Robot 3-D Perception Project

January 27, 2018

In the first three sections we discuss the general principles developed in exercises 1,2,3 applied to an example on how to prepare the scene 1 for object recognition. In the final section, we discuss how the parameters introduced by these general methods can be tweaked to fit object recognition in an environment with three different scatterings of objects.

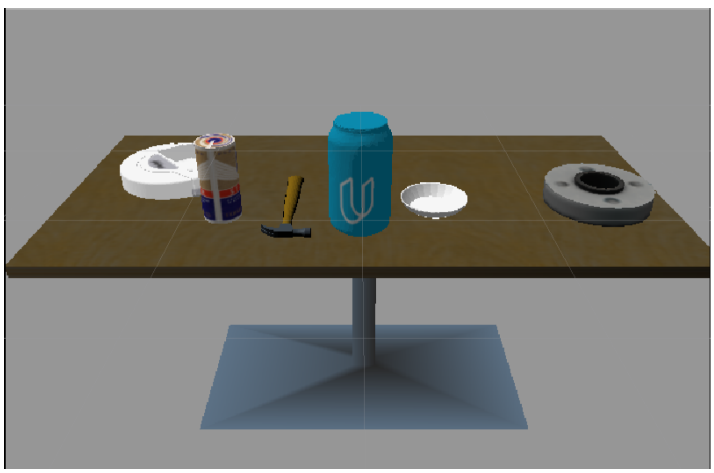

Figure 1

Exercise 1: Calibration, Filtering, and Segmentation

The first thing that we want to do is to isolate a point cloud containing only the objects of interest to be visualized in Rviz. To do this, we write a node that subscribes to the *sensor_stick/point_cloud* topic so that whenever a message of type point cloud arrives it is filtered through the *pcl_callback*() function. This function returns segmented images of the table the objects sit on and the objects themselves and publishes them to Rviz for view. Next, we go over the steps of this filtering process.

First, we implement voxel grid downsampling to get a lower resolution point cloud that still captures the most important features of each object. Done correctly, decreasing the density in the point clouds allows for less compuation time with minimal loss of object features. After turning the ROS message to PCL, the code to implement it is given by the snippet:

```
# Convert ROS msg to PCL data
    cloud = ros_to_pcl(ros_msg)
# Create a VoxelGrid filter object for our input point cloud
    vox = cloud.make_voxel_grid_filter()

    # Choose a voxel (also known as leaf) size
    LEAF_SIZE = .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()
```

The following 2 is our image of interest after passing a VoxelGrid Downsampling Filter with leaf size .01 to it.

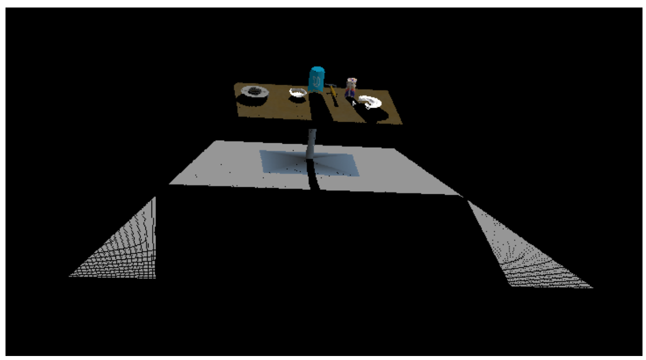

Figure 2

The next step is to crop the image to produce the region of interest. The code to crop the image after it has been calibrated by VoxelGrid Downsampling and saved as *cloud_filtered* is given by:

```
# 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 = .76
    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()
```

The following 3 shows a Pass Through Filter running through the vertical z-axis cutting off the region below .76 and above 1.1. What remains in the image is simply the objects on the table and the top of the table.

Finally we run the RANSAC algorithm to segment the table from the objects on the table. The code to implement this is given by:

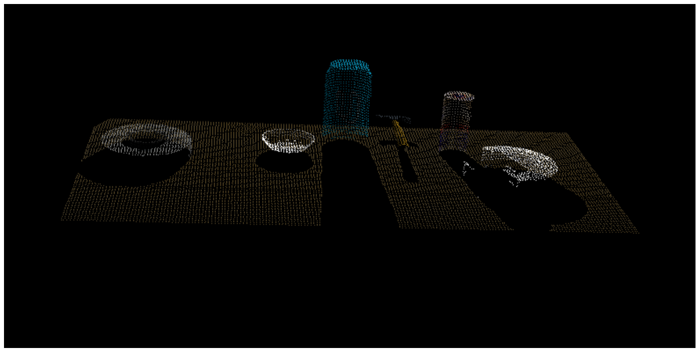

Figure 3

```
# Create RANSAC plane segmentation
    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
    max_distance = .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 (table)
    extracted_inliers = cloud_filtered.extract(inliers, negative=
    False)

    # Extract outliers (objects)
    extracted_outliers = cloud_filtered.extract(inliers, negative=
    True)
```

By choosing a best fit model of a plane, any points a max distance of .01 away from it is considered an inlier and is an indice extracted by the RANSAC model to produce the set of points classed as table 4.

Anything that lies outside the range of inliers is considered an outlier and can also be extracted as such which leaves us with the set of objects on the table 5.

Exercise 2: Clustering for Segmentation

After the calibration, filtering, and segmentation in exercise 1, we proceed to further distinguish between the objects by applying an Euclidean Clustering algorithm (a.k.a. DBSCAN algorithm) that identifies separate clusters of points for each object depending on how near they are to each other and casts them into different colors. We create another publisher so that the *pcl_callback*() function publishes a cluster cloud updated by the Euclidean Clustering algorithm onto Rviz. After, we set the cluster tolerance for point distance threshold to be .05,

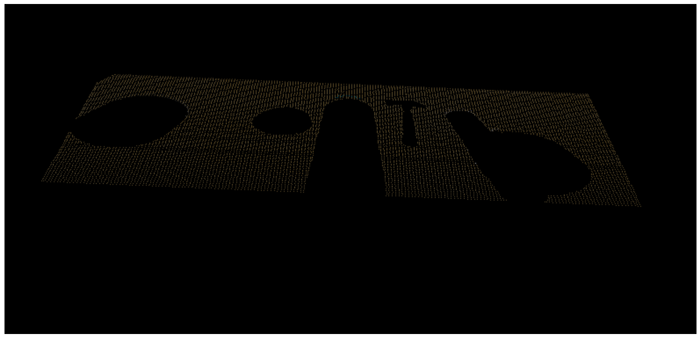

Figure 4

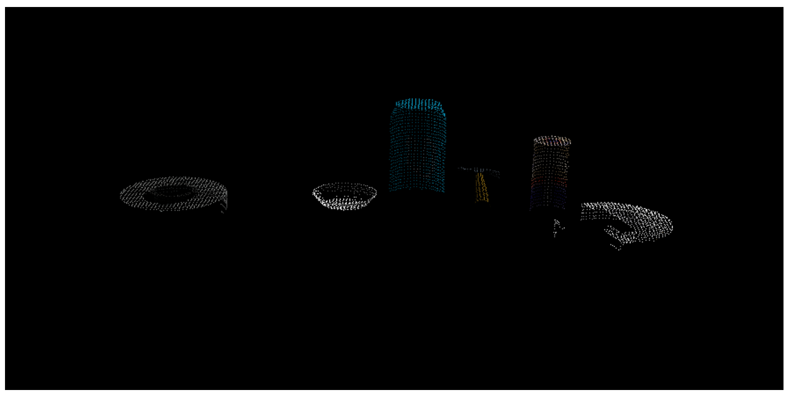

Figure 5

set the minimum cluster size to be 10 and the maximum cluster size to be 2000 to obtain the following segmentation of objects 6.

The code to cluster is given by:

```
# Euclidean Clustering
    white_cloud = XYZRGB_to_XYZ(extracted_outliers)# Apply function to convert XYZRGB to XYZ
    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)
    ec.set_ClusterTolerance(0.05)
    ec.set_MinClusterSize(10)
    ec.set_MaxClusterSize(2000)
    # Search the k-d tree for clusters
    ec.set_SearchMethod(tree)
    # Extract indices for each of the discovered clusters
    cluster_indices = ec.Extract()

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

```

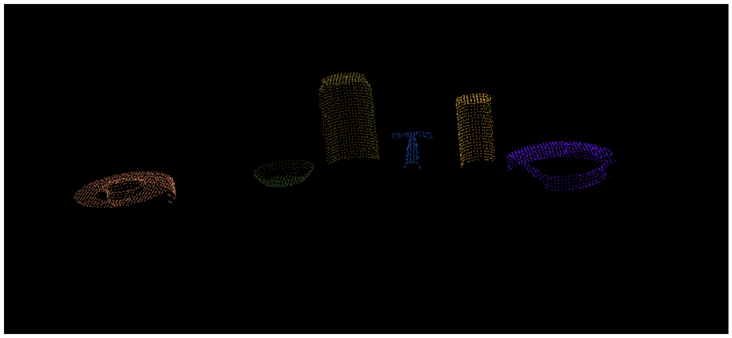

Figure 6

```
    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])])

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

The maximum cluster size is typically set to be the point size of the largest object to be classified and the minimum cluster size should be small enough to account for all the points in the smallest object. The cluster tolerance should take into consideration voxel grid leaf size making sure that it is large enough so that the minimum cluster size is fulfilled to get a large enough cluster class to create core cluster members and one should make sure that it is small enough so that neighboring objects are not classified in the same class as a single object. Further, the extra feature of the DBSCAN algorithm is that when the tolerance of an epsilon ball is set to be small enough, the noise that was not originally filtered out of the statistical outlier filter and the RANSAC filter is taken to be an outlier to any DBSCAN clustered object and is therefore filtered out.

Exercise 3: Object Recognition

Finally, after we have detected the number of objects in the image, we can identify what objects they are by employing a machine learning algorithm. We first find features that capture the properties of the objects. In this case, we will be using HSV channels and x,y,z normal orientations as features to assign labels to the objects. We encode this information in the form of histogram data with the following code. First, we define the *compute_color_histograms*() function which partitions the range of the color channels into 50 bins for each channel: H, S, and V.

```python
def compute_color_histograms(cloud, using_hsv=False):
    # Compute histograms for the clusters
    point_colors_list = []

    # Step through each point in the point cloud
    for point in pc2.read_points(cloud, skip_nans=True):
        rgb_list = float_to_rgb(point[3])
        if using_hsv:
            point_colors_list.append(rgb_to_hsv(rgb_list) * 255)
        else:
            point_colors_list.append(rgb_list)

    # Populate lists with color values
    channel_1_vals = []
    channel_2_vals = []
    channel_3_vals = []

    for color in point_colors_list:
        channel_1_vals.append(color[0])
        channel_2_vals.append(color[1])
        channel_3_vals.append(color[2])

    # Compute histograms
    chan_1_hist = np.histogram(channel_1_vals, bins=50, range=(0, 256))
    chan_2_hist = np.histogram(channel_2_vals, bins=50, range=(0, 256))
    chan_3_hist = np.histogram(channel_3_vals, bins=50, range=(0, 256))

    # Concatenate and normalize the histograms
    hist_features = np.concatenate((chan_1_hist[0], chan_2_hist[0], chan_3_hist[0])).astype(np.float64)
    normed_features = hist_features / np.sum(hist_features)

    return normed_features
```

Second, we define the function *compute_normal_histograms*() which partitions the range of normals in the x,y,z directions into 10 bins each.

```python
def compute_normal_histograms(normal_cloud):
    norm_x_vals = []
    norm_y_vals = []
    norm_z_vals = []

    for norm_component in pc2.read_points(normal_cloud,
                                          field_names=('normal_x', 'normal_y', 'normal_z'),
                                          skip_nans=True):
        norm_x_vals.append(norm_component[0])
        norm_y_vals.append(norm_component[1])
        norm_z_vals.append(norm_component[2])

    # Compute histograms of normal values
    norm_x_hist = np.histogram(norm_x_vals, bins=10, range=(-1,1))
    norm_y_hist = np.histogram(norm_y_vals, bins=10, range=(-1,1))
    norm_z_hist = np.histogram(norm_z_vals, bins=10, range=(-1,1))

    # Concatenate and normalize the histograms
    hist_features = np.concatenate((norm_x_hist[0], norm_y_hist[0], norm_z_hist[0])).astype(np.float64)
    normed_features = hist_features / np.sum(hist_features)

```

```
    return normed_features
```

Having set this up, we can start collecting samples of our objects observed from different perspectives from our training.launch environment in gazebo. For each object, we take 200 samples of the object. Note that as we increase the number of samples taken to train our model, the better our classifier will be to correctly identify our object. To run the program that captures these features and saves these labeled features in a .sav file we have the code in our *capture_features.py* script:

```
#!/usr/bin/env python
import numpy as np
import pickle
import rospy

from sensor_stick.pcl_helper import *
from sensor_stick.training_helper import spawn_model
from sensor_stick.training_helper import delete_model
from sensor_stick.training_helper import initial_setup
from sensor_stick.training_helper import capture_sample
from sensor_stick.features import compute_color_histograms
from sensor_stick.features import compute_normal_histograms
from sensor_stick.srv import GetNormals
from geometry_msgs.msg import Pose
from sensor_msgs.msg import PointCloud2


def get_normals(cloud):
    get_normals_prox = rospy.ServiceProxy('/feature_extractor/
    get_normals', GetNormals)
    return get_normals_prox(cloud).cluster


if __name__ == '__main__':
    rospy.init_node('capture_node')
    test_num = 0
    models=[]

    models.append([\
        'beer',
        'bowl',
        'create',
        'disk_part',
        'hammer',
        'plastic_cup',
        'soda_can'])

    models.append([\
        'biscuits',
        'soap',
        'soap2'])

    models.append([\
        'biscuits',
        'soap',
        'book',
        'soap2',
        'glue'])

    models.append([\
        'sticky_notes',
        'book',
```

```
        'snacks',
        'biscuits',
        'eraser',
        'soap2',
        'soap',
        'glue'])
    # Disable gravity and delete the ground plane
    initial_setup()
    labeled_features = []

    for model_name in models[test_num]:
        spawn_model(model_name)

        for i in range(200):
            # make 200 samples
            sample_was_good = False
            try_count = 0
            while not sample_was_good and try_count < 200:
                sample_cloud = capture_sample()
                sample_cloud_arr = ros_to_pcl(sample_cloud).to_array()

                # Check for invalid clouds.
                if sample_cloud_arr.shape[0] == 0:
                    print('Invalid cloud detected')
                    print(model_name)
                    try_count += 1
                else:
                    sample_was_good = True

            # Extract histogram features
            chists = compute_color_histograms(sample_cloud, using_hsv=True)
            normals = get_normals(sample_cloud)
            nhists = compute_normal_histograms(normals)
            feature = np.concatenate((chists, nhists))
            labeled_features.append([feature, model_name])

        delete_model()

    if test_num == 0:
        pickle.dump(labeled_features, open('training_set0.sav', 'wb'))
    elif test_num == 1:
        pickle.dump(labeled_features, open('training_set1.sav', 'wb'))
    elif test_num == 2:
        pickle.dump(labeled_features, open('training_set2.sav', 'wb'))
    elif test_num == 3:
        pickle.dump(labeled_features, open('training_set3.sav', 'wb'))
```

After we are done taking samples, we create an SVM classifier with sigmoid kernel in *train_svm.py* script and save our trained classifier in a .save file for later use.

```
#!/usr/bin/env python
import pickle
import itertools
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
```

```python
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn import cross_validation
from sklearn import metrics
from sklearn.ensemble import RandomForestClassifier

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, '{0:.2f}'.format(cm[i, j]),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Load training data from disk
test_num = 0
if test_num == 0:
    training_set = pickle.load(open('training_set0.sav', 'rb'))
elif test_num == 1:
    training_set = pickle.load(open('training_set1.sav', 'rb'))
elif test_num == 2:
    training_set = pickle.load(open('training_set2.sav', 'rb'))
elif test_num == 3:
    training_set = pickle.load(open('training_set3.sav', 'rb'))

# Format the features and labels for use with scikit learn
feature_list = []
label_list = []

for item in training_set:
    if np.isnan(item[0]).sum() < 1:
        feature_list.append(item[0])
        label_list.append(item[1])

print('Features in Training Set: {}'.format(len(training_set)))
print('Invalid Features in Training set: {}'.format(len(training_set)-len(feature_list)))

X = np.array(feature_list)
# Fit a per-column scaler
X_scaler = StandardScaler().fit(X)
# Apply the scaler to X
X_train = X_scaler.transform(X)
```

```
y_train = np.array(label_list)

# Convert label strings to numerical encoding
encoder = LabelEncoder()
y_train = encoder.fit_transform(y_train)

# Create classifier
clf = svm.SVC(kernel='sigmoid')

# Set up 5-fold cross-validation
kf = cross_validation.KFold(len(X_train),
                            n_folds=5,
                            shuffle=True,
                            random_state=1)

# Perform cross-validation
scores = cross_validation.cross_val_score(cv=kf,
                                          estimator=clf,
                                          X=X_train,
                                          y=y_train,
                                          scoring='accuracy'
                                         )
print('Scores: ' + str(scores))
print('Accuracy: %0.2f (+/- %0.2f)' % (scores.mean(), 2*scores.std()))

# Gather predictions
predictions = cross_validation.cross_val_predict(cv=kf,
                                          estimator=clf,
                                          X=X_train,
                                          y=y_train
                                         )

accuracy_score = metrics.accuracy_score(y_train, predictions)
print('accuracy score: '+str(accuracy_score))

confusion_matrix = metrics.confusion_matrix(y_train, predictions)

class_names = encoder.classes_.tolist()


#Train the classifier
clf.fit(X=X_train, y=y_train)

model = {'classifier': clf, 'classes': encoder.classes_, 'scaler': X_scaler}

# Save classifier to disk
if test_num == 0:
    pickle.dump(model, open('model0.sav', 'wb'))
elif test_num == 1:
    pickle.dump(model, open('model1.sav', 'wb'))
elif test_num == 2:
    pickle.dump(model, open('model2.sav', 'wb'))
elif test_num == 3:
    pickle.dump(model, open('model3.sav', 'wb'))

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(confusion_matrix, classes=encoder.classes_,
                      title='Confusion matrix, without normalization')
```

```python

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(confusion_matrix, classes=encoder.classes_,
    normalize=True,
                      title='Normalized confusion matrix')

plt.show()
```

Notice that we also have a print out of the confusion matrices that tell us the accuracy of our model 7. The overall accuracy for our model is approximately 92%.

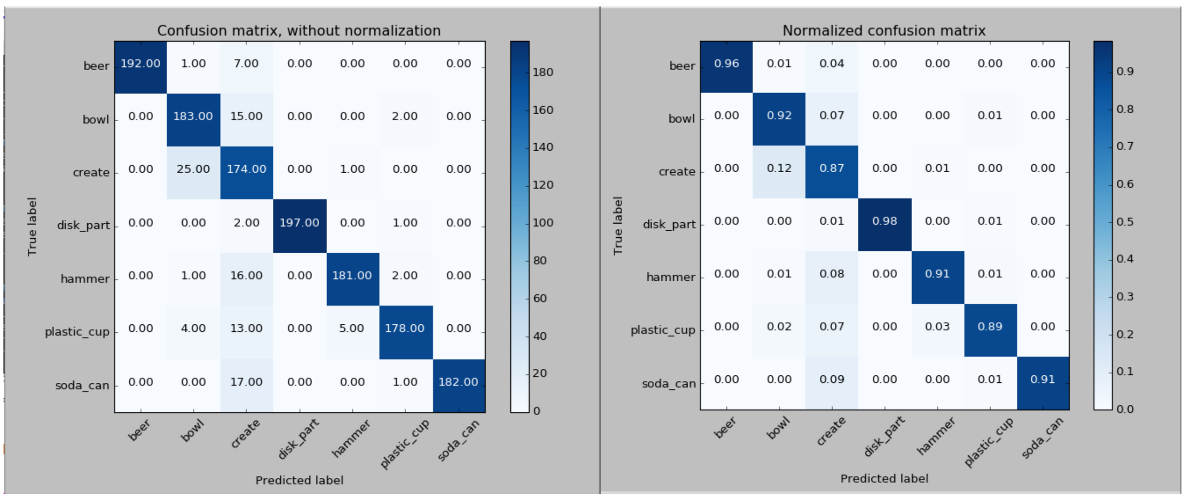

Figure 7

Finally, we can implement a classification by running our *object_recognition.py* script. The result is given in 8

Testing 3-D Perception Environments

Putting everything together, we test out our object recognition pipeline in a different environment. We have the code given in *project_template.py*:

```python
#!/usr/bin/env python

# 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 *
```

Figure 8

```python

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
import matplotlib.pyplot as plt


# 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


# 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


# 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)


# 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)
    # TODO: Statistical Outlier Filtering
    outlier_filter = cloud.make_statistical_outlier_filter()

    # Set the number of neighboring points to analyze for any given point
    outlier_filter.set_mean_k(50)

    # Set threshold scale factor
    x = 1.0

    # Any point with a mean distance larger than global (mean distance+x*std_dev) will be considered outlier
    outlier_filter.set_std_dev_mul_thresh(x)

    # Finally call the filter function for magic
    cloud = outlier_filter.filter()
    # TODO: Voxel Grid Downsampling
    # 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 = .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_final.pcd'
    pcl.save(cloud_filtered, filename)
    # TODO: PassThrough Filter
    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 = .6
    axis_max = .8
    passthrough.set_filter_limits(axis_min, axis_max)

    # Finally use the filter function to obtain the resultant point cloud.
    cloud_filtered = passthrough.filter()

    # Make another passthrough filter through x-axis
    passthrough = cloud_filtered.make_passthrough_filter()
    filter_axis = 'x'
    passthrough.set_filter_field_name(filter_axis)
    axis_min = .4
```

```
    axis_max = 1
    passthrough.set_filter_limits(axis_min, axis_max)
    cloud_filtered = passthrough.filter()

    # Make another passthrough filter through y-axis
    passthrough = cloud_filtered.make_passthrough_filter()
    filter_axis = 'y'
    passthrough.set_filter_field_name(filter_axis)
    axis_min = -.6
    axis_max = .4
    passthrough.set_filter_limits(axis_min, axis_max)
    cloud_filtered = passthrough.filter()

    filename = 'pass_through_filtered_final.pcd'
    pcl.save(cloud_filtered, filename)
    # 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 = .01
    seg.set_distance_threshold(max_distance)
    # TODO: Extract inliers and outliers
    # 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_final.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_final.pcd'
    pcl.save(extracted_outliers, filename)

    # TODO: Euclidean Clustering
    white_cloud = XYZRGB_to_XYZ(extracted_outliers)  # Apply function to convert XYZRGB to XYZ
    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(0.02)
    ec.set_MinClusterSize(50)
    ec.set_MaxClusterSize(2000)
```

```
    # Search the k-d tree for clusters
    ec.set_SearchMethod(tree)
    # Extract indices for each of the discovered clusters
    cluster_indices = ec.Extract()

    # 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])])

    # Create new cloud containing all clusters, each with unique color
    cluster_cloud = pcl.PointCloud_PointXYZRGB()
    cluster_cloud.from_list(color_cluster_point_list)
    filename = 'cluster_final.pcd'
    pcl.save(cluster_cloud, filename)
    # TODO: Create Cluster-Mask Point Cloud to visualize each cluster separately
    cloud_objects = extracted_outliers
    cloud_table = extracted_inliers
    # TODO: Convert PCL data to ROS messages
    ros_cloud_objects = pcl_to_ros(cloud_objects)
    ros_cloud_table = pcl_to_ros(cloud_table)
    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_cluster_pub.publish(ros_cluster_cloud)
    # Exercise-3 TODOs:

    # Classify the clusters! (loop through each detected cluster one at a time)
    detected_objects_labels = []
    detected_objects_list = []
    for index, pts_list in enumerate(cluster_indices):
        # Grab the points for the cluster
        pcl_cluster = cloud_objects.extract(pts_list)
        ros_cluster = pcl_to_ros(pcl_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))
        # Make the prediction
        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
```

```python
        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_list.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_list)
    # 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()
    if len(detected_objects_labels) > 0:
        try:
            pr2_mover(detected_objects_list)
        except rospy.ROSInterruptException:
            pass
    else:
        print("No objects detected.")


# function to load parameters and request PickPlace service
def pr2_mover(object_list):
    # TODO: Initialize variables
    test_scene_num = Int32()
    object_name = String()
    object_group = String()
    arm_name = String()
    pick_pose = Pose()
    place_pose = Pose()
    dict_list = []
    # TODO: Get/Read parameters
    object_list_param = rospy.get_param('/object_list')
    drop_position = rospy.get_param('/dropbox')
    test_scene_num.data = test_num
    # TODO: Parse parameters into individual variables
    drop_position_right = drop_position[1]['position']
    drop_position_left = drop_position[0]['position']
    # TODO: Rotate PR2 in place to capture side tables for the collision map

    # TODO: Loop through the pick list
    for i in range(0, len(object_list_param)):
        object_name.data = object_list_param[i]['name']
        object_group.data = object_list_param[i]['group']
        # TODO: Get the PointCloud for a given object and obtain it's centroid
        labels = []
        centroids = []
        for objects in object_list:
            labels.append(objects.label)
            points_arr = ros_to_pcl(objects.cloud).to_array()
            centroids.append(np.mean(points_arr, axis=0)[:3])
        if all(object_name.data != labels[j] for j in range(0, len(labels))):
```

```
            dict_list.append("%s not detected." % (object_name.data))
        else:
            # TODO: Assign the arm to be used for pick_place
            if object_group.data == 'green':
                arm_name.data = 'right'
            elif object_group.data == 'red':
                arm_name.data = 'left'
            # TODO: Create 'place_pose' for the object
            if arm_name.data == 'right':
                place_pose.position.x = drop_position_right[0]
                place_pose.position.y = drop_position_right[1]
                place_pose.position.z = drop_position_right[2]
            elif arm_name.data == 'left':
                place_pose.position.x = drop_position_left[0]
                place_pose.position.y = drop_position_left[1]
                place_pose.position.z = drop_position_left[2]
            # TODO: Create 'pick_pose' for the object
            for j in range(0, len(labels)):
                if object_name.data == labels[j]:
                    pick_pose.position.x = np.asscalar(centroids[j][0])
                    pick_pose.position.y = np.asscalar(centroids[j][1])
                    pick_pose.position.z = np.asscalar(centroids[j][2])
                    # 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)
                    # 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

    # TODO: Output your request parameters into output yaml file
    if test_scene_num.data == 1:
        send_to_yaml('output_1.yaml', dict_list)
    elif test_scene_num.data == 2:
        send_to_yaml('output_2.yaml', dict_list)
    elif test_scene_num.data == 3:
        send_to_yaml('output_3.yaml', dict_list)


if __name__ == '__main__':

    # TODO: ROS node initialization
    test_num = 3
```

```
    rospy.init_node('Perception', anonymous=True)
    # TODO: Create Subscribers
    pcl_sub = rospy.Subscriber("pr2/world/points", pc2.PointCloud2,
     pcl_callback, queue_size=1)
    # 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_cluster_pub = rospy.Publisher("/pcl_cluster", 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)
    # TODO: Load Model From disk
    if test_num == 1:
        model = pickle.load(open('model1.sav', 'rb'))
    elif test_num == 2:
        model = pickle.load(open('model2.sav', 'rb'))
    elif test_num == 3:
        model = pickle.load(open('model3.sav', 'rb'))

    clf = model['classifier']
    encoder = LabelEncoder()
    encoder.classes_ = model['classes']
    scaler = model['scaler']
    # Initialize color_list
    get_color_list.color_list = []

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

Notice that we have tweaked the parameters for calibration, filtration, and cluster segmentation. We have also added a filter to take out the noise in the images. In VoxelGrid Downsampling, we have maintained the same leaf size as in exercise 1 at .01. In Pass Through Filteration, we have changed the z-axis bounds for croping to be from .6 to .8 and have added a x-axis and y-axis pass through filters with .4 to 1 and $-.6$ to .4 bounds respectively. This effectively isolates the region of interest of object and table that we are trying to classify. On the other hand, the RANSAC parameter for table segmentation remains the same at .01. These parameters are suitable to be used in all three worlds.

However, in Euclidean Clustering, we experiment with new tolerance of .02, minimum cluster size of 100 and maximum cluster size of 2000 which proves to work for the first two worlds, but fails to segment the glue from the book in the third world. A solution to distinguish the glue from the book is to simply lower the minimum cluster size from 100 to 50. This proves a reasonable suggestion as follows from previous discussion on Euclidean Clustering that we must choose a minimum cluster size that does not exceed the maximum number of points clustered as glue because it is the smallest object in point size that needs to be classified. Further, one may ask why segmentation would work perfectly with the given parameters of the DBSCAN algorithm in worlds one and two. At first glance, in comparison of the voxel grid downsampling leaf size to the DBSCAN parameters, one would assume the cluster tolerance to be too small and minimum cluster size too large so that no core cluster members could be found and all members would be cast as edge members or outliers.

However, the DBSCAN algorithm is run on the 2-d projection of the 3-d voxel partitioned point cloud in the direction facing the original image. This means in some regions where there is extension of the object into the 3rd dimension, the density of points in this 2-d projection is not always so that each point is .01 distance away from the next point as set by the voxel downsampled leaf size which accounts for the reason why core points with min cluster size of 100 points and a radius size of .02 is possible to be found in order to initiate a cluster classification made together with its density reachable points.

The parameters implemented for object recognition for the three different worlds differ first in choice of training number. In world 1 the number of training examples used was 50 for each object. In world 2 the number of training examples used was 100 for each object and the number of training examples used for world 3 was 200. The binning schemes that were prescribed were also different. In world 1 the binning scheme for HSV was 256 bins for each channel of range $(0, 256)$ while the bins for the normals were 10 for each normalized direction with range $(-1, 1)$. World 2 and 3 shared the same HSV and normal bin size at 50 for each channel of HSV and 10 for each channel of normal direction within the usual range possibilities. Finally, the classifier that was used for all three worlds was the SVM with sigmoid kernel. After being trained, all three worlds had high accuracy ranges at above 90% and classified all the objects in their respective worlds correctly. Observe the results here: 9, 10, 11.

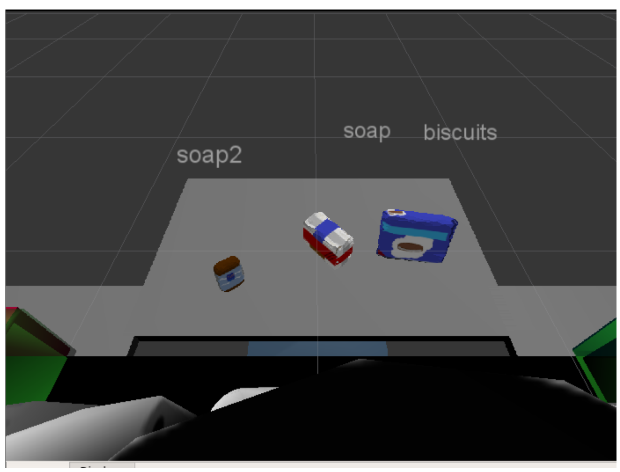

Figure 9

The respective .yaml files of the three worlds are also listed below:

```
WORLD 1
object_list:
- arm_name: right
  object_name: biscuits
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
```

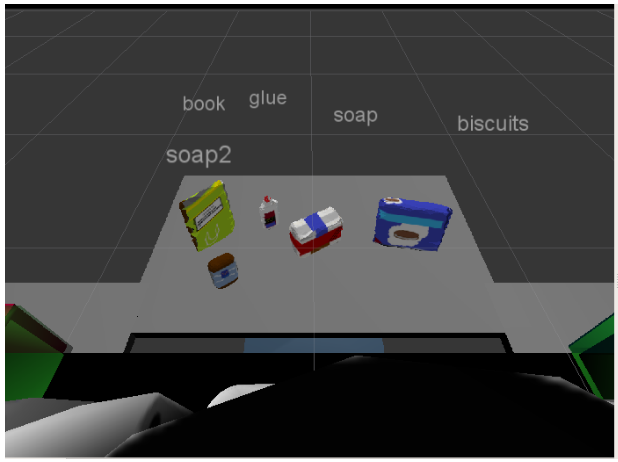

Figure 10


```
      x: 0.54229336977005
      y: -0.24221345782279968
      z: 0.7052517533302307
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: -0.71
      z: 0.605
  test_scene_num: 1
- arm_name: right
  object_name: soap
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.5437864661216736
      y: -0.01916990615427494
      z: 0.6744173169136047
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: -0.71
      z: 0.605
  test_scene_num: 1
- arm_name: left
  object_name: soap2
```

Figure 11

```
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.4480173885822296
      y: 0.2213219851255417
      z: 0.6799272298812866
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: 0.71
      z: 0.605
  test_scene_num: 1
```

```
WORLD 2
object_list:
- arm_name: right
  object_name: biscuits
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.5717878937721252
      y: -0.24866198003292084
      z: 0.7050904035568237
  place_pose:
    orientation:
      w: 0.0
```

```
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: -0.71
      z: 0.605
  test_scene_num: 2
- arm_name: right
  object_name: soap
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.5608817338943481
      y: 0.0031042008195072412
      z: 0.6749479174613953
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: -0.71
      z: 0.605
  test_scene_num: 2
- arm_name: left
  object_name: book
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.5806224942207336
      y: 0.2783866226673126
      z: 0.7099695801734924
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: 0.71
      z: 0.605
  test_scene_num: 2
- arm_name: left
  object_name: soap2
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
```

```
    position:
      x: 0.4480023980140686
      y: 0.22405624389648438
      z: 0.6799744367599487
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: 0.71
      z: 0.605
  test_scene_num: 2
- arm_name: left
  object_name: glue
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.6311597228050232
      y: 0.13127191364765167
      z: 0.6789787411689758
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: 0.71
      z: 0.605
  test_scene_num: 2
```

```
WORLD 3
object_list:
- arm_name: left
  object_name: sticky_notes
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.4395598769187927
      y: 0.21759212017059326
      z: 0.6820430755615234
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: 0.71
      z: 0.605
```

```yaml
  test_scene_num: 3
- arm_name: left
  object_name: book
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.4911796748638153
      y: 0.08414322137832642
      z: 0.7110996842384338
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: 0.71
      z: 0.605
  test_scene_num: 3
- arm_name: right
  object_name: snacks
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.42847874760627747
      y: -0.33822470903396606
      z: 0.7092233896255493
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: -0.71
      z: 0.605
  test_scene_num: 3
- arm_name: right
  object_name: biscuits
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.5894694328308105
      y: -0.2184552401304245
      z: 0.7043547630310059
  place_pose:
    orientation:
      w: 0.0
```

```
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: -0.71
      z: 0.605
  test_scene_num: 3
- arm_name: left
  object_name: eraser
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.6071921586990356
      y: 0.28297457098960876
      z: 0.6465416550636292
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: 0.71
      z: 0.605
  test_scene_num: 3
- arm_name: right
  object_name: soap2
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.45495903491973877
      y: -0.04433096945285797
      z: 0.6737387776374817
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: -0.71
      z: 0.605
  test_scene_num: 3
- arm_name: right
  object_name: soap
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
```

```
    position:
      x: 0.6789422631263733
      y: 0.004983594175428152
      z: 0.6749618053436279
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: -0.71
      z: 0.605
  test_scene_num: 3
- arm_name: left
  object_name: glue
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.611974835395813
      y: 0.1423550844192505
      z: 0.6811114549636841
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: 0.71
      z: 0.605
  test_scene_num: 3
```