is saved as a .mat file.

The equation of an infinite plane is in the form:

$$a \cdot \mathbf{x} + b \cdot \mathbf{y} + c \cdot \mathbf{z} = d \tag{1}$$

The equation of the background plane is recorded as $[0.00, -0.57, 0.82, -749.30]^T$. Figure 1 shows the range data from view 11, whilst Figure 2 shows the range data of this view with the background range data removed.

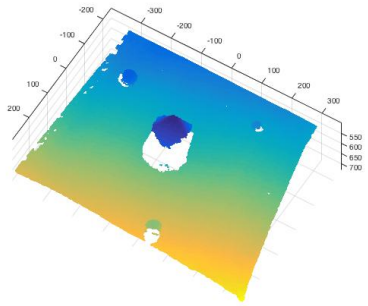

Figure 1: Range data view 11

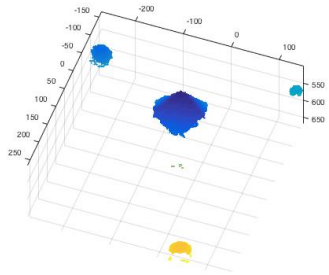

Figure 2: With background plane removed

3 Sphere Extraction

In the views there are three balls. The `main2.m` script acts to extract the data that belong to these balls. It iterates over each of the points performing k-means clustering (using the squared Euclidean distance) to partition the point cloud data into 4 clusters. As the object is the largest object the mode index is used to determine which cluster belongs to the object and not the balls, and this data separated.

It then uses the hue value of the hsv values of each point to determine which ball each point belongs to. Outliers are then removed before the provided `spherefit()` function is used to fit a sphere to the 3D data. This returns the centre point of each ball in each view. Figure 3 shows the detected ball points in the first view, with the centres marked by a black cross.

4 Fuse the XYZ data

The data from the three balls is now used to register and fuse the XYZ data from all the views. View number 2 is chosen as the baseline view that all the views are to be registered to.

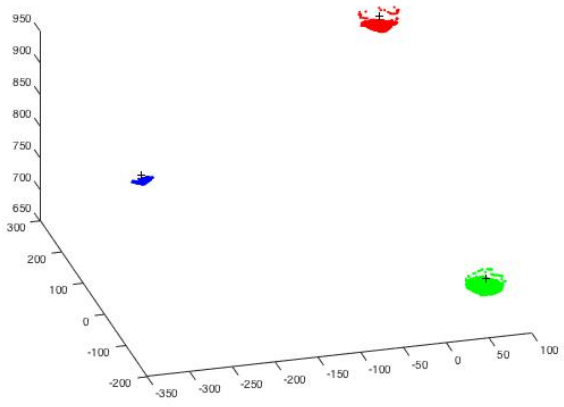

Figure 3: Detected ball points in view 1

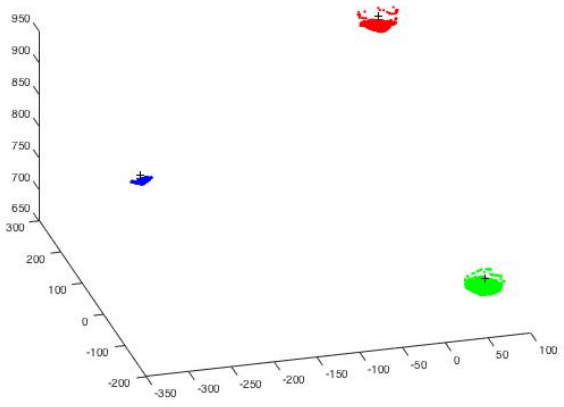

Figure 4: Detected ball points in view 3

The function `estPose()` is uses singular value decomposition to find the optimal rotation. It finds the covariance matrix, and uses singular value decompostition to break the covariance into the left singular vectors, singular values, and right singular vectors. The rotation is then defined as:

$$H = \sum_{i=1}^{N} (P_A^i - centroid_A)(P_B^i - centroid_B)^T \tag{2}$$

$$[U, S, V] = svd(H) \tag{3}$$

$$R = VU^T \tag{4}$$

`estPose()` then checks whether the 'reflection case' is returned, and if so recomputes R. It finally calculates the translation, using the following equation:

$$t = -R \times centroid_A \times centroid_B \tag{5}$$

`main3.m` then rotates and translates the points in each view, before adding them to the `fuse_obj` matrix. These points are shown in Figure 5.

The quality of the registration can be partially evaluated by computing the normal vector of the floor in each rotated view and evaluating how parallel they are. Ideally these normals

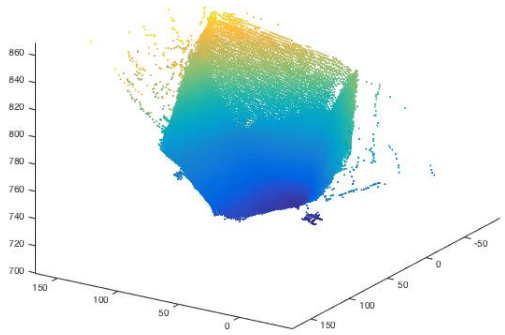

Figure 5: Fused range data

Table 1: caption

should all be parallel. However, our registration was not perfect which lead to noisy data, with an average 16 degree difference between normals.

5 Extract the planes of the object

The next part tackles extracting the 9 planes of the object from the point cloud data (this code is found in `main4.m`). It first removes some noise using the `pcdenoise()` function provided in MATLAB, before extracting the planes using `extract_planes()`. This again selects a random point, and attempts to fit a plane to it and its surrounding points. If this plane has a small error then all the points on the plane are determined using the `get_all_points()` function. This is similar to that used earlier, but in addition uses RGB values so only points that are similar in colour are included (using the `within_range()` function). Having fitted a plane to the points, it removes the fitted points from the point cloud, saving the remaining points as `remaining_point_cloud` and using these to find the next plane. Figure 6 shows the point cloud with a plane having been extracted.

The equations of the nine planes are shown in the following table:

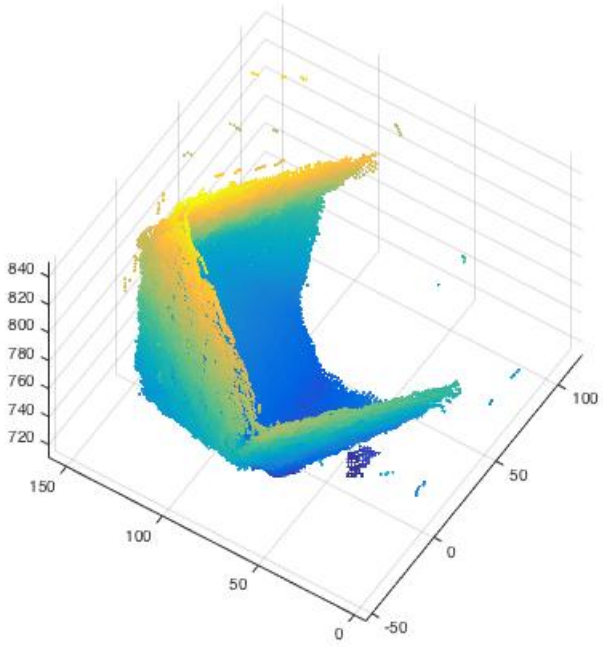

Figure 6: Fused range data with plane extracted

-0.8492	0.4971	0.1781	-213.9971
0.5103	0.8016	0.3115	-370.7114
0.5355	0.7041	0.4663	-379.8958
0.0151	-0.4906	0.8713	-592.1585
-0.7681	-0.4601	0.4453	-237.4682
-0.1682	0.3739	0.9121	-680.8882
0.7975	-0.1216	0.5909	-409.2544
-0.8443	0.4370	0.3100	-209.5638
0.1932	-0.9740	0.1183	41.6623

Some difficulty was encountered due to the noise introduced by some points being badly registered. This noise meant that on some runs, planes were accepted that did not line up to those of the object. This lead to problems with the next iteration as the data left from the previous iteration no longer resembled the correct shape. This resulted in incorrect models being created in the next question. Further work would ensure a more reliable method of selecting the correct planes. Here, the equations of the suitable planes (to be used in the next question) were noted following a successful run of the script.

6 Build a 3D model of the object

One method of drawing the model was, given the plane equations, to use our `planes_intersect_point()` function to calculate the location of each of the vertices. `draw_patches()` then utilised MATLABs patch capabilities to draw the model. The ideal image of the model should be similar to that shown in Figure 7, whilst figure 8 shows the model drawn from our plane equations.

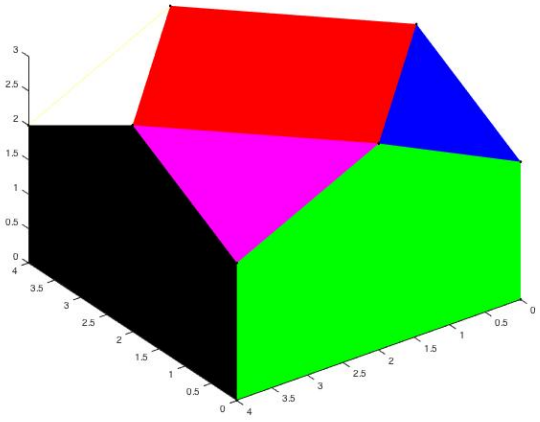

Figure 7: Ideal object model

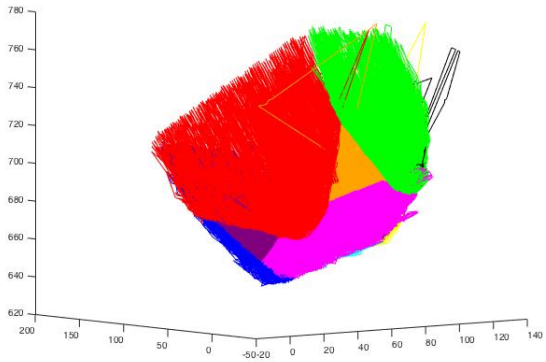

Figure 8: Our object model

The angle between the outward facing normals of each of these patches gives an indication of the accuracy of the plane equations. These angles are calculated using equation (6).

$$arccos(\frac{n_1 \cdot n_2}{\sqrt{a_1^2 + b_1^2 + c_1^2} \cdot \sqrt{a_2^2 + b_2^2 + c_2^2}}) \tag{6}$$

where

$$n_1 = (a_1, b_1, c_1) \; and \; n_2 = (a_2, b_2, c_2) \tag{7}$$

The following table contains a matrix showing the angles between the normals of our model (the order corresponds to the order of the equations listed previously).

0	10.5	55.2	89.9	90.2	86.2	52.8	121.9	122.3
10.5	0	64.6	87.3	89.5	96.6	59.1	128.0	130.9
55.2	64.5	0	123.2	115.7	46.7	65.5	100.0	67.3
89.9	87.3	123.2	0	10.3	94.2	57.8	59.0	128.2
90.2	89.5	115.7	10.3	0	84.0	51.2	51.0	120.2
86.2	96.6	46.7	94.2	84.0	0	53.8	53.3	54.2
52.8	59.1	65.5	57.8	51.2	53.8	0	69.2	108.0
121.9	128.0	100.0	59.0	51.0	53.3	69.2	0	69.3
122.3	130.9	67.3	128.2	120.2	54.2	108.0	69.3	0

7 Appendix

main.m:

```
%%
% Script for extracting and removing the ground plane from the image data
%%

% Set output format
format bank
% Colour array
mcolor = {'g', 'r', 'b', 'y', 'm', 'w', 'k'};
% Load data
load('av_pcl.mat');
% Clear current figure
clf
fg_clouds = {};

% Iterate over each view
for f = 1 : 16
    R = pcl_cell{f};
    % Multiply XYZ points?
    R(:,:,4:6) = R(:,:,4:6) * 1000;
    [L,W,~] = size(R);
    % Create depth cloud (307200-by-6) i.e. each point and its RGB & XYZ values
    depth_cloud = reshape(R, L*W, 6);

    % remove 0 row
    depth_cloud = depth_cloud(logical(depth_cloud(:,4))  ...
        & logical(depth_cloud(:,5)) & logical(depth_cloud(:,6)),:);
```

```
    accepted = false;
    while ~accepted
        % Get point cloud without background
        fg_cloud = backgroundsub(depth_cloud);

        % Remove ouliners
        [Row, ~] = size(fg_cloud);
        % Mean of X,Y and Z
        centroid = mean(fg_cloud(:,4:6));
        stdv = sum(std(fg_cloud(:,4:6)),2);
        filter = sum(abs(fg_cloud(:,4:6) - repmat(centroid, Row, 1)),2) ...
            < (repmat(stdv, Row, 1) * 3.0);
        fg_cloud = fg_cloud(filter',:);

        % k-means
        [idx, C, sumd, D] = kmeans(fg_cloud(:,4:6), 4, 'Replicates', 3);
        distanceMatrix = squareform(pdist(C));
        stdvDist = std(mean(distanceMatrix))
        if stdvDist >= 41 && stdvDist <= 45
            accepted = true;
        end
    end
    fg_clouds{f} = fg_cloud;
end
```

backgroundsub.m:

```
%%
% Function which returns the points not on background plane
%%
function [remaining] = backgroundsub(depth_cloud)

    [L,~] = size(depth_cloud);
    while (L > 100000)

        [L,~] = size(depth_cloud);
        % Select a patch to fit a plane to
        [oldlist, plane] = select_patch(depth_cloud);

        % Grow plane until break clause reached
        while (true)

            % Get points in cloud that are on plane with oldlist
            [newlist, depth_cloud] = getallpoints(plane, oldlist, depth_cloud, L);
```

```
            [Nold, ~] = size(oldlist);
            [Nnew, ~] = size(newlist);

            % If plane is growing, continue to grow
            if Nnew > Nold + 50
                [plane, ~] = fitplane(newlist(:,4:6));
                oldlist = newlist;
            else % Else return plane and points on plane
               [plane, fit] = fitplane(newlist(:,4:6))
               remaining = depth_cloud;
               size(remaining);
              break;
            end
        end
    [L,~] = size(depth_cloud);
end
end
```

select_patch.m:

```
%%
% Function to find a candidate planar patch
%%

function [fitlist, plane] = select_starting_patch2(point_cloud)

[num_points] = size(point_cloud,1);
tmpnew = zeros(num_points,6);
tmprest = zeros(num_points,6);

% pick a random point until a successful plane is found
success = 0;
while (~success)
    % Pick a random point out of the point cloud
    idx_point = floor(num_points*rand) + 1;
    % XYZ coordinates of random point
    idx_point_xyz = point_cloud(idx_point, 1:3);
    % RGB values of index point
    idx_point_RGB = point_cloud(idx_point, 4:6);
    % find points in the neighborhood of the given point
    threshold_distance = 6;
    % Initialise counts
    fitcount = 0;
    restcount = 0;
```

```
    for i = 1 : num_points

        dist = norm(point_cloud(i,1:3) - idx_point_xyz);
        % How many points are within threshold distance of selected point
        % and of roughly the same colour
        if (dist < threshold_distance) && (within_range(50, 50,
        point_cloud(i, 4:6), idx_point_RGB))
            fitcount = fitcount + 1;
            tmpnew(fitcount,:) = point_cloud(i,:);
        else
            restcount = restcount + 1;
            tmprest(restcount,:) = point_cloud(i,:);
        end
    end

    % If there are a certain amount of points nearby...
    if fitcount > 10
        % Fit a plane to the points near the randomly selected point
        [plane, resid] = fitplane(tmpnew(1:fitcount,1:3));
        % Is the fitting error below a threshold?
        if resid < 5
            fitlist = tmpnew(1:fitcount,:);
            return
        end
    end
end
```

get_all_points.m:

```
%%
% Function that selects all points in pointlist P that fit the plane
% and are within TOL of a point already in the plane (oldlist)
% Modified from Bob Fisher
%%

function [newlist,remaining] = get_all_points(plane, oldlist, pointlist, NP)

  point_xyz = ones(4,1);
  point_rgb = ones(3,1);
  [number_in_pointlist,~] = size(pointlist);
  [number_in_oldlist,~] = size(oldlist);
  DISTTOL = 5; % Threshold distance          % Bigger?
  PLANETOL = 10;                            % ...?
  tmpnewlist = zeros(NP,6);                 % ...?
  % initialize fit list
```

```
  tmpnewlist(1:number_in_oldlist,:) = oldlist;
  % initialize unfit list
  tmpremaining = zeros(NP,6);
  % Initialise counts
  countnew = number_in_oldlist;
  countrem = 0;
  % Calculate mean colour of old list
  mean_val = mean(oldlist);
  mean_colour = mean_val(4:6);
  %% Changed here
  oldlist_center = mean(oldlist(:,1:3));
  dist2center = max(sum(abs(oldlist(:,1:3) - ...
  repmat(oldlist_center, number_in_oldlist,1)),2));
  max_dist = (dist2center * 30);

  for i = 1 : number_in_pointlist
    point_xyz(1:3) = pointlist(i,1:3);
    point_rgb(1:3) = pointlist(i,4:6);
    notused = 1;

    % Does point lie within threshold of plane?
    if (abs(point_xyz'*plane) < DISTTOL) && (norm(oldlist_center - P(i,1:3)) ...
    < max_dist) % && (within_range(50, 50, mean_colour, point_rgb))
      % see if an existing nearby point already in the set
      for k = 1 : number_in_oldlist
        % Is the point under question far away from any point already in the set?
        if (norm(oldlist(k,:) - pointlist(i,:)) < PLANETOL)
          countnew = countnew + 1;
          tmpnewlist(countnew,:) = pointlist(i,:);
          notused = 0;
          break;
        end
      end
    end

    if (notused)
      countrem = countrem + 1;
      tmpremaining(countrem,:) = pointlist(i,:);
    end

  end

  newlist = tmpnewlist(1:countnew,:);
  remaining = tmpremaining(1:countrem,:);
end
```

main2.m:

```
%%
% Assignment 2 - Question 2
% Extract and describe the 3 spheres
%%

% Load foreground point clouds
load fg.mat;

% Array of empty matrices for each ball and object
balls = cell(1,16);
obj = cell(1,16);

% Iterate over each view
for f = 1 :16

    % Current cloud
    cloud = fg_clouds{f};

    good_cluster = false;
    while (~good_cluster)

        % K-means clustering to separate objects
        % C are centroid locations
        [idx, C, sumd, D] = kmeans(cloud(:,4:6), 4, 'Replicates', 3);
        distanceMatrix = squareform(pdist(C));
        stdvDist = std(mean(distanceMatrix));
        if stdvDist >= 41 && stdvDist <= 45
            good_cluster = true;
        end
    end

    figure(f)
    clf
    hold on
    %     pcshow(cloud(:,4:6))
    %     plot3(C(:,1),C(:,2),C(:,3),'k+')

    disp(f)
    %% For each cluster change the HSV values to RGB values
    hsvOfClust = zeros(numel(unique(idx)), 1);
    for i = 1 : numel(unique(idx))
        clust = cloud(idx == i,:);
        color = mean(clust(:,1:3));
        color = color / 255;
```

```
        hsvcolor = hsv2rgb(color);
        hsvOfClust(i) = hsvcolor(2);
    end

    [B,I] = sort(hsvOfClust);

    object_cloud = cloud(idx == mode(idx),:);
    obj{f} = object_cloud;

    % Create cell for each ball
    ball_clouds = cell(1,3);
    j = 0;
    for i = 1 : size(I,1)
        if I(i) == mode(idx)
            continue;
        end
        j = j + 1;
        ball_cloud = cloud(idx == I(i),:);
        ball_clouds{j} = ball_cloud;
    end

    % remove ouliner
    for i = 1 : size(ball_clouds,2)
        cloud = ball_clouds{i};
        distanceMatrix = squareform(pdist(cloud(:,4:6)));
        filter = mean(distanceMatrix) < mean(mean(distanceMatrix)) * 1.5;
        cloud = cloud(filter',:);
        ball_clouds{i} = cloud;
    end

    %% Fit a sphere to the 3D data
    center = zeros(3,3);
    [center(1,:),~] = sphereFit(ball_clouds{1}(:,4:6));
    [center(2,:),~] = sphereFit(ball_clouds{2}(:,4:6));
    [center(3,:),~] = sphereFit(ball_clouds{3}(:,4:6));
    balls{f} = center;

    plot3(center(:,1), center(:,2), center(:,3), 'k+')
    plot3(ball_clouds{1}(:,4), ball_clouds{1}(:,5), ball_clouds{1}(:,6),'r.')
    plot3(ball_clouds{2}(:,4), ball_clouds{2}(:,5), ball_clouds{2}(:,6),'g.')
    plot3(ball_clouds{3}(:,4), ball_clouds{3}(:,5), ball_clouds{3}(:,6),'b.')
end
save balls.mat balls
save obj.mat obj
```

estPose.m:

```
%%
% Function that calculates rotation and translation matrices
% Inputs: A: point cloud, B: baseline view
%%

function [ R, t ] = estPose(A, B)

    centroid_A = mean(A);
    centroid_B = mean(B);

    N = size(A,1);

    %% Find the optimal rotation using singular value decomposition
    % Get covariance H
    H = (A - repmat(centroid_A, N, 1))' * (B - repmat(centroid_B, N, 1));
    % Perform svd
    [U,S,V] = svd(H);
    % Get rotation
    R = V*U';

    % Check whether it is the reflection case
    if det(R) < 0
        disp('Reflection detected\n');
        V(:,3) = V(:,3) * -1;
        R = V*U';
    end
    % Calculate translation
    t = -R*centroid_A' + centroid_B';

end
```

main3.m:

```
%%
% Assignment 2 - Question 3: Registration
%%
% Load required data
load balls.mat
load fg.mat
load obj.mat

% Choose base
base = balls{2};

fuse_obj = [];
```

```
for f = 1: size(balls,2)
    % Get xyz data of current view
    cloud = fg_clouds{f}(:,4:6);
    [L,~] = size(obj{f});
    % Get the rotation matrix and the translation
    [R, t] = estPose(balls{f}, base);
    % Translate the object
    trans_obj = R * obj{f}(:,4:6)' + repmat(t, 1, L);
    trans_obj = trans_obj';
    % Keep colour info
    trans_obj = [trans_obj, obj{f}(:,1:3)];
    % Add fused data from this view to fused matrix
    fuse_obj = [fuse_obj;trans_obj];
end
save fuse.mat fuse_obj
```

main4.m:

```
%% Assignment 2 - Questions 4 and 5 (plane extraction and modelling)

% Clear figures
clf
% Load point cloud
load('fuse.mat');
point_cloud_noisy = fuse_obj;
% Colour list
colour_list = {'g', 'r', 'b', 'y', 'm', 'w', 'k'};

%% Remove some noise
% Turn point cloud into object
point_cloud_noisy_object = pointCloud(point_cloud_noisy(:,1:3));
% Remove noise from this object
[~, ~, outlier_indices] = pcdenoise(point_cloud_noisy_object, 'Threshold', 1);
% Remove ouliers from point cloud
point_cloud = removerows(point_cloud_noisy,'ind',outlier_indices);
% Get size of point cloud
[number_of_points, ~] = size(point_cloud);
% Plot this object
pcshow(point_cloud(:, 1:3))

%% Extract planes
[planes, points_matrix] = extract_planes(point_cloud, colour_list);

%% Calculate Points
ground_plane = [0, -0.57, 0.82, -749.3];
```

```
all_planes = [planes; ground_plane];
get_vertices(all_planes)
%% Draw Patches
draw_patches(vertices)
```

extract_planes.m:

```
%% Function which extracts planes

function [plane_list, points_matrix] = extract_planes(point_cloud, colours)

    [num_points_in_cloud, ~] = size(point_cloud);
    % Store plane parameters
    plane_list = zeros(20,4);
    remaining_point_cloud = point_cloud;

    % Iterate for each plane
    for plane_no = 1:9

        % Select a random small surface patch
        [old_list, plane] = select_starting_patch2(remaining_point_cloud);

        %% Grow patch
        while (true)

            % Find neighbouring points in plane
            still_growing = false;
            [list_of_points_on_plane, remaining] = getallpoints(plane, ...
             old_list, remaining_point_cloud, num_points_in_cloud);
            remaining_point_cloud = remaining;
            % Store
            [number_points_plane, ~] = size(list_of_points_on_plane);
            [length_oldlist, ~] = size(old_list);
            % If plane still growing...
            if (number_points_plane > length_oldlist + 40)
                [plane, ~] = fitplane(list_of_points_on_plane(:, 1:3));
                old_list = list_of_points_on_plane;
            else % Record plane and points
                [plane, ~] = fitplane(list_of_points_on_plane(:, 1:3));
                plane_list(plane_no,:) = plane';
                points_matrix{plane_no} = list_of_points_on_plane;
                break
            end
        end
        [num_points_in_cloud,~] = size(remaining_point_cloud);
```

```
        pcshow(remaining_point_cloud(:,1:3))
    end
end
```