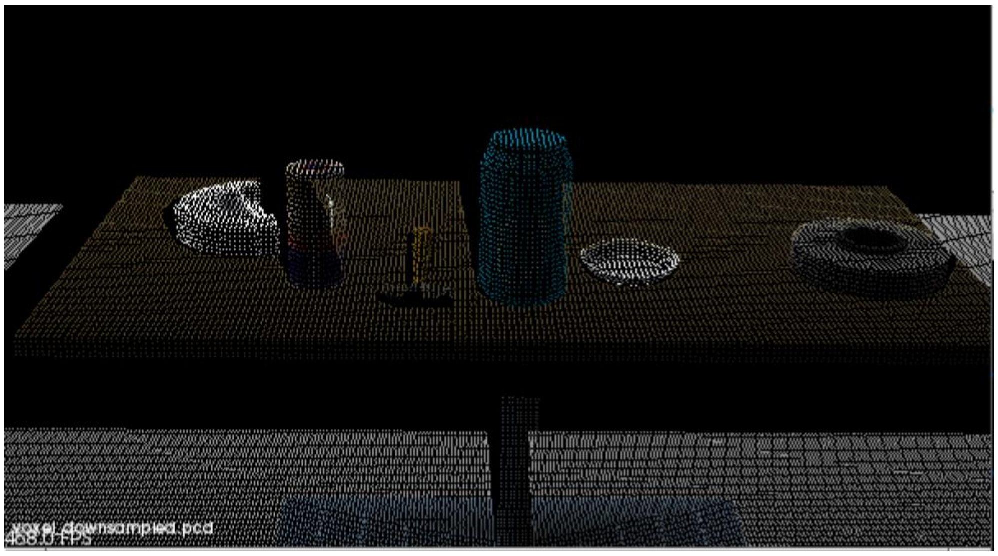
3D Perception Project

By Patrick Wellins

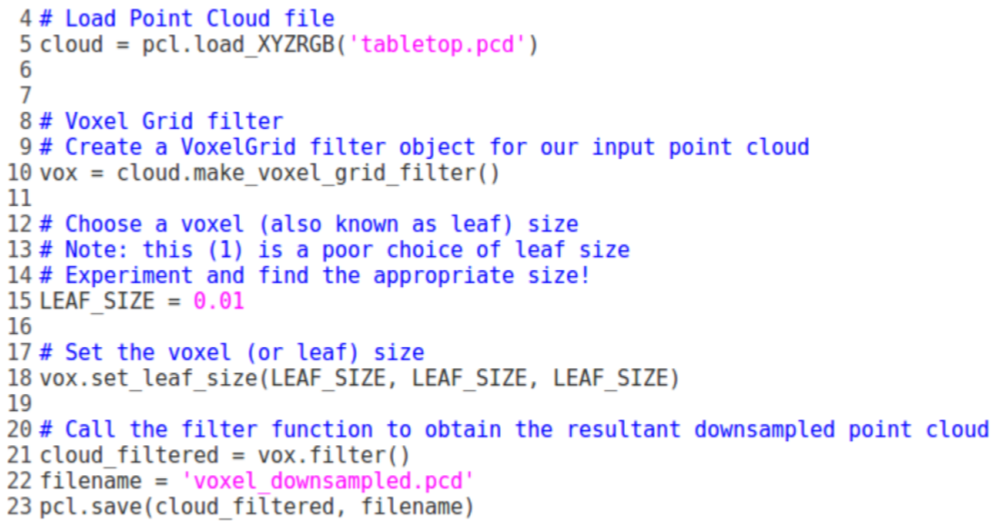
Exercise 1

Voxel Grid Down Sampling

voxel_downsampled.pcd



The image voxel_downsampled.pcd shows the down sampled point cloud data of the table top scene. Down sampling RGBD point cloud data can allow for faster computation without drastically reducing information. The code below shows how the above scene is down sampled. The value for ‘LEAF_SIZE’ determines the size of the voxel. The leaf size is 0.01 which is one centimeter.

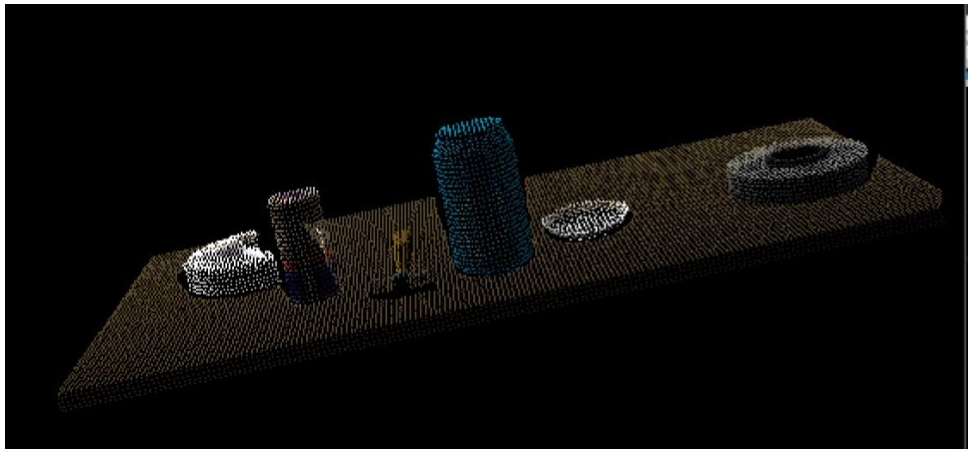


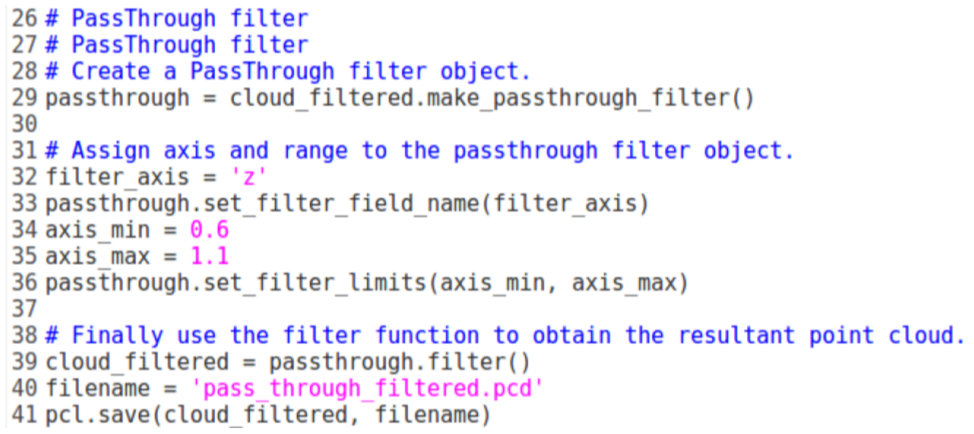
```
# Load Point Cloud file
cloud = pcl.load_XYZRGB('tabletop.pcd')


# Voxel Grid filter
# Create a VoxelGrid filter object for our input point cloud
vox = cloud.make_voxel_grid_filter()

# Choose a voxel (also known as leaf) size
# Note: this (1) is a poor choice of leaf size
# Experiment and find the appropriate size!
LEAF_SIZE = 0.01

# Set the voxel (or leaf) size
vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)

# Call the filter function to obtain the resultant downsampled point cloud
cloud_filtered = vox.filter()
filename = 'voxel_downsampled.pcd'
pcl.save(cloud_filtered, filename)
```

Pass Through Filtering

Pass through filtering isolates a region of interest of the down sampled point cloud. The point cloud is cropped on the z axis to isolate the table and objects. The code below shows the z axis set to the range [0.6, 1.1], this isolates the table top. The x and y axes can also be constrained using pass through filtering.

pass_through_filtered.pcd



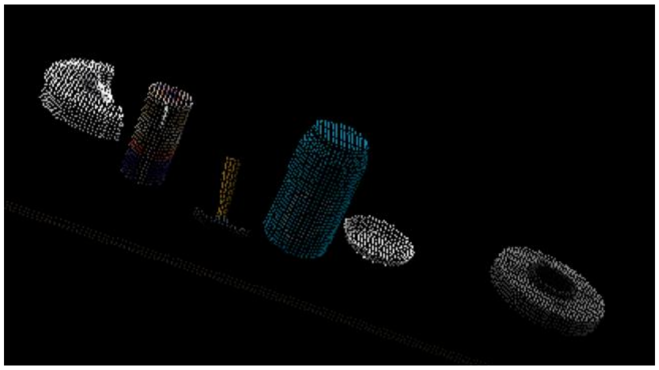


```
# PassThrough filter
# PassThrough filter
# Create a PassThrough filter object.
passthrough = cloud_filtered.make_passthrough_filter()

# Assign axis and range to the passthrough filter object.
filter_axis = 'z'
passthrough.set_filter_field_name(filter_axis)
axis_min = 0.6
axis_max = 1.1
passthrough.set_filter_limits(axis_min, axis_max)

# Finally use the filter function to obtain the resultant point cloud.
cloud_filtered = passthrough.filter()
filename = 'pass_through_filtered.pcd'
pcl.save(cloud_filtered, filename)
```

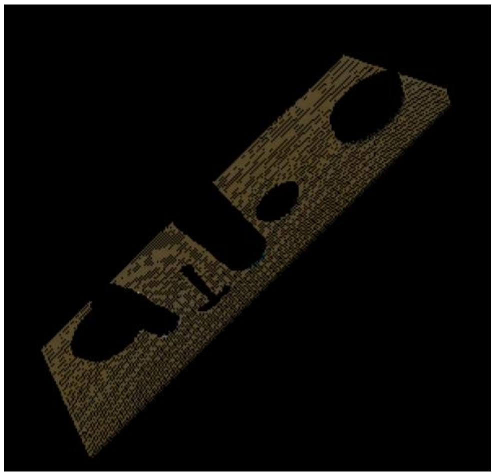
RANSAC Plane Fitting

RANSAC Plane Fitting is used to isolate shapes in the point cloud. In this case, the RANSAC algorithm is used to isolate the table in the point cloud and store the table as ‘extracted_inliers’. The points in the point cloud that are not inliers correspond to the objects on the table. This is helpful because it makes observing only the objects on the table possible. This is done by setting the points in the point cloud that are not in ‘extracted_inliers’ to ‘extracted_outliers’.

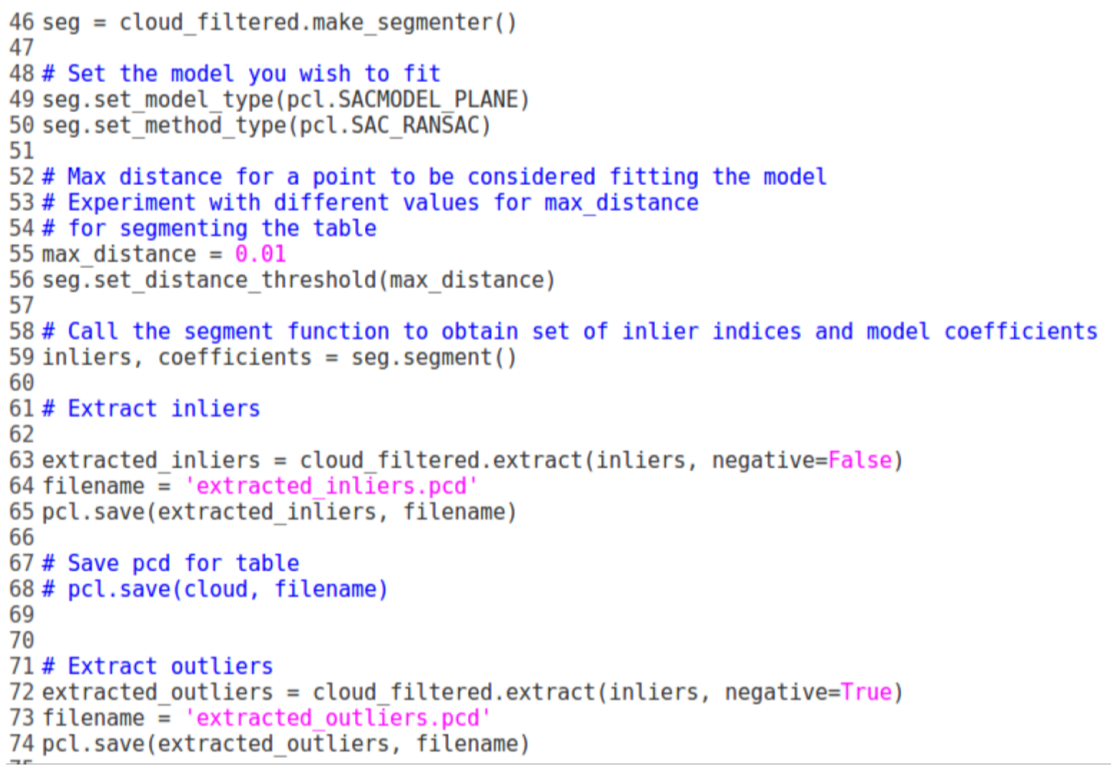
extracted_outliers.pcd



extracted_inliers.pcd



The code below shows the RANSAC filter model and the partitioned point cloud data.



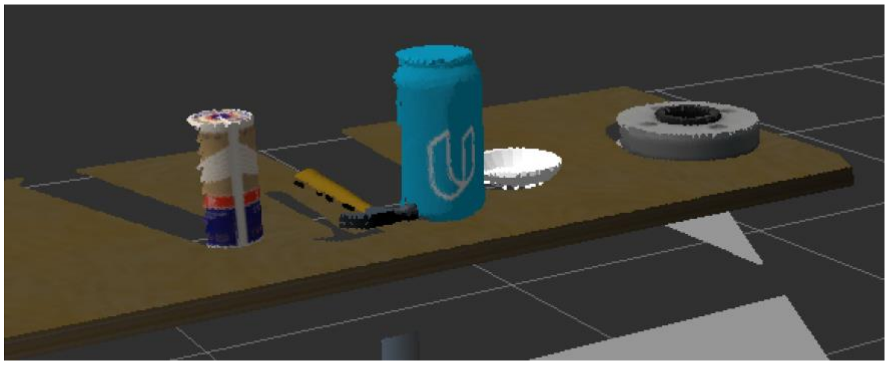
```python
seg = cloud_filtered.make_segmenter()

# Set the model you wish to fit
seg.set_model_type(pcl.SACMODEL_PLANE)
seg.set_method_type(pcl.SAC_RANSAC)

# Max distance for a point to be considered fitting the model
# Experiment with different values for max_distance
# for segmenting the table
max_distance = 0.01
seg.set_distance_threshold(max_distance)

# Call the segment function to obtain set of inlier indices and model coefficients
inliers, coefficients = seg.segment()

# Extract inliers

extracted_inliers = cloud_filtered.extract(inliers, negative=False)
filename = 'extracted_inliers.pcd'
pcl.save(extracted_inliers, filename)

# Save pcd for table
# pcl.save(cloud, filename)


# Extract outliers
extracted_outliers = cloud_filtered.extract(inliers, negative=True)
filename = 'extracted_outliers.pcd'
pcl.save(extracted_outliers, filename)
```

Exercise 2

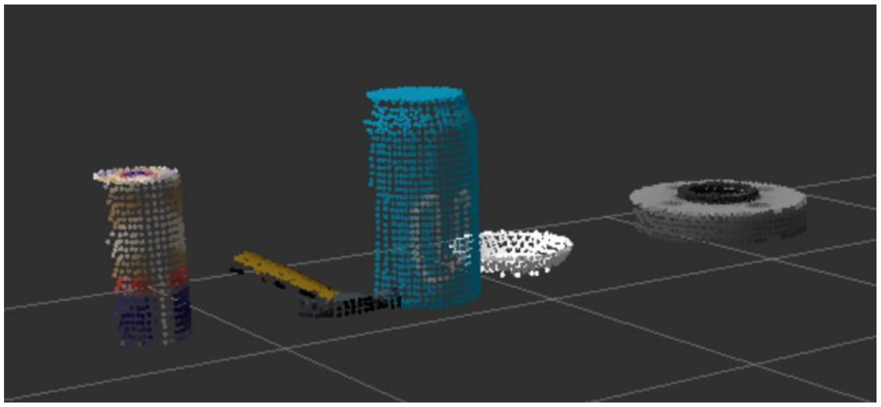
In exercise 2, I filled in the template.py script with code to cluster the objects on the table. To accomplish the task of publishing the ‘pcl_cluster’, the ‘cloud_objects’ are converted into XYZ coordinate data, then the Euclidean clustering algorithm identifies objects as clusters. A specific color is then assigned to the points that belong to a given cluster. To effectively create clusters of the objects on the table, certain parameters are set. Cluster tolerance for the distance threshold is set to 0.03. The distance threshold determines the radius around the point, this radius affects which surrounding points are determined to be neighbors. The minimum cluster size is set to 100 and the maximum cluster size is set to 20,000. Adjusting these parameters will determine what objects show up as clusters in the point cloud.

The images /sensor_stick/point_cloud, /pcl_objects, /pcl_cluster show the progression of going from the original point cloud data to clustered objects. The distinct colors represent the distinct objects on the table.

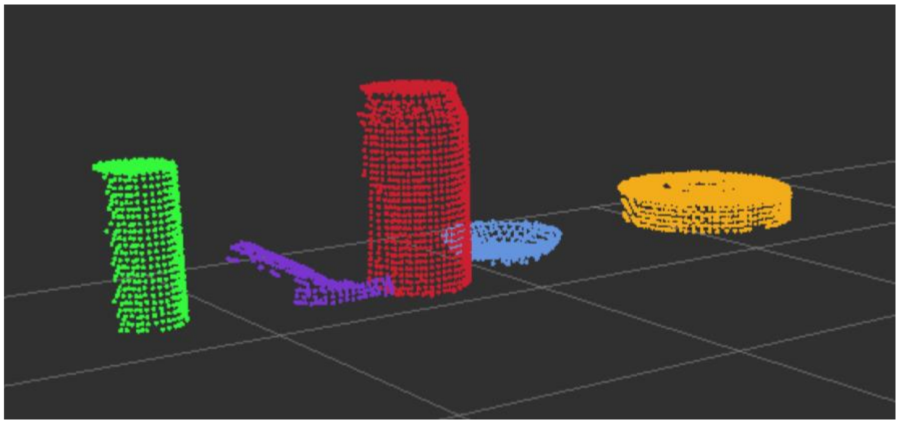
/sensor_stick/point_cloud

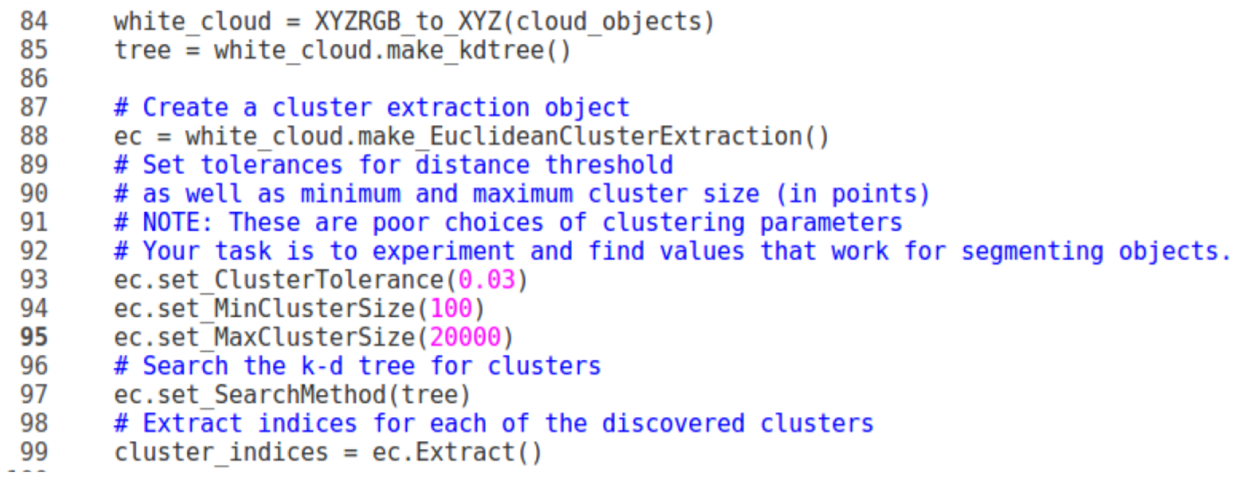


/pcl_objects

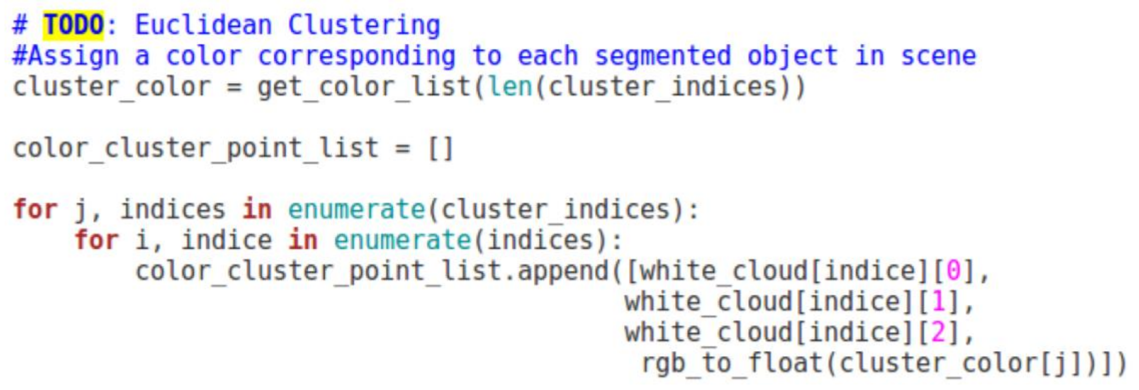


/pcl_cluster





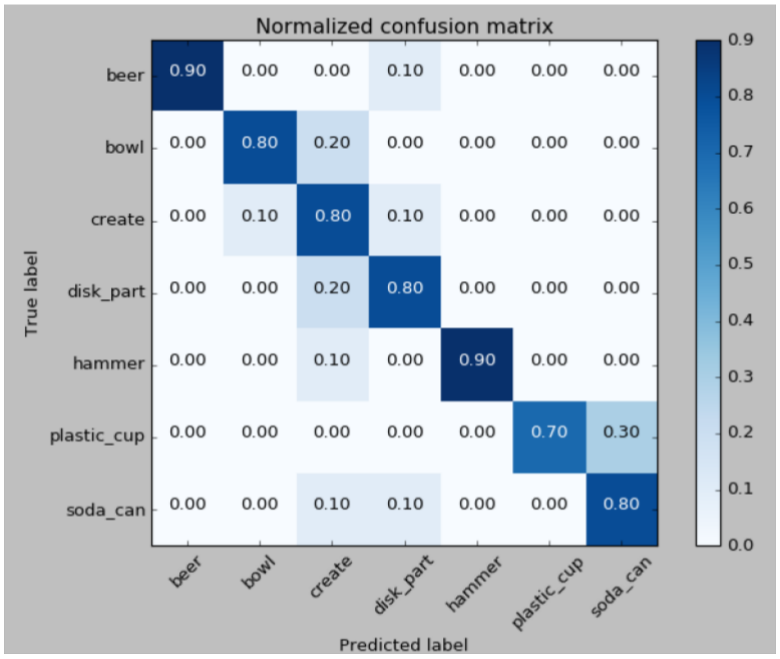
```
    white_cloud = XYZRGB_to_XYZ(cloud_objects)
    tree = white_cloud.make_kdtree()

    # Create a cluster extraction object
    ec = white_cloud.make_EuclideanClusterExtraction()
    # Set tolerances for distance threshold
    # as well as minimum and maximum cluster size (in points)
    # NOTE: These are poor choices of clustering parameters
    # Your task is to experiment and find values that work for segmenting objects.
    ec.set_ClusterTolerance(0.03)
    ec.set_MinClusterSize(100)
    ec.set_MaxClusterSize(20000)
    # Search the k-d tree for clusters
    ec.set_SearchMethod(tree)
    # Extract indices for each of the discovered clusters
    cluster_indices = ec.Extract()
```



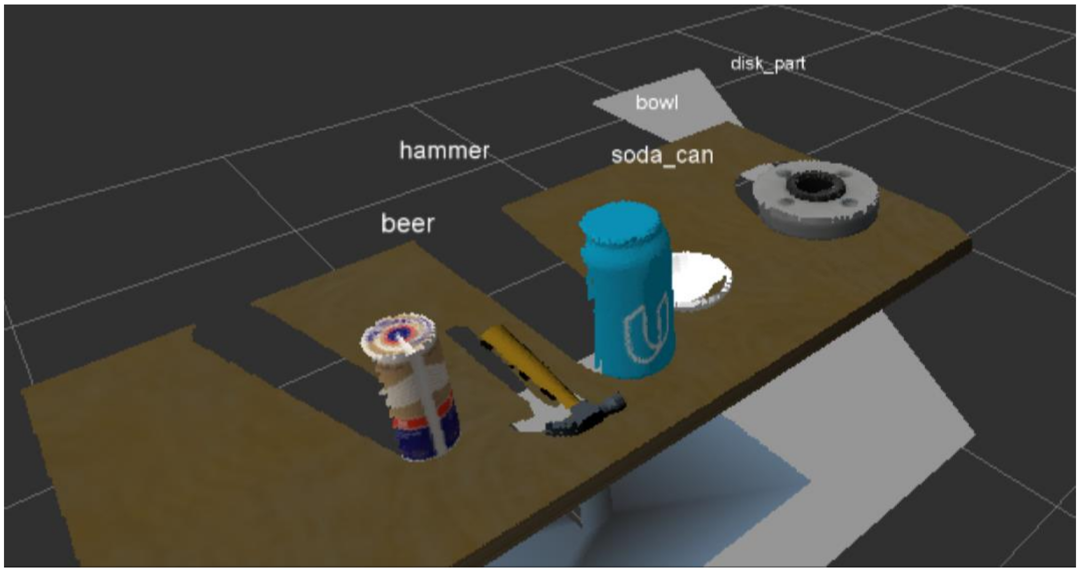
```
# TODO: Euclidean Clustering
#Assign a color corresponding to each segmented object in scene
cluster_color = get_color_list(len(cluster_indices))

color_cluster_point_list = []

for j, indices in enumerate(cluster_indices):
    for i, indice in enumerate(indices):
        color_cluster_point_list.append([white_cloud[indice][0],
                                        white_cloud[indice][1],
                                        white_cloud[indice][2],
                                         rgb_to_float(cluster_color[j])])
```

Exercise 3

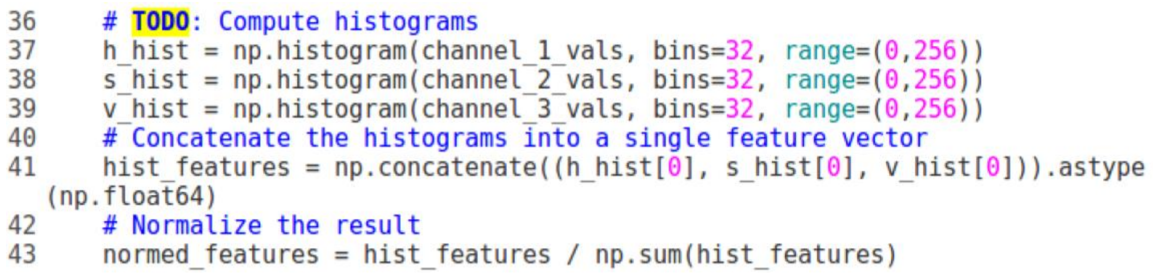
Machine learning is utilized in exercise 3 to identify the objects on the table. Feature data corresponding to the objects on the table and a SVM classifier are used to achieve this task. Features of the table objects are generated by randomly spawning the objects in different orientations in RViz. For exercise 3, I adjusted the settings in capture_features.py to spawn the object 9 times. The features consist of color information and surface normal information. The distribution of surface normal information indicates the shape of the object. These features are then used to train a SVM classifier with a linear kernel to identify objects. The SVM separates different classes of points with a hyper plane. A special feature of the SVM is that the hyper plane can be placed in position to maximize the margin between different classes of points. The confusion matrix below shows the performance of the SVM on the training set. The accuracy ranges from 0.70 – 0.90 for identifying the various objects. The SVM was powerful enough to accurately label all the objects on the table for exercise 3.



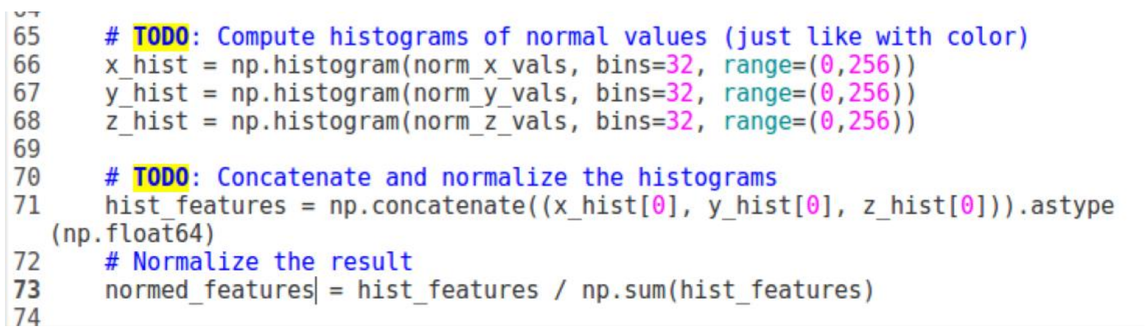
The image below shows the table with objects. The objects are identified with the trained SVM.



The two blocks of code below from features.py show how the color and normal vector data is placed into feature data. For both the color data and the normal vector data, three histograms are concatenated and then normalized to create feature vectors.

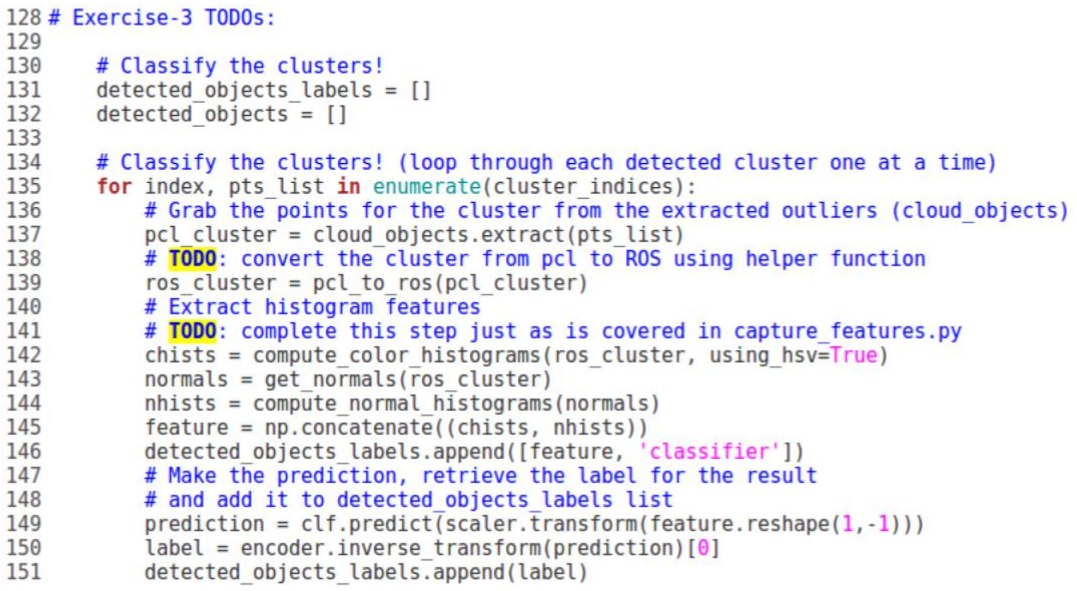


```
    # TODO: Compute histograms
    h_hist = np.histogram(channel_1_vals, bins=32, range=(0,256))
    s_hist = np.histogram(channel_2_vals, bins=32, range=(0,256))
    v_hist = np.histogram(channel_3_vals, bins=32, range=(0,256))
    # Concatenate the histograms into a single feature vector
    hist_features = np.concatenate((h_hist[0], s_hist[0], v_hist[0])).astype(np.float64)
    # Normalize the result
    normed_features = hist_features / np.sum(hist_features)
```

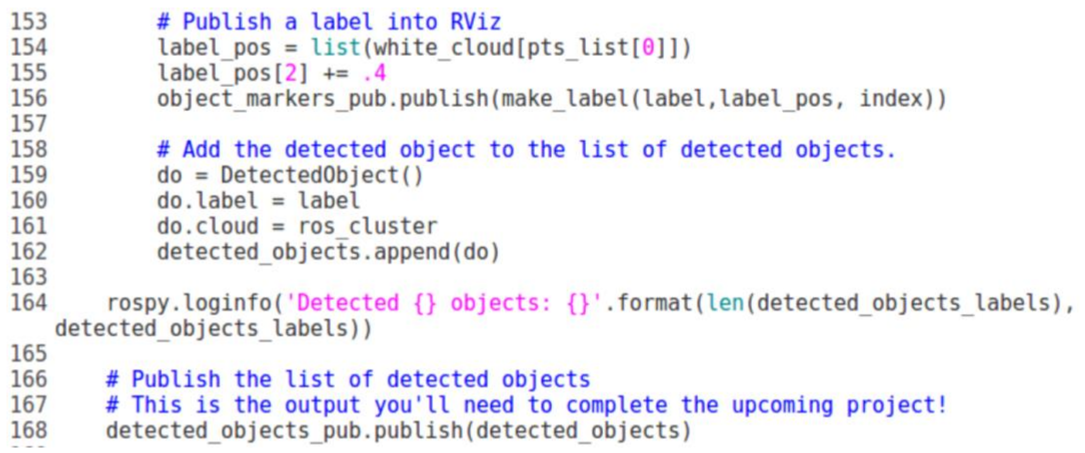


```
    # TODO: Compute histograms of normal values (just like with color)
    x_hist = np.histogram(norm_x_vals, bins=32, range=(0,256))
    y_hist = np.histogram(norm_y_vals, bins=32, range=(0,256))
    z_hist = np.histogram(norm_z_vals, bins=32, range=(0,256))

    # TODO: Concatenate and normalize the histograms
    hist_features = np.concatenate((x_hist[0], y_hist[0], z_hist[0])).astype(np.float64)
    # Normalize the result
    normed_features = hist_features / np.sum(hist_features)

```

The code below shows the process of converting point cloud clusters into labeled data. The trained SVM is responsible for matching the pattern of the data to the correct label. The labeled objects are then published.



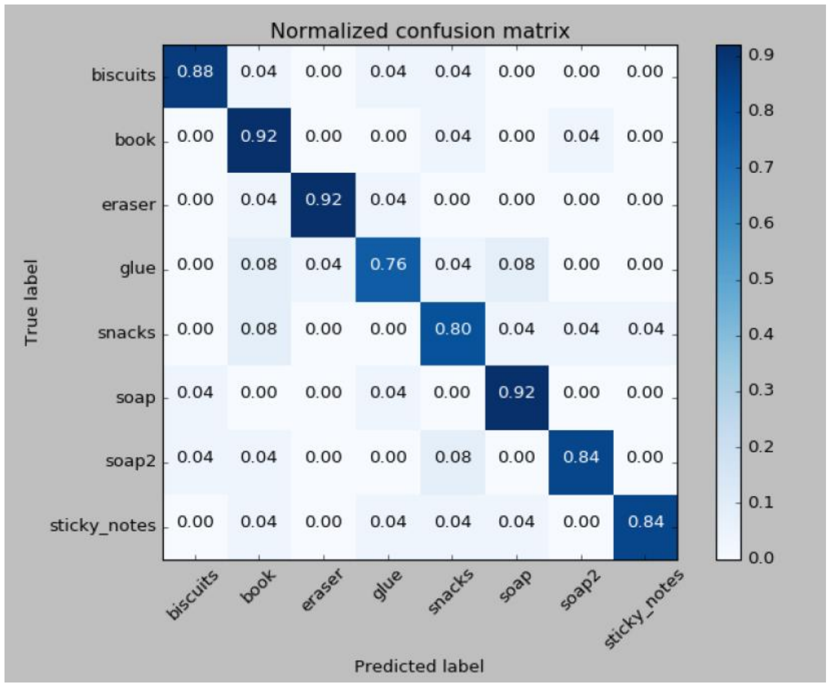
```
# Exercise-3 TODOs:

    # Classify the clusters!
    detected_objects_labels = []
    detected_objects = []

    # Classify the clusters! (loop through each detected cluster one at a time)
    for index, pts_list in enumerate(cluster_indices):
        # Grab the points for the cluster from the extracted outliers (cloud_objects)
        pcl_cluster = cloud_objects.extract(pts_list)
        # TODO: convert the cluster from pcl to ROS using helper function
        ros_cluster = pcl_to_ros(pcl_cluster)
        # Extract histogram features
        # TODO: complete this step just as is covered in capture_features.py
        chists = compute_color_histograms(ros_cluster, using_hsv=True)
        normals = get_normals(ros_cluster)
        nhists = compute_normal_histograms(normals)
        feature = np.concatenate((chists, nhists))
        detected_objects_labels.append([feature, 'classifier'])
        # Make the prediction, retrieve the label for the result
        # and add it to detected_objects_labels list
        prediction = clf.predict(scaler.transform(feature.reshape(1,-1)))
        label = encoder.inverse_transform(prediction)[0]
        detected_objects_labels.append(label)
```



```
        # Publish a label into RViz
        label_pos = list(white_cloud[pts_list[0]])
        label_pos[2] += .4
        object_markers_pub.publish(make_label(label,label_pos, index))

        # Add the detected object to the list of detected objects.
        do = DetectedObject()
        do.label = label
        do.cloud = ros_cluster
        detected_objects.append(do)

    rospy.loginfo('Detected {} objects: {}'.format(len(detected_objects_labels),
detected_objects_labels))

    # Publish the list of detected objects
    # This is the output you'll need to complete the upcoming project!
    detected_objects_pub.publish(detected_objects)
```

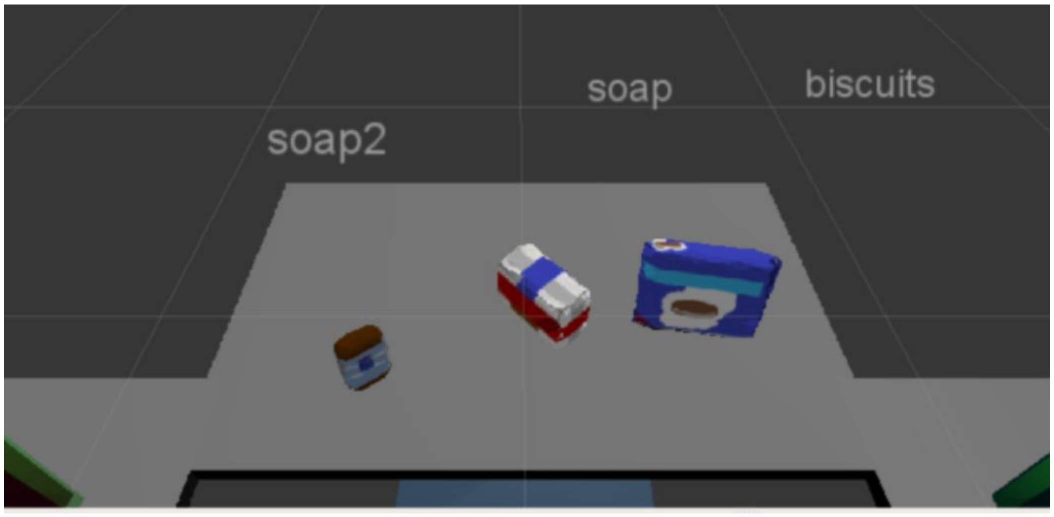
3D Perception

For the 3D Perception project, I used the previously developed code from exercises 1 – 3 to identify objects on three test scenes. I generated new feature data for the new objects that appear in the three different test scenes and trained a SVM. To meet the accuracy requirements for the three scenes, I recorded more feature data by spawning the objects 25 times. I kept the same parameters for the SVM, that were used in exercise 3, and the same number of bins for the histograms that construct the feature data. It is also worth noting that the color data is in HSV form, this prevents different levels of light in the environment from distorting the color information of the objects. Below is an image of the confusion matrix that shows the performance of the trained SVM. The accuracy scores for the objects range from 0.76 – 0.92, this level of performance was good enough to meet the accuracy criteria for the project.

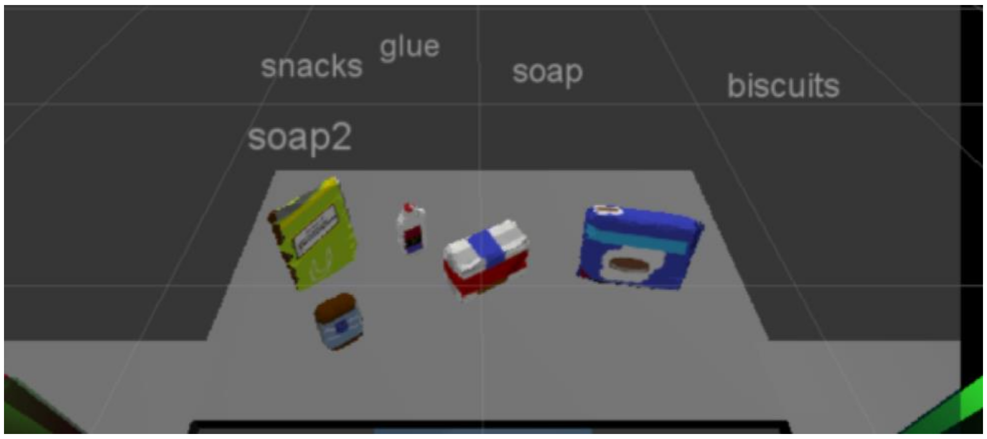


True label \ Predicted label	biscuits	book	eraser	glue	snacks	soap	soap2	sticky_notes
biscuits	0.88	0.04	0.00	0.04	0.04	0.00	0.00	0.00
book	0.00	0.92	0.00	0.00	0.04	0.00	0.04	0.00
eraser	0.00	0.04	0.92	0.04	0.00	0.00	0.00	0.00
glue	0.00	0.08	0.04	0.76	0.04	0.08	0.00	0.00
snacks	0.00	0.08	0.00	0.00	0.80	0.04	0.04	0.04
soap	0.04	0.00	0.00	0.04	0.00	0.92	0.00	0.00
soap2	0.04	0.04	0.00	0.00	0.08	0.00	0.84	0.00
sticky_notes	0.00	0.04	0.00	0.04	0.04	0.04	0.00	0.84

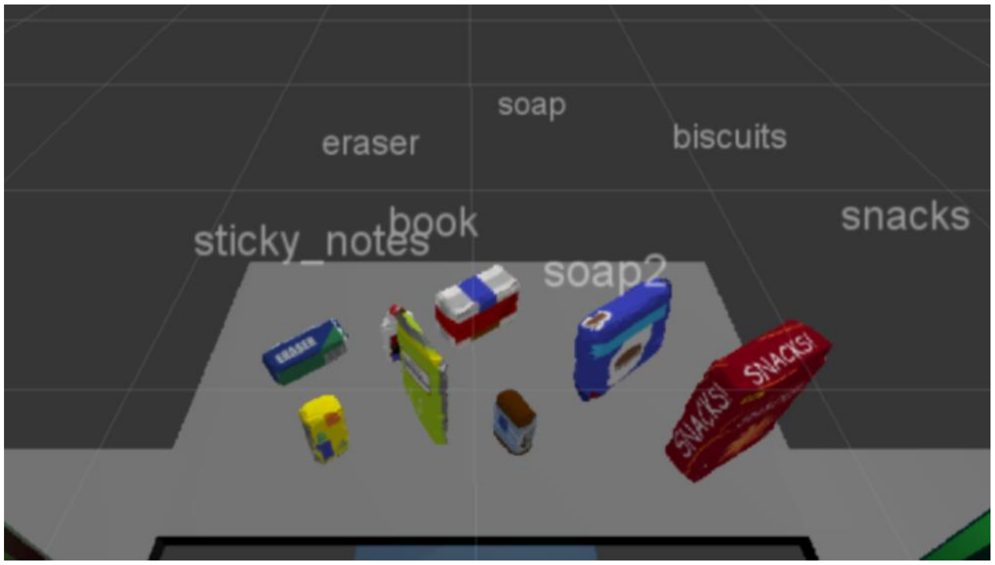
The image below shows the labeled items in the first test1.world. The SVM classifies each item accurately. I had to modify the code that was in the exercises to prevent the red bin in the scene from being identified as snacks. To solve this problem, I created a pass-through filter to adjust the x – axis dimensions. Restricting the x – dimensions of the point cloud prevented the red bin from being identified as snacks.



The SVM classified 4 out of 5 items correctly in test2.world. The book is misidentified as snacks in this scene. Here are a few potential ways I can build a stronger object recognition system. The first is generating more feature data, spawning the objects 100 times could lead to stronger model performance. The second approach is adjusting the parameters of the features, changing the binning of the histograms that build the feature vectors could improve model performance. The hyper-parameters of the SVM model could be optimized with cross validation and grid search. Using an 'rbf' kernel can be helpful when working with non-linearly separable data and would possibly lead to stronger results for this task.



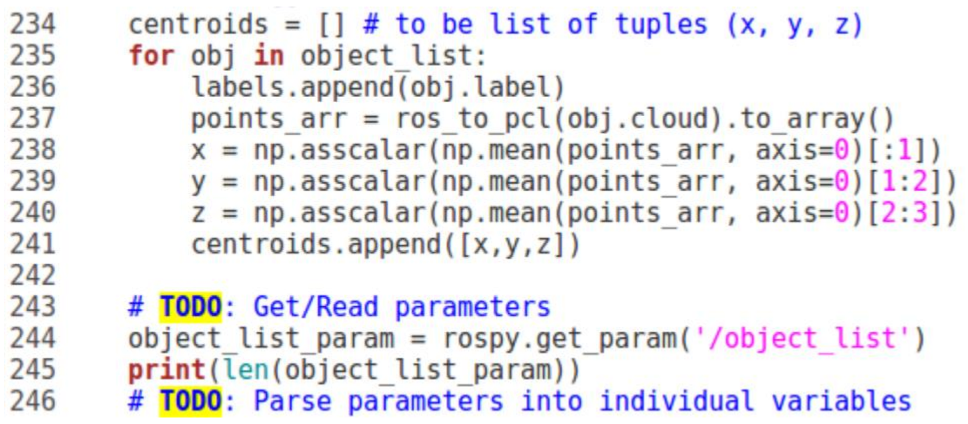
In test world 3, the classifier identified 7 out of 8 items in the scene. The only item that was not identified is glue, I suspect glue was not identified due to it being obstructed by the book. A way to prevent this problem is to have multiple cameras positioned at different angles that run the same image recognition algorithms. The downside of this approach is increased cost and complexity, however there can be gains in accuracy. The second approach to solve this problem is for the robot to pick and place the items that it can see which will allow the unseen items to be identified.



Output yaml files

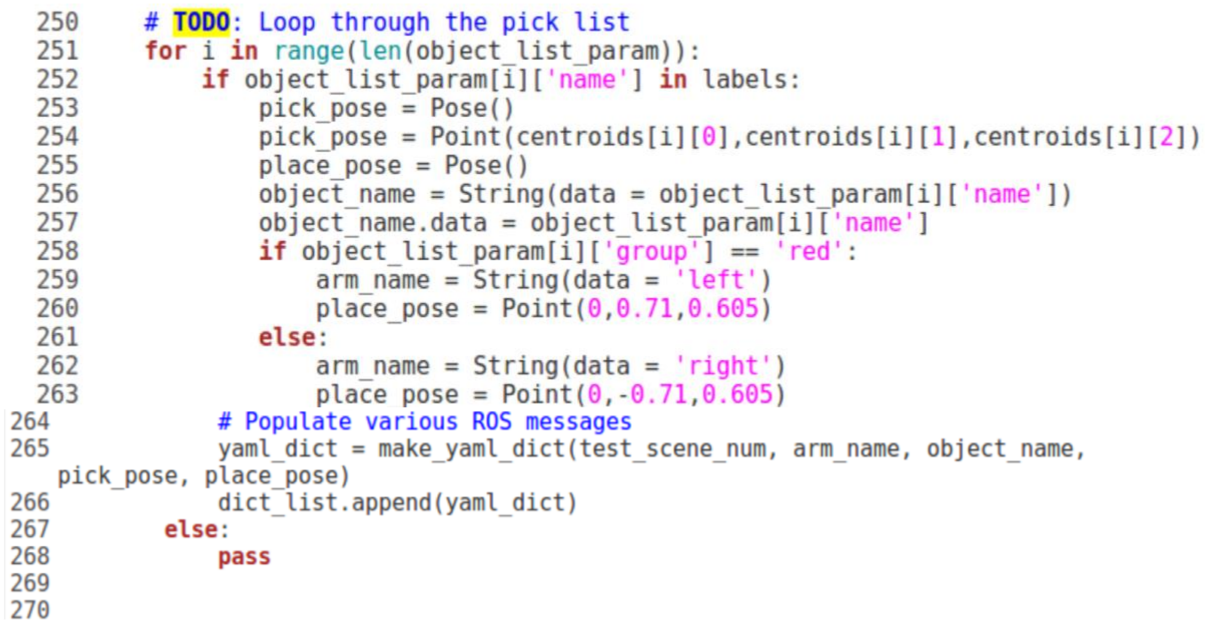
I built the pr2_mover function to output various ROS messages to yaml files. The input of the pr2_mover function is the detected object list that is generated from the pcl_callback function. This list has point cloud information along with corresponding labels. The ROS messages that are created include the which arm the PR2 uses for the pick and place, the object name, the pick pose, the place pose, and the test scene number.

The block of code below is responsible for collecting the centroid values of the objects, these centroid values are used to construct the ‘pick_pose’ ROS message.



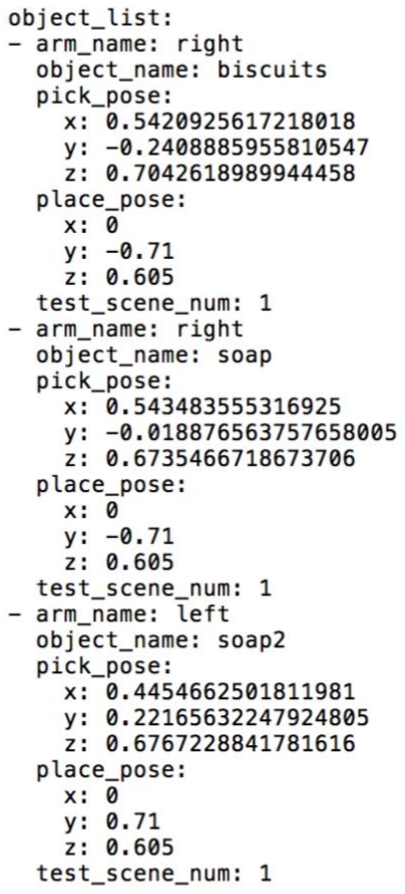
```python
centroids = [] # to be list of tuples (x, y, z)
for obj in object_list:
    labels.append(obj.label)
    points_arr = ros_to_pcl(obj.cloud).to_array()
    x = np.asscalar(np.mean(points_arr, axis=0)[:1])
    y = np.asscalar(np.mean(points_arr, axis=0)[1:2])
    z = np.asscalar(np.mean(points_arr, axis=0)[2:3])
    centroids.append([x,y,z])

# TODO: Get/Read parameters
object_list_param = rospy.get_param('/object_list')
print(len(object_list_param))
# TODO: Parse parameters into individual variables
```

The block of code below is a loop that builds a list of yaml dictionaries that contain ROS messages. There is some logic in the loop that decides which arm is used and the place pose. This is determined by the color of the bin that the object is to be placed in, for example on line 258 there is condition that checks if the bin is red, if the bin is red then the left arm is used. The place pose is also determined by the color of the bin that the object is to be placed in.



```python
250     # TODO: Loop through the pick list
251     for i in range(len(object_list_param)):
252         if object_list_param[i]['name'] in labels:
253             pick_pose = Pose()
254             pick_pose = Point(centroids[i][0],centroids[i][1],centroids[i][2])
255             place_pose = Pose()
256             object_name = String(data = object_list_param[i]['name'])
257             object_name.data = object_list_param[i]['name']
258             if object_list_param[i]['group'] == 'red':
259                 arm_name = String(data = 'left')
260                 place_pose = Point(0,0.71,0.605)
261             else:
262                 arm_name = String(data = 'right')
263                 place_pose = Point(0,-0.71,0.605)
264             # Populate various ROS messages
265             yaml_dict = make_yaml_dict(test_scene_num, arm_name, object_name,
    pick_pose, place_pose)
266             dict_list.append(yaml_dict)
267         else:
268             pass
269
270
```

Here is the output_1.yaml file.



```
object_list:
- arm_name: right
  object_name: biscuits
  pick_pose:
    x: 0.5420925617218018
    y: -0.2408885955810547
    z: 0.7042618989944458
  place_pose:
    x: 0
    y: -0.71
    z: 0.605
  test_scene_num: 1
- arm_name: right
  object_name: soap
  pick_pose:
    x: 0.543483555316925
    y: -0.018876563757658005
    z: 0.6735466718673706
  place_pose:
    x: 0
    y: -0.71
    z: 0.605
  test_scene_num: 1
- arm_name: left
  object_name: soap2
  pick_pose:
    x: 0.4454662501811981
    y: 0.22165632247924805
    z: 0.6767228841781616
  place_pose:
    x: 0
    y: 0.71
    z: 0.605
  test_scene_num: 1
```