Arnold Kompaniyets

Robo ND (Term 1)
May 30, 2018

3D Perception: Pick and Place
Write-Up

Code Analysis:

In order to describe all the steps which were taken in successfully identifying various objects with an RGB-D camera, it would be best to go step by step through the Python code run in the final Pick and Place Project environment. The code will be detailed in parts from start to finish:

Helpers:

```
# Import modules
import numpy as np
import sklearn
from sklearn.preprocessing import LabelEncoder
import pickle
from sensor_stick.srv import GetNormals
from sensor_stick.features import compute_color_histograms
from sensor_stick.features import compute_normal_histograms
from visualization_msgs.msg import Marker
from sensor_stick.marker_tools import *
from sensor_stick.msg import DetectedObjectsArray
from sensor_stick.msg import DetectedObject
from sensor_stick.pcl_helper import *

import rospy
import tf
from geometry_msgs.msg import Pose
from std_msgs.msg import Float64
from std_msgs.msg import Int32
from std_msgs.msg import String
from pr2_robot.srv import *
from rospy_message_converter import message_converter
import yaml
```

This first chunk of code serves to import all of the necessary modules that will be required to make the object recognition possible. Numpy is imported as usual for various math functions. Sklearn is used here for the machine learning step to have the robot make

predictions based on trained data. There are also quite a few modules imported from the "sensor_stick" code made in exercises 2 and 3. This includes code to compute concatenated histograms (one for color and one for normal vectors, i.e. the 3D shape of the object), used later in the code to help the robot properly identify matching objects based on color and shape analysis. Helping code is also used to help place markers on the screen for easier visualization of the code's success.

Continuing, a few extra modules needed to be imported in order to successfully transform the output data in a list of identifiers and positions for the pick and place server to use via a '.yaml' file. This includes a few data types in standard messages (Float64, Int32, and String), as well as a data type in geometry messages.

```
# Helper function to get surface normals
def get_normals(cloud):
    get_normals_prox = rospy.ServiceProxy('/feature_extractor/get_normals',
        GetNormals)
    return get_normals_prox(cloud).cluster
```

There were also a few helper functions included in this code. The first of these implements the GetNormals function on a given point cloud in order to return a cluster of normal values corresponding to the input pointcloud, or in other words, purely the depth information from the RGB-D point cloud.

```
# Helper function to create a yaml friendly dictionary from ROS messages
def make_yaml_dict(test_scene_num, arm_name, object_name, pick_pose,
place_pose):
    yaml_dict = {}
    yaml_dict["test_scene_num"] = test_scene_num.data
    yaml_dict["arm_name"]  = arm_name.data
    yaml_dict["object_name"] = object_name.data
    yaml_dict["pick_pose"] =
        message_converter.convert_ros_message_to_dictionary(pick_pose)
    yaml_dict["place_pose"] =
        message_converter.convert_ros_message_to_dictionary(place_pose)
    return yaml_dict
```

Another helper function is able to take as input all of the ROS messages that will be made later on in this code, 5 total, and convert them into a Python dictionary (to be used as the .yaml file later on).

```
# Helper function to output to yaml file
def send_to_yaml(yaml_filename, dict_list):
    data_dict = {"object_list": dict_list}
    with open(yaml_filename, 'w') as outfile:
        yaml.dump(data_dict, outfile, default_flow_style=False)
```

The last helper function to discuss, this bit of code takes a series of dictionaries, created using the function described above, and places all of them into a specified .yaml file.

Next, in order to better understand the workings of the 'pcl_callback' function, I think it is prudent to first look at the ros node initialization code block, located at the very end of the Python file:

Node initialization:

```
if __name__ == '__main__':

    # TODO: ROS node initialization
    rospy.init_node('project', anonymous=True)
```

Firstly, the node is initialized. I chose to title it "project".

```
    # TODO: Create Subscribers
    pcl_sub = rospy.Subscriber("/pr2/world/points", pc2.PointCloud2, pcl_callback,
        queue_size=1)
```

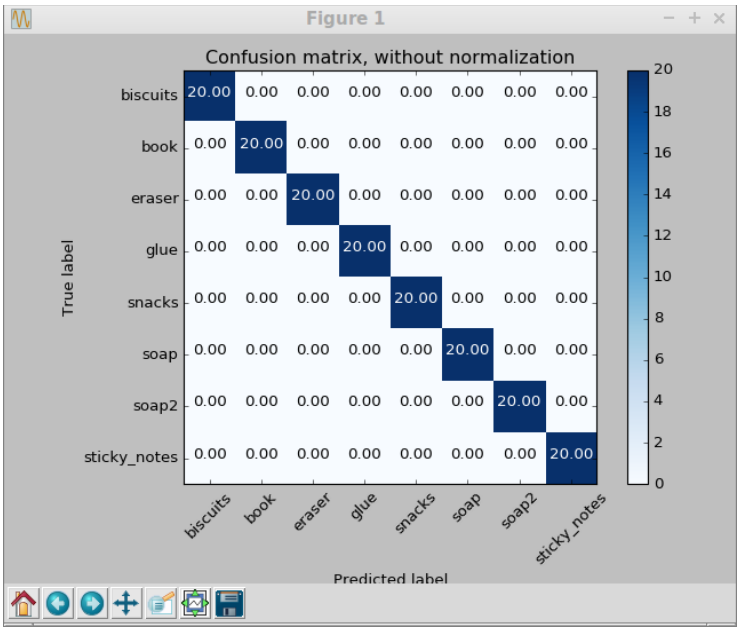
Next, a subscriber is needed. The publisher "/pr2/world/points" outputs pointcloud data of everything in front of the robot's camera, including the table, all of the objects, and even extraneous noise that would be present in a real-world scenario. As can be seen in this code, the "pcl_callback" function is called whenever the subscriber receives data, which will be described in full detail below.

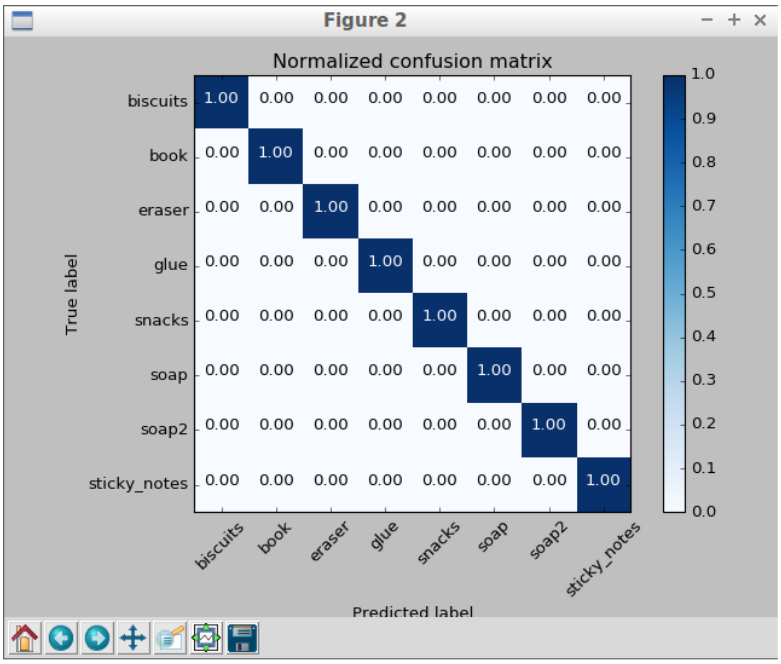
```
    # TODO: Create Publishers
    pcl_objects_pub = rospy.Publisher("/pcl_objects", PointCloud2, queue_size=1)
    pcl_table_pub = rospy.Publisher("/pcl_table", PointCloud2, queue_size=1)
    pcl_clusters_pub = rospy.Publisher("/pcl_clusters", PointCloud2, queue_size=1)
    object_markers_pub = rospy.Publisher("/object_markers", Marker, queue_size=1)
    detected_objects_pub = rospy.Publisher("/detected_objects", DetectedObjectsArray,
        queue_size=1)
```

Continuing, publishers need to be created. While not all of these publishers are absolutely necessary for the code to run properly, they do a fantastic job in visualizing the various parts of the function and were quite helpful in testing and troubleshooting. The "/pcl_objects" publisher outputs only the pointcloud data of the objects on top of the table, whereas the "/pcl_table" publisher outputs the table only. The "/pcl_clusters" publisher shows the same pointcloud data as the first publisher, but this time divides each object into each own pointcloud cluster. The last two publishers are directly related to the object recognition step, with "/object_markers" publishing labels over the detected objects and "/detected_objects" printing out an array of all the detected objects.

```
# TODO: Load Model From disk
model = pickle.load(open('model.sav', 'rb'))
clf = model['classifier']
encoder = LabelEncoder()
encoder.classes_ = model['classes']
scaler = model['scaler']
```

This step is very important in providing the program with correct object parameters when matching provided camera data. The first step is in loading the 'model.sav' file located within the current directory. This particular file actually took a bit of time to generate, because it was a combination of the 'capture_features.py' and 'train_svm.py' files used in Exercise 3. The 'capture_features.py' function took multiple views of each object that could potentially appear on the table and generated a list of features and parameters. In my function, I had the program take 20 views for each object to generate valid pointcloud data (instead of the starting 5). This data was then used to derive concatenated color histograms (using HSV instead of RGB), concatenated normal vector histograms, all eventually concatenated into one single array for SVM to use. Altogether, this data was called 'training_set.sav' and used in the 'train_svm.py' code. Within this python file, the array of previously attained features is scaled using StandardScaler().fit() and then trained using an SVM 'linear' kernel. Once the classifier is trained, it is saved under the 'model.sav' file, which is exactly the file loaded within the code above. One of the benefits of using the 'train_svm.py' program to train the feature data is that two plots are shown afterwards to illustrate the accuracy of the training features (one plot showing the raw data and the second normalizing the data to 1 for each object). With the settings used above, the classifier was able to correctly predict each of the eight objects within this project 100 % of the time:






```
# Initialize color_list
get_color_list.color_list = []
```

This simply initializes a list of random colors to be used in object clustering described below.

```
# TODO: Spin while node is not shutdown
while not rospy.is_shutdown():
    rospy.spin()
```

As with most nodes programmed thus far, it is instructed to spin, or simply keep running, until it is explicitly shut down (versus stopping as soon as a loop is run).

Now with an understanding of how the node is initialized, we can discuss exactly what happens in the 'pcl_callback' function when pointcloud data is received by the subscriber:

Callback function:

```
# Callback function for your Point Cloud Subscriber
def pcl_callback(pcl_msg):

# Exercise-2 TODOs:

    # TODO: Convert ROS msg to PCL data
    cloud = ros_to_pcl(pcl_msg)
```

When working with the cloudpoint data, the first step is to transform the data cloud from ROS to PCL. This will allow the pointcloud to be manipulated with the various filters below.

```
    # TODO: Statistical Outlier Filtering
    outlier_filter = cloud.make_statistical_outlier_filter()
    outlier_filter.set_mean_k(25)
    x = 0.05
    outlier_filter.set_std_dev_mul_thresh(x)
    cloud_filtered = outlier_filter.filter()
```

Just as in attaining real-life data using an RGB-D camera, the pointcloud data attained in this project also contains various noise which must be minimized. First, the filter itself is created using the "make_statistical_outlier_filter()" function. Following this, I chose to set the "mean_k" to 25, or in other words, 25 neighboring points are analyzed when searching for outlier point data. Afterwards, after much experimentation, I set the threshold scale factor to 0.05 (meaning that, presumably, any point greater than 5cm away than the nearest bunch of 25 points counts as an outlier). After the filter is run, not all outlier points are eliminated, but while running the rest of the code, it appeared fully sufficient.

```
    # TODO: Voxel Grid Downsampling
    # Create a VoxelGrid filter object for our input point cloud
```

```
vox = cloud_filtered.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 = 0.004

# 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)
```

The second step was to invoke the voxel grid filter, which divides the pointcloud data into easier to manipulate 3D boxes. As with all other filters, this filter is first created using the PCL-provided function. Also with much experimentation, a leaf size of 0.004 was selected in order to, presumably, set each 3D box to have a side length of 4mm. This makes for quite a lot of voxels, which does lead to more necessary computation, but at the same time also allows for greater detail in both generating color histograms and depth parameters for various objects.

```
# TODO: 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 = 0.61
axis_max = 1.2
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)
```

The first passthrough filter used in this code acts to crop the vertical, or z, axis to include only the tabletop and the objects in question. After creating this filter, the vertical axis is selected, and the range is designated (61 cm to 120 cm). The manipulated pointcloud data is then saved.

```
passthrough_2 = cloud_filtered.make_passthrough_filter()

filter_axis = 'y'
passthrough_2.set_filter_field_name(filter_axis)
axis_min = -0.55
axis_max = 0.5
passthrough_2.set_filter_limits(axis_min, axis_max)

cloud_filtered = passthrough_2.filter()
filename = 'pass_through_filtered_2.pcd'
pcl.save(cloud_filtered, filename)
```

The second passthrough filter is used to crop the horizontal, or y, axis. This is primarily done to eliminate the drop-off bins from the field of view and prevent the algorithm from identifying either bin as a potential object. As with the previous passthrough filter, the object is created, the axis is selected, and then the range is set (this time -55 cm to 50 cm). This re-edited pointcloud is then saved.

```
# TODO: RANSAC Plane Segmentation
# Create the segmentation object
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()
```

The next step to be implemented is to separate the table from the objects in question. Since the shape and location of the objects could greatly vary, it is much easier to search for the shape of the table instead. After the filter object is created, a pre-loaded model of a simple plane was used as the template shape. A threshold of 1cm error was set to allow a pointcloud to fit the model shape, which appeared plenty sufficient to recognize the table. These points are called “inliers”.

```
# TODO: Extract inliers and outliers
# Extract inliers
extracted_inliers = cloud_filtered.extract(inliers, negative=False)
filename = 'extracted_inliers.pcd'
pcl.save(extracted_inliers, filename)

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

After the inliers are identified, they are extracted into a separate file. This is not particularly useful though. Instead, the points outside the inliers can be extracted, which would correspond to the objects on the table. These are saved as outliers.

```
# TODO: Euclidean Clustering
white_cloud = XYZRGB_to_XYZ(extracted_outliers)
tree = white_cloud.make_kdtree()
```

The last step in filtering was to identify each object as its own individual cluster. The first step to this was to eliminate color data, since only the locations of the points in question matter for the purpose of clustering. This was done by converting the outlier pointcloud from type XYZRGB to simply type XYZ. Then, a k-dimensional tree was made using this pointcloud.

```
# TODO: Create Cluster-Mask Point Cloud to visualize each cluster separately
# 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(0.01)
ec.set_MinClusterSize(200)
ec.set_MaxClusterSize(4500)
# Search the k-d tree for clusters
ec.set_SearchMethod(tree)
# Extract indices for each of the discovered clusters
cluster_indices = ec.Extract()
```

As always, a filter object was first created, only this time initialized using the make_EuclideanClusterExtraction() function. Next, specific parameters for the desired

clusters are set. The minimum cluster size had to be set high enough so that the algorithm wouldn't create plethora mini-clusters but instead only one per object. The maximum cluster size had to be set high enough in order to include bigger objects on the table; otherwise, the larger clusters would simply be ignored. The cluster tolerance was also set, in this case to 0.01 or, presumably, 1 cm. This fairly low tolerance allows the algorithm to classify clusters more than a centimeter apart, which is perfectly fine for this selection of objects (since they are all relatively box shaped versus more complex cluster shapes).

With the parameters set, the k-dimensional tree is used to identify potential clusters, the indices of which from the tree are set as "cluster_indices".

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

color_cluster_point_list = []
```

Two lists are initialized here, one being a list of random colors the length of detected objects, and the other being a blank list that will be populated with the points of each cluster.

```
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])])

#Create new cloud containing all clusters, each with unique color
cluster_cloud = pcl.PointCloud_PointXYZRGB()
cluster_cloud.from_list(color_cluster_point_list)
```

Within each of the index clusters detected, the color_cluster_point_list is appended, adding the XYZ coordinates of the indexed points (using the 'white_cloud' cluster created earlier) and assigning a color to said point (one color point object cluster).

Lastly, a new pointcloud is created of type XYZRGB and populated with the location and color of all objects identified in the 'for' loops above.

```
# TODO: Convert PCL data to ROS messages
ros_cloud_objects = pcl_to_ros(extracted_outliers)
ros_cloud_table = pcl_to_ros(extracted_inliers)
ros_cluster_cloud = pcl_to_ros(cluster_cloud)
```

```
    # TODO: Publish ROS messages
    pcl_objects_pub.publish(ros_cloud_objects)
    pcl_table_pub.publish(ros_cloud_table)
    pcl_clusters_pub.publish(ros_cluster_cloud)
```

In order to properly visualize the success of the algorithm implemented above, three different pointclouds are published. The pointclouds of the table, the objects, and the colored clusters are converted from PCL back to ROS and then published to their respective topics, identified inside the node initialization.

With each object now identified and in its own pointcloud cluster, the objects can now be properly identified:

```
# Exercise-3 TODOs:

    # Classify the clusters! (loop through each detected cluster one at a time)
    detected_objects_labels = []
    detected_objects = []
```

To start, two blank lists are created, one specifically for the labels (meaning the names of the objects, such as soap, biscuits, etc.) and one for the full set of information for each object (label, pointcloud, etc.), the latter of which would be used by the mover function to reach and place the objects.

```
    for index, pts_list in enumerate(cluster_indices):
        # Grab the points for the cluster
        pcl_cluster = extracted_outliers.extract(pts_list)
        ros_cluster = pcl_to_ros(pcl_cluster)
```

In going through each index cluster in “cluster_indices”, first the indices of a given object are converted to a PCL point cluster and then to a ROS point cluster.

```
        # Compute the associated feature vector
        chists = compute_color_histograms(ros_cluster, using_hsv=True)
        normals = get_normals(ros_cluster)
        nhists = compute_normal_histograms(normals)
        feature = np.concatenate((chists, nhists))
```

With the ROS cluster in hand for a given object, feature histograms can be made. First, concatenated HSV color histograms are made using a function from sensor_stick.features. Next, normal vectors of the object's surface are collected and turned into concatenated histograms using yet another function from sensor_stick.features. Both of these data sets are then concatenated into one feature set for use by the SVM.

```
    # Make the prediction
    prediction = clf.predict(scaler.transform(feature.reshape(1,-1)))
    label = encoder.inverse_transform(prediction)[0]
    detected_objects_labels.append(label)
```

Having the feature set, the previously trained SVM can be used to make a prediction, as is done above using the "clf" classifier object (after the scaled and reshaped to the size expected by the SVM). The label, or object name, associated with each prediction is then added to the appropriate list.

```
    # 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))
```

In order to best visualize the SVM prediction, the label name is published to RViz by taking the XYZ points of the object associated with the given label and moving up the z-axis position of the label 40 cm (so that the words aren't directly on the object).

```
    # 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
  detected_objects_pub.publish(detected_objects)
```

The rest of this 'for' loop is used to append the 'detected_objects' list. This is done using the custom 'DetectedObject()' message type, so information is added to it in the

appropriate fashion. Both the label and the pointcloud cluster of the detected object is added, and then the completed message is appended to the 'detected_objects' list.

After the entire 'for' loop is performed for each of the objects, the completed list of labels is published on the command prompt screen using 'rospy.loginfo'. The list of detected objects is also published via its appropriate publisher.

```
    # Suggested location for where to invoke your pr2_mover() function within
    # pcl_callback()
    # Could add some logic to determine whether or not your object detections are robust
    # before calling pr2_mover()
    try:
        pr2_mover(detected_objects)
    except rospy.ROSInterruptException:
        pass
```

At this point, the 'pr2_mover' function is invoked using the list of detected objects. The code for the function is described in detail below:

```
# function to load parameters and request PickPlace service
def pr2_mover(object_list):

    # TODO: Initialize variables
    labels = []
    centroids = []
    dict_list = []

    object_name = String()
    arm_name = String()
    pick_pose = Pose()
    place_pose = Pose()
```

For this function, a number of variables need to be created. A list is made for labels, centroids, as well as the yaml dictionaries that will be compiled at the end of function. Additionally, each variable to be used for the pick/place service, such as object name and arm name, is initialized as its desired message type (string type for the object and arm names, and the geometric message Pose() for both the pick and place poses).

```
    # TODO: Get/Read parameters
    object_list_param = rospy.get_param('/object_list')
```

```
    dropbox_param = rospy.get_param('/dropbox')
    test_scene_num = Int32()
    test_scene_num.data = 2
```

Next, a list of parameters is attained from two separate topics (one from the object list topic and one from the dropbox). These will be used later. Additionally, since the test scene number will not change for the entire instance of this function, a message of type 'Int32()' is initialized and assigned (the integer 2 in this particular instance since pick list 2 is used).

```
    # TODO: Rotate PR2 in place to capture side tables for the collision map
```

Not performed in this function.

```
    # TODO: Loop through the pick list
    for x in object_list:
        # TODO: Get the PointCloud for a given object and obtain its centroid.
            labels.append(x.label)

            pc2 = ros_to_pcl(x.cloud)
            pc2_xyz = XYZRGB_to_XYZ(pc2)
            pc2_xyz_arr = pc2_xyz.to_array()

            v = np.mean(pc2_xyz_arr[:,0])
            w = np.mean(pc2_xyz_arr[:,1])
            y = np.mean(pc2_xyz_arr[:,2])

            centroids.append([np.asscalar(v),np.asscalar(w),np.asscalar(y)])
```

The first 'for' loop implemented goes through each object in "object_list" (which will be 'detected_objects' in this case). The label assigned to each object is added to the 'labels' list. Then, the pointcloud cluster is converted to PCL format. Following, the pointcloud is changed in type from XYZRGB to only have position data (type XYZ). Then, this pointcloud is transformed to an array, which effectively transforms the data into a matrix, rows corresponding to each point in the object cluster and columns corresponding to x,y,z position.

The mean of each axis is then derived and the centroids list is appended with the resulting x,y,z values, corresponding to the center of each object.

```
    for i in range(len(labels)):

        pick_pose.position.x = centroids[i][0]
        pick_pose.position.y = centroids[i][1]
        pick_pose.position.z = centroids[i][2]
```

Once the list of centroids is attained, yet another 'for' loop can be run. For each identified object, the position of the 'pick_pose' message is set to the x,y,z coordinates of said object's centroid.

```
        # TODO: Create 'place_pose' for the object

        object_name.data = object_list_param[i]['name']
        object_group = object_list_param[i]['group']

        if object_group == "red":
                position_l = dropbox_param[0]['position']
                place_pose.position.x = position_l[0]
                place_pose.position.y = position_l[1]
                place_pose.position.z = position_l[2]

        else:
                position_r = dropbox_param[1]['position']
                place_pose.position.x = position_r[0]
                place_pose.position.y = position_r[1]
                place_pose.position.z = position_r[2]
```

Creating the place_pose message is a bit simpler. First, the object's group is determined, which corresponds to the dropbox each object is supposed to be placed in (either the red box or the green box). This information is included as part of the parameters derived from the object_list topic. If the object group was found to be red, the place_pose message was set to the x,y,z coordinates for the red dropbox derived from the dropbox topic. Otherwise, meaning the green box is associated with the given object, the x,y,z position of that box is used for the place_pose message instead.

```
        # TODO: Assign the arm to be used for pick_place

        if object_group == "red":
                arm_name.data = "left"

        else:
```

```
            arm_name.data = "right"
```

In order to assign an arm for a given object, the object_group parameter was used again. If the object_group was assigned as “red”, the arm_name message’s data was assigned as “left” (since the red dropbox is by the left arm). Otherwise, the data was assigned as “right” (since the green dropbox is next to the right arm).

```
    # TODO: Create a list of dictionaries (made with make_yaml_dict()) for later output
to
    # yaml format
    yaml_dict = make_yaml_dict(test_scene_num, arm_name, object_name,
                pick_pose, place_pose)

    dict_list.append(yaml_dict)
```

Once all of the desired messages are filled with data, they can be made into a dictionary, using the ‘make_yaml_dict’ function from the start of the code. This function creates a standard Python dictionary, with each message defined as its own key. Once the dictionary is made, it is added to the ‘dict_list’.

```
    # Wait for 'pick_place_routine' service to come up
    rospy.wait_for_service('pick_place_routine')

    try:
        pick_place_routine = rospy.ServiceProxy('pick_place_routine', PickPlace)

        # TODO: Insert your message variables to be sent as a service request
        resp = pick_place_routine(test_scene_num, object_name, arm_name,
                pick_pose, place_pose)

        print ("Response: ",resp.success)

    except rospy.ServiceException, e:
        print "Service call failed: %s"%e
```

With all of the 5 messages assigned, the ‘pick_place_routine’ service can be run. The ros master first waits until the desired service comes up. Then, it tries the ‘pick_place_routine’, printing out whether the service succeeded or failed (or if it otherwise failed in some manner).

```
# TODO: Output your request parameters into output yaml file
send_to_yaml('output_2.yaml', dict_list)
```

To finish off the code, the completed 'dict_list' is copied to a pre-made output .yaml file using the 'send_to_yaml' helped function defined at the start of this program. For this particular instance, since test scene 2 was used, the output file was titled 'output_2.yaml'.

Miscellaneous Thoughts:

Overall, after much testing, the various values and parameters used in the program above functioned quite well to identify any and all objects in question, as well as locate/move the objects with the robotic arms. Having said that, though, there is certainly room to progress further. I did not spend much time considering object collision, so there could be times when an object would be bumped or moved by the robotic arm unintentionally, leading to problems down the line when the bumped object needed to be picked up. This would require further coding to allow the RGB-D camera to develop a more thorough collision map. After this, of course, there is the more challenging map to explore. This would certainly require more intricate filtering and collision mapping. The possible objects are now present in three possible viewing angles (not just directly in front of the robot), so the robot would need to be moved to properly spot all of the objects. Additionally, the objects lay at different elevations, so multiple passthrough filters would need to be made for each cluster in question. After each object is identified in this more complex environment, the rest of the code could function, more or less, the same, in terms of identifying the centroids and drop-off destination (simply with a more intricate collision map). With some additional work, I am confident I could make this happen.