Perception project

Write-up by Dalia Vitkauskaite (daliavi@gmail.com)

Project requirements

In the PR2 Pick and Place simulator, there are three different tabletop configurations. The PR2 robot is equipped with RGB-D camera. The main objective for the project is to implement a perception pipeline to handle the camera data.

For a passing submission, your code must succeed in recognizing:

- 100% (3/3) objects in test1.world
- 80% (4/5) objects in test2.world
- 75% (6/8) objects in test3.world

A successful pick and place operation involves passing the correct request parameters to the pick_place_server. Hence, for each test scene, correct values must be output to .yaml format for following parameters (see /pr2_robot/config/output.yaml for an example):

- Object Name (Obtained from the pick list)
- Arm Name (Based on the group of an object)
- Pick Pose (Centroid of the recognized object)
- Place Pose (Not a requirement for passing but needed to make the PR2 happy)
- Test Set Number

Name your output files output_1.yaml, output_2.yaml, and output_3.yaml, respectively.

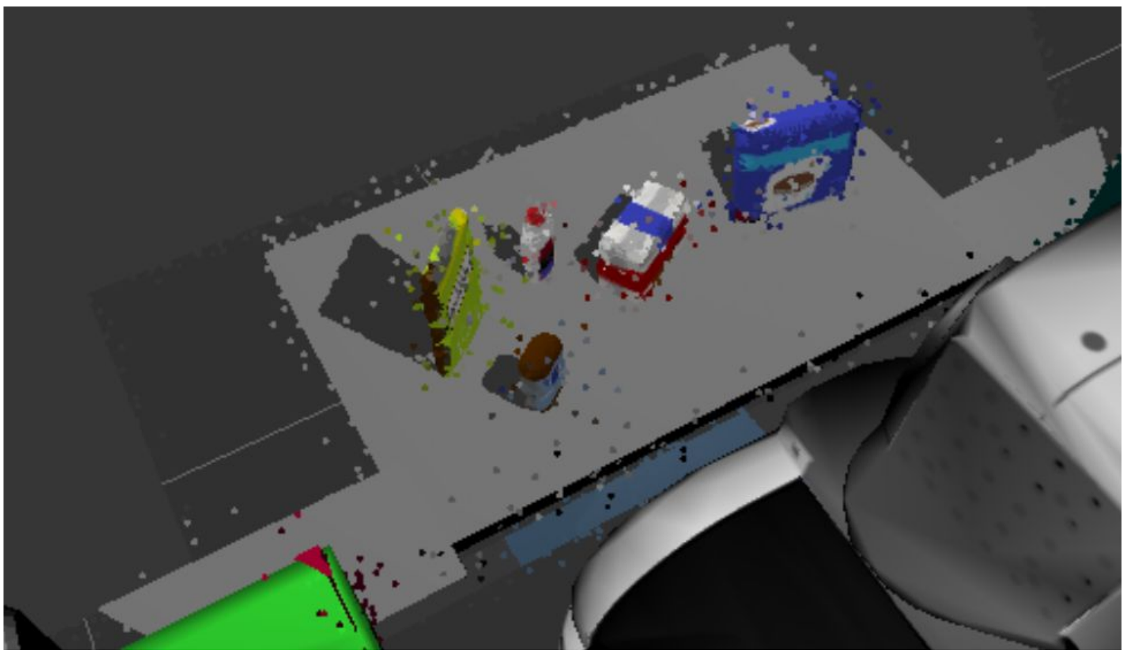
Filtering and Segmentation (Exercise 1)

With the 3D point clouds with majority of the data corresponds to an unwanted background objects or other things that must be filtered out before you can perform object detection or recognition.

Using Exercise 1 steps I implemented the pipeline for filtering and RANSAC plane fitting.

I filled in the `pcl_callback()` function with the code shown below in this section. I am going to use code samples and images from the Test 2 world scenario, as it was the last configuration I tested my project with.

The original image from the robots camera came with some noise. Here is an example from the Test 2 world:

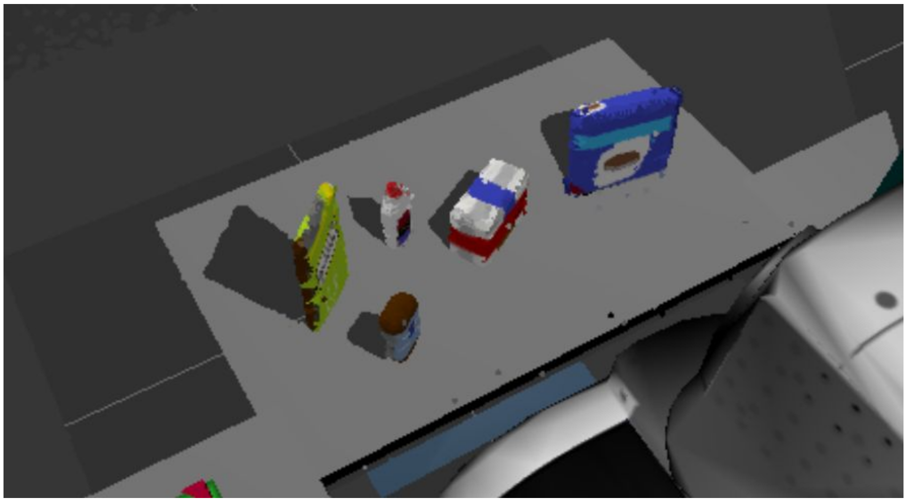


Statistical outlier filter

There was one extra step in the project, compared to the Exercise 1. I had to implement Statistical outlier filter, to remove the noise from the image.

```
def statistical_outlier_filter(cloud):
   # creating a filter object
   outlier_filter = cloud.make_statistical_outlier_filter()
   # Set the number of neighboring points to analyze for any given point
   outlier_filter.set_mean_k(20)
   # Set threshold scale factor
   x = 0.1
   # 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_filtered = outlier_filter.filter()
   return cloud_filtered
```

The image below demonstrates the results after applying statistical outlier filter:



Voxel Filter

RGB-D cameras provide feature rich and dense point clouds. Running computation on a full resolution point cloud can be slow and may not yield any improvement on results obtained using a more sparsely sampled point cloud. Therefore the next step in the image perception pipeline was Voxel Grid Filter.

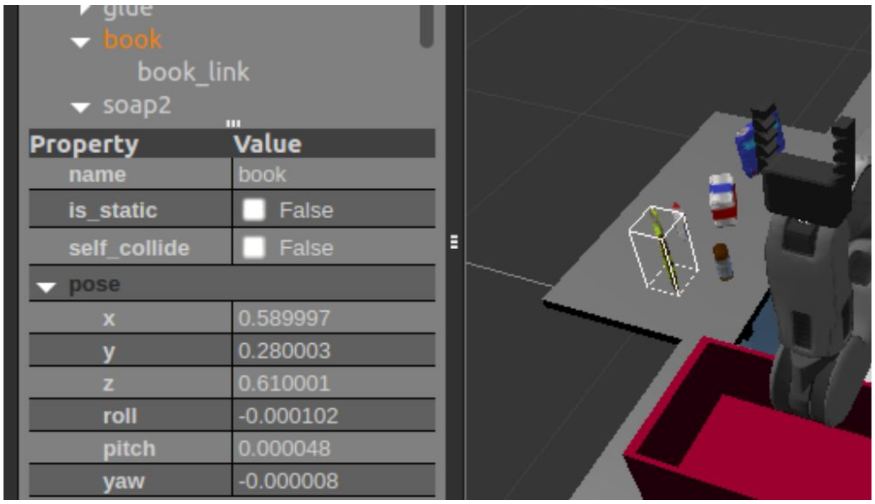
```
def voxel_grid_filter(cloud):
   # Create a VoxelGrid filter object for the point cloud
   vox = cloud.make_voxel_grid_filter()
   # Choose a voxel (leaf) size (in meters)
   LEAF_SIZE = 0.002
   # Set the voxel (or leaf) size
   vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
   cloud_filtered = vox.filter()
   return cloud_filtered
```

The image below shows objects after Voxel Grid Filter was applied.

PassThrough Filter

The PassThrough Filter works like a cropping tool, which allows you to crop any given 3D point cloud by specifying an axis with cut-off values along that axis. My region of interest was the table plane with objects on it.

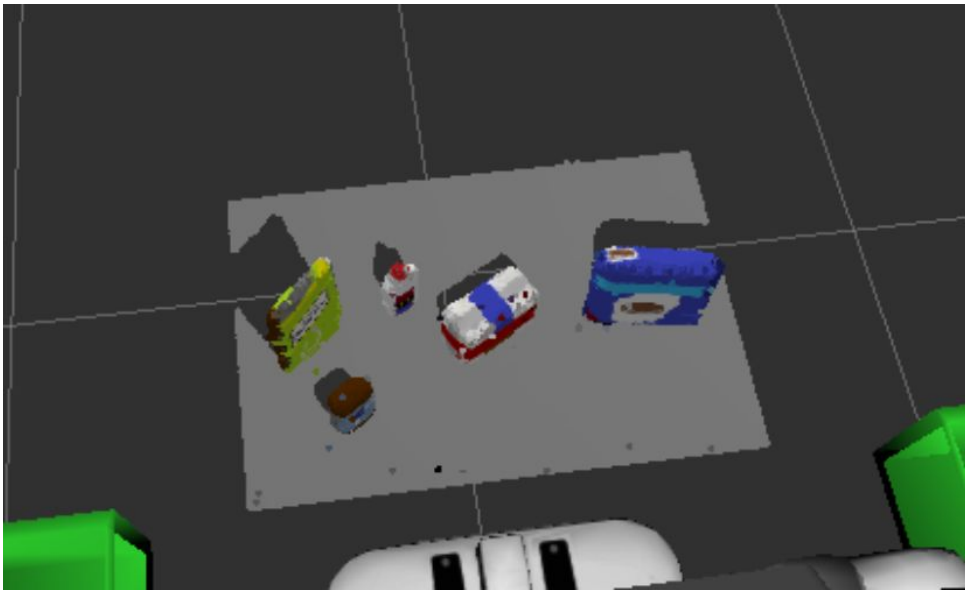
In the project it was not enough to filter z axis. Additionally, I had to filter y axis to get rid of the corners of the boxes captured by the camera of the robot. To get an idea on what values for min and max axis to use, I looked in Gazebo world properties (see the image below).



Next, is the function I used to apply PassThrough filter.

```
def pass_through_filter(cloud):
   # Create a PassThrough filter object.
   passthrough = cloud.make_passthrough_filter()

   # Assign axis and range to the passthrough filter object.
   filter_axis = 'z'
   passthrough.set_filter_field_name(filter_axis)
   axis_min = 0.6097
   axis_max = 0.9
   passthrough.set_filter_limits(axis_min, axis_max)

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

   passthrough = cloud_filtered.make_passthrough_filter()

   # Assign axis and range to the passthrough filter object.
   filter_axis = 'y'
   passthrough.set_filter_field_name(filter_axis)
   axis_min = -0.4
   axis_max = 0.4
   passthrough.set_filter_limits(axis_min, axis_max)

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

The image below shows filtered objects and table:

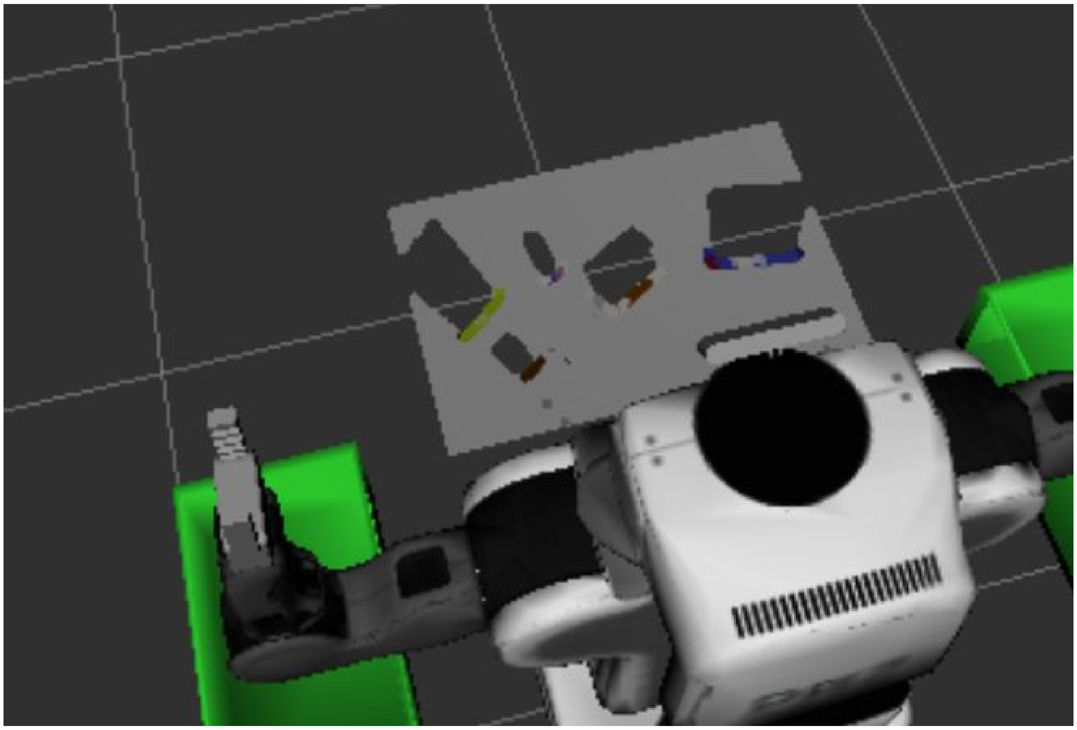


RANSAC Plane Segmentation

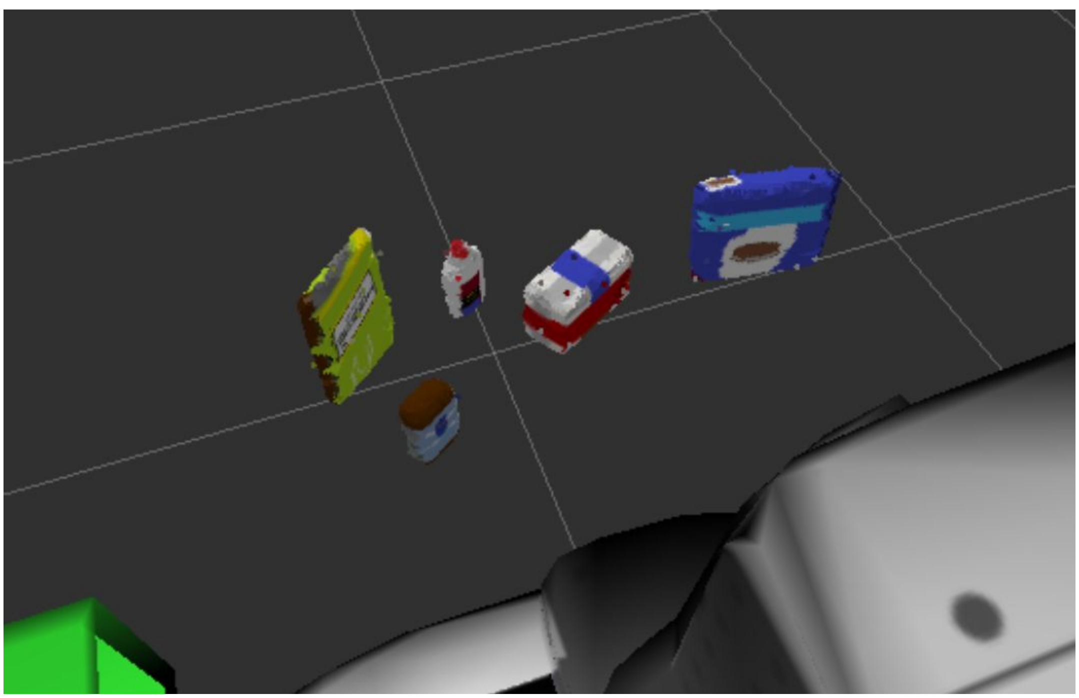
Next in the perception pipeline, I needed to remove the table itself from the scene. To do this I used a technique known as Random Sample Consensus or "RANSAC". It is an algorithm, that can be used to identify points in a dataset that belong to a particular model. The RANSAC algorithm assumes that all of the data in a dataset is composed of both inliers and outliers. Since the top of the table in the scene is the single most prominent plane, I used RANSAC to identify points that belong to the table (inliers) and filter them out.

```
def ransac_plane_segmentation(cloud):
   # Create the segmentation object
   seg = cloud.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 = 0.02
   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
   plane = cloud.extract(inliers, negative=False)

   # Extract outliers
   objects = cloud.extract(inliers, negative=True)

   return plane, objects
```

The image shows inliers. I tried different `max_distance` values, however, smaller values left too much noise around cropped objects.



Next, is the image showing the objects (outliers):



Clustering for Segmentation (Exercise 2)

DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. This algorithm is a nice alternative to k-means when you don't know how many clusters to expect in your data, but you do know something about how the points should be clustered in terms of density (distance between points in a cluster).

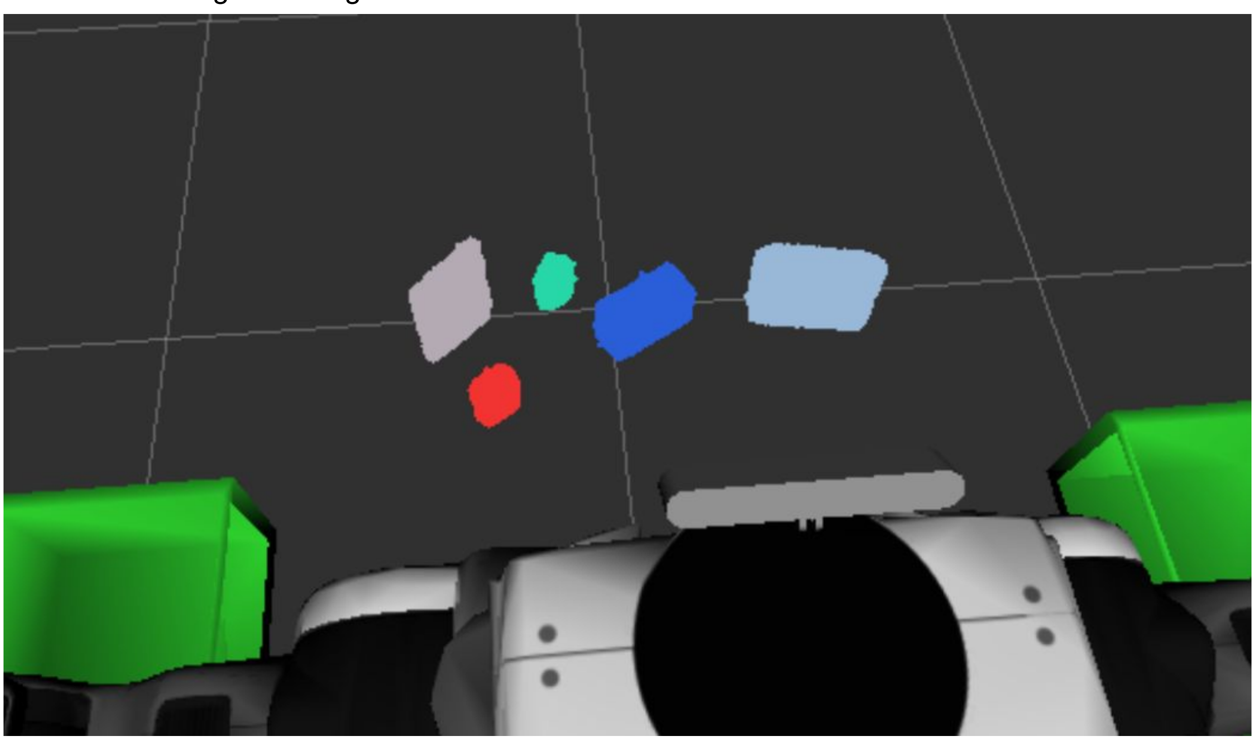
The DBSCAN algorithm creates clusters by grouping data points that are within some threshold distance from the nearest other point in the data.

The algorithm is sometimes also called “Euclidean Clustering”, because the decision of whether to place a point in a particular cluster is based upon the “Euclidean distance” between that point and other cluster members.

Euclidean clustering with PCL

For this project I used [PCL library]() function called `EuclideanClusterExtraction()` to perform a DBSCAN cluster search on my 3D point cloud. In order to perform Euclidean Clustering, first I had to construct a k-d tree from the `cloud_objects` point cloud. The k-d tree data structure is used in the Euclidean Clustering algorithm to decrease the computational burden of searching for neighboring points.

Below is the image showing clusters in Test 2 world:



The code added to the `pcl_callback()` function is shown below.

```
# PCL's Euclidean Clustering algorithm requires a point cloud with only spatial
information
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)
ec.set_ClusterTolerance(0.01)
ec.set_MinClusterSize(500)
ec.set_MaxClusterSize(50000)

# 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)
```

Object recognition (Exercise 3)

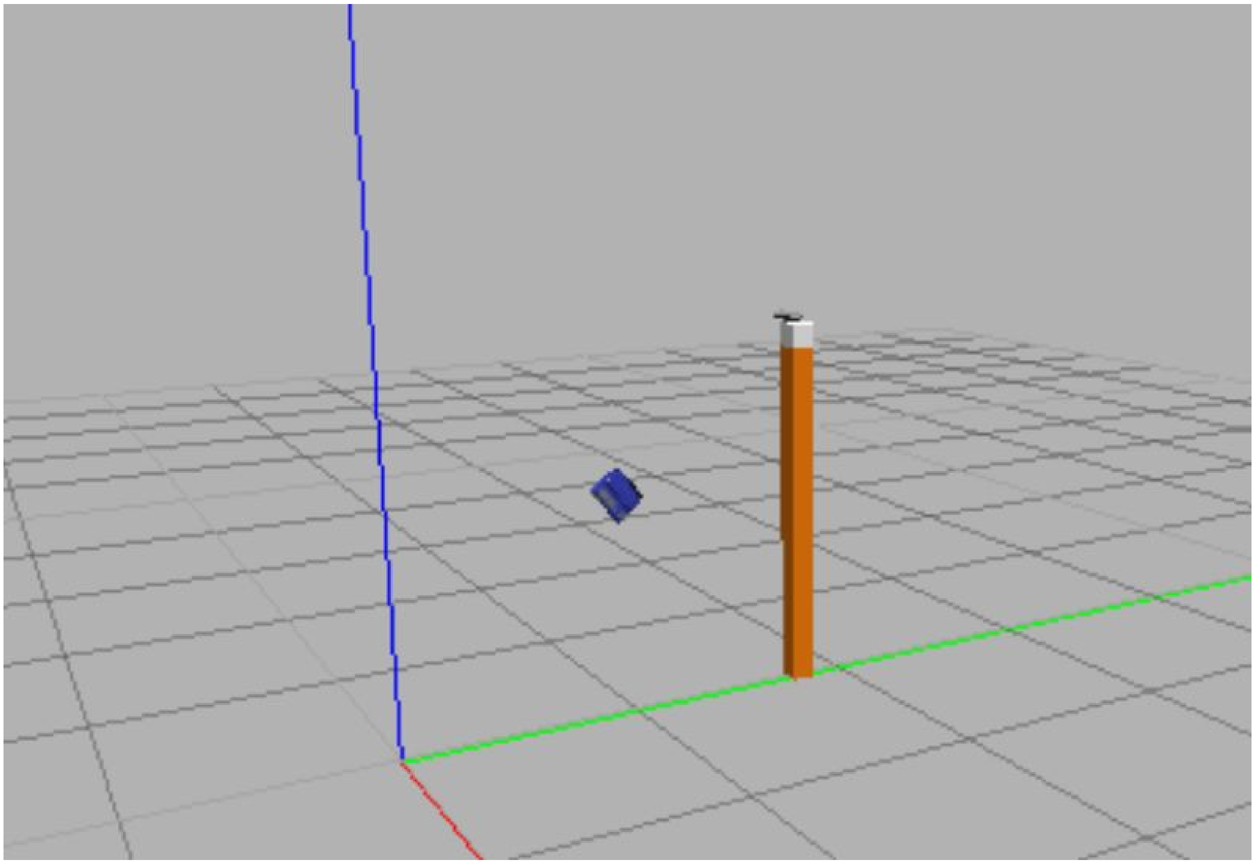
We would like the robot to be able to recognize objects in its surroundings just like people do. In any given image or 3D point cloud we might find a variety of objects of different shapes and sizes. In many robotics applications there will be a particular object we are looking at, it might be anywhere in the scene and even behind other objects. No matter what method is used to identify objects in the data set, it all comes down to the features that best describe the objects we are looking for. With the features, we can then train a classifier to recognize the object in the point cloud.

Feature Generation

I ran run the `capture_features.py` script to capture and save features for each of the environments. For the Test2 and Test 3 worlds I increased the loop range from 50 to 150, so I can capture more features and better identify the objects.

The model used for Test2.world was:

```
models = [
  'biscuits',
  'soap',
  'book',
  'soap2',
  'Glue']
```



Histograms

Next, to convert the color information into features that can be used in the classification, I built up the color values into a histogram. The color histogram of a known image can be compared with regions of a test image. Locations with similar color distributions will reveal a close match. Like this, objects that appear in slightly different poses and orientations will still be matched. Variations in image size can also be accommodated by normalizing the histograms. However, relying only on color values might result some false positives.
In the point cloud I have partial information on the 3D shapes of the object. I can compare the distributions of points with the ground truth. To do this I use a metric that captures shape and one of such metrics is the distribution of surface normals (normal - a unit vector perpendicular to that surface).

To achieve this I filled in `compute_color_histograms()` and `compute_normal_histograms()` functions in `/sensor_stick/src/sensor_stick/features.py` file. I used HSV color model to improve accuracy.
See the code bellow.

```
def compute_color_histograms(cloud, using_hsv=True):

   # 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
   ch_1_hist = np.histogram(channel_1_vals, bins=32, range=(0, 256))
   ch_2_hist = np.histogram(channel_2_vals, bins=32, range=(0, 256))
   ch_3_hist = np.histogram(channel_3_vals, bins=32, range=(0, 256))

   # Concatenate the histograms into a single feature vector
   hist_features = np.concatenate((ch_1_hist[0], ch_2_hist[0],
ch_3_hist[0])).astype(np.float64)
   # Normalize the result
   normed_features = hist_features / np.sum(hist_features)
   # Return the feature vector
   return normed_features

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 (just like with color)
   norm_x_hist = np.histogram(norm_x_vals, bins=32, range=(0, 256))
   norm_y_hist = np.histogram(norm_y_vals, bins=32, range=(0, 256))
   norm_z_hist = np.histogram(norm_z_vals, bins=32, range=(0, 256))
```

```python
    # Concatenate the histograms into a single feature vector
    hist_features = np.concatenate((norm_x_hist[0], norm_y_hist[0],
norm_z_hist[0])).astype(np.float64)
    # Normalize the result
    normed_features = hist_features / np.sum(hist_features)
    # Return the feature vector
    return normed_features
```

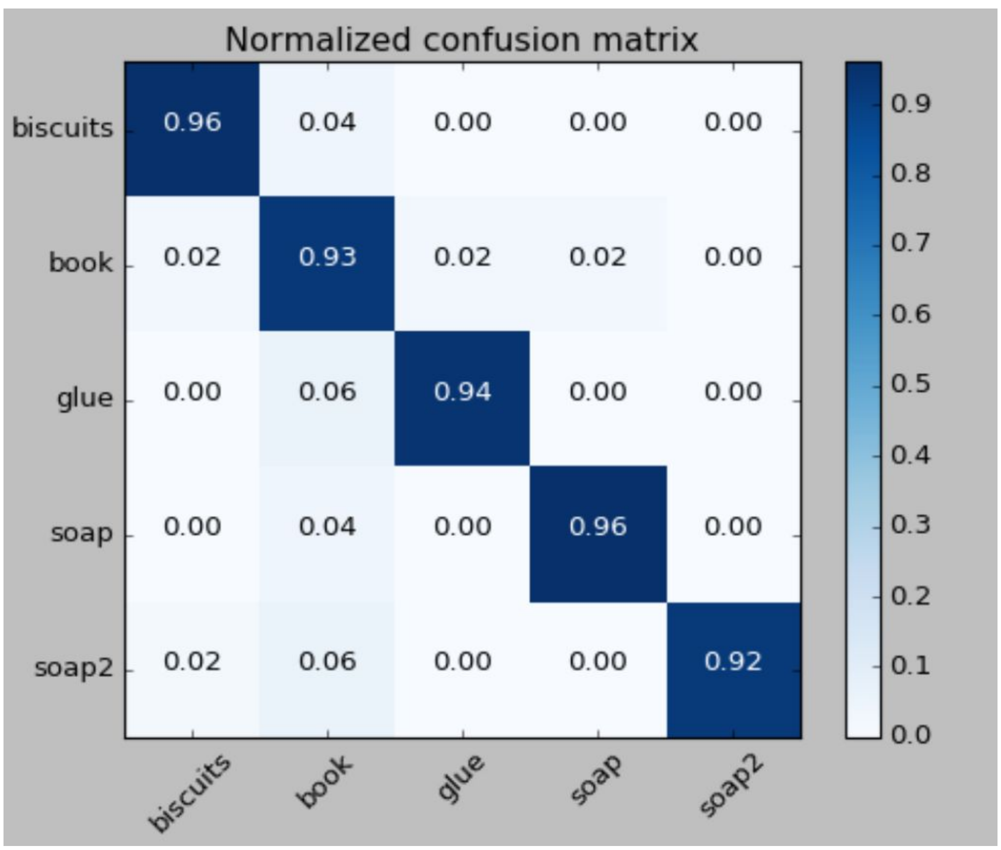
SVM training

Support Vector Machine or "SVM" is a supervised machine learning algorithm that allows you to characterize the parameter space of your dataset into discrete classes. SVMs work by applying an iterative method to a training dataset, where each item in the training set is characterized by a feature vector and a label.

The SVM has been trained using `train_svm.py`. Below is an example showing the results for The text output at the terminal regarding overall accuracy of Test2 world classifier:

```
Features in Training Set: 250
Invalid Features in Training set: 9
Scores: [ 0.93877551  0.91666667  0.97916667  0.95833333  0.91666667]
Accuracy: 0.94 (+/- 0.05)
accuracy score: 0.941908713693
```

Normalized confusion matrix showing relative accuracy of Test2 world classifier for the various objects:



Recognition

At this point I have a trained classifier ready to do object recognition. To see images with object recognition in different worlds refer to the next section.

The following code was added to `pcl_callback()` function:

```
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)

       #convert the cluster from pcl to ROS using helper function
       ros_cluster = pcl_to_ros(pcl_cluster)

       # Extract histogram features
       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, 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)
```

Pick and Place Setup

I performed object recognition for all three tabletop setups (test*.world) and got 100% accuracy in all the cases. Then I constructed the messages that would comprise a valid PickPlace request and saved them them to .yaml format.

The .yaml files can be found here:
https://github.com/daliavi/RoboND-Perception-Project/tree/master/pr2_robot/config

I tested the scenarios in this order Test 1, Test 3 and Test 2.

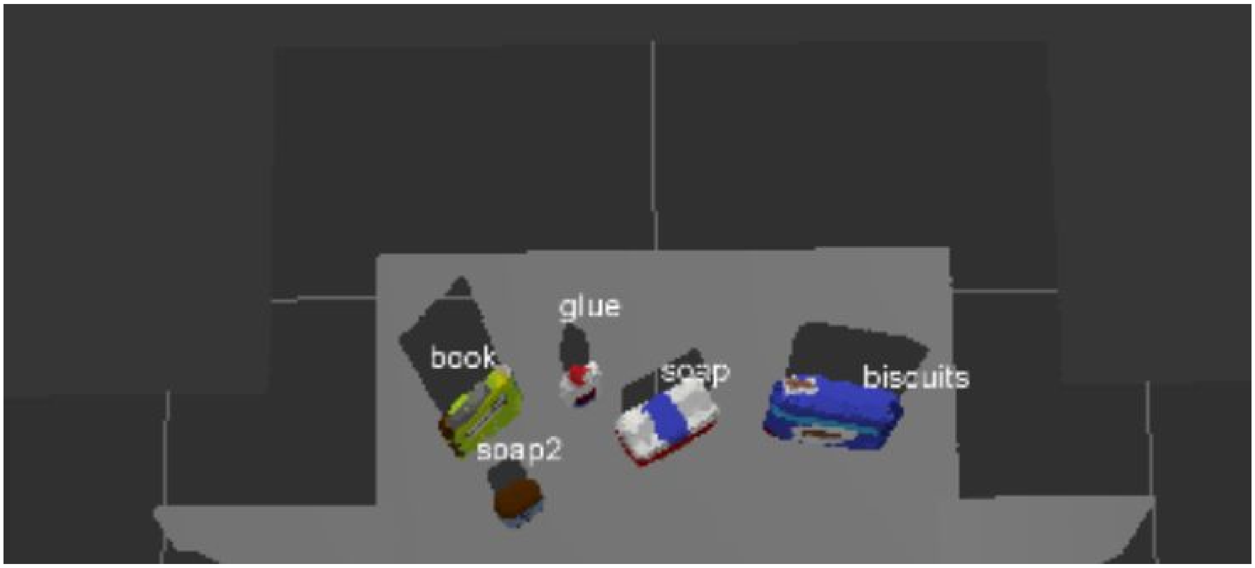
Test1.world recognition part went smoothly. I could recognize all 3 objects with the first run.

With the first test run of Test3.world I was able to recognize only 7 objects. I looked at the cluster view and I noticed that “glue” did not even have it’s cluster cloud. I went back to my code and decreased MinClusterSize value from 1000 to 500. With the second run, I had 8 objects, however 70% of the time “glue” was misclassified as “soap2”. Next, I went back to `sensor_stick` project and ran `capture_features.py` again with loop range(150), it was range(50) before. This time, this time the objects were recognized correctly as you can see in the image bellow.

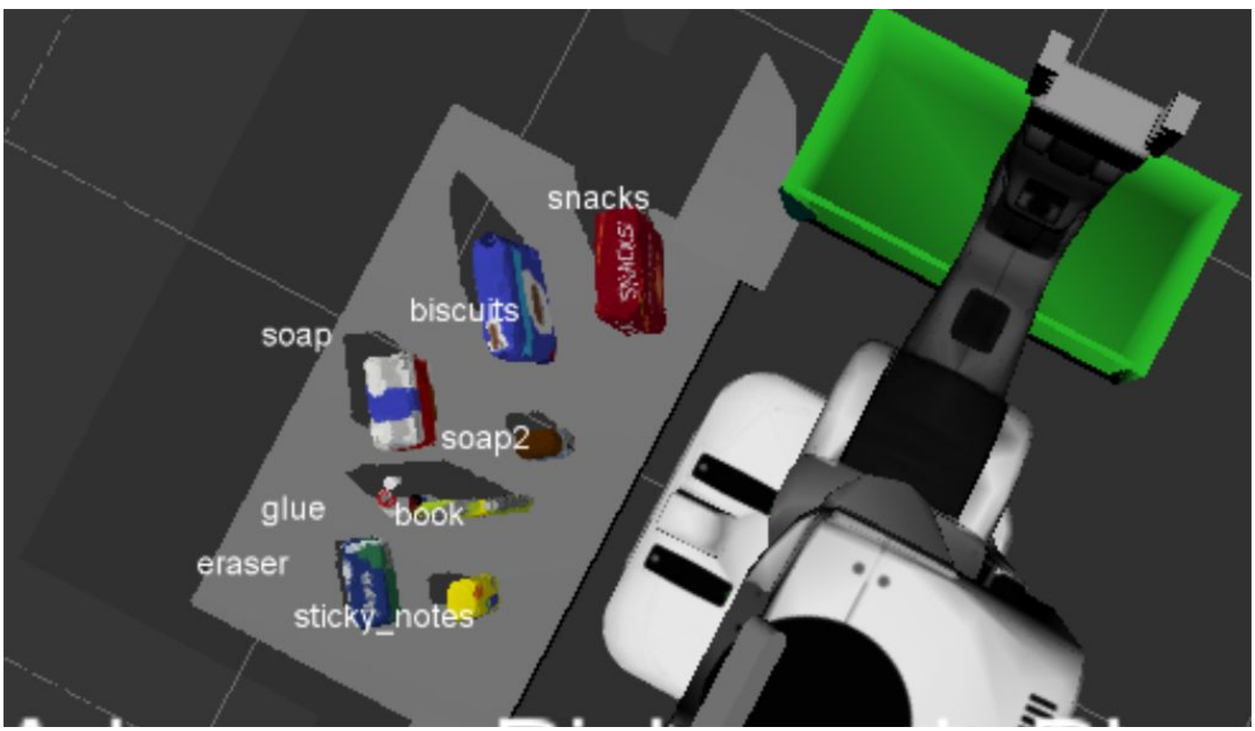
I ran Test2.world with the same setup as Test3.world and I did not have any issues. I managed to recognize all the objects with the first run.

Test1.world Recognition

Test2.world Recognition



Test3.world Recognition



Request parameters to the pick_place_server

Here is the function to load the parameters and request PickPlace service

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

    # Initialize variables
    test_scene_num = Int32()  # message type std_msgs/Int32
    pick_pose = Pose()  # message type geometry_msgs/Pose tuple of float64 .x, .y, .z
    place_pose = Pose()

    object_name = String()
    arm_name = String()  # "right" or "left"
    test_scene_num.data = TEST_SCENE
    dict_list = []  # the list of dictionaries will be used to create yaml files

    # Get/Read parameters
    object_list_param = rospy.get_param('/object_list')
    dropbox_param = rospy.get_param('/dropbox')

    # Parse parameters into dictionaries
    object_group_dict = {}
    for d in object_list_param:
        object_group_dict[d['name']] = d['group']

    group_position_dict = {}
    for d in dropbox_param:
        group_position_dict[d['group']] = d['position']


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

    # Loop through the detected object list
    for object in object_list:
        # checking if object is in the object param list
        if object.label in object_group_dict:
            print "Label", object.label
            object_name.data = str(object.label)
            # Get the PointCloud for a given object and obtain it's centroid
            points_arr = ros_to_pcl(object.cloud).to_array()
            centr = np.mean(points_arr, axis=0)[:3]
            # Create pick pose in ROS format
            #pose = [np.asscalar(centr[0]), np.asscalar(centr[1]), np.asscalar(centr[2])]
            pick_pose.position.x = np.asscalar(centr[0])
            pick_pose.position.y = np.asscalar(centr[1])
            pick_pose.position.z = np.asscalar(centr[2])
            print "Pick: ", pick_pose

            # Create 'place_pose' for the object
```

```
        pose = np.array(group_position_dict[object_group_dict[object.label]])
        place_pose.position.x = np.asscalar(pose[0])
        place_pose.position.y = np.asscalar(pose[1])
        place_pose.position.z = np.asscalar(pose[2])
        print "Place ", place_pose

        # Assign the arm to be used for pick_place
        if object_group_dict[object.label] == 'green':
            arm_name.data = 'right'
        else:
            arm_name.data = 'left'

        # Create a list of dictionaries (made with make_yaml_dict()) for later output to yaml format
        dict_list.append(make_yaml_dict(test_scene_num, arm_name, object_name, pick_pose, place_pose))


    # Output your request parameters into output yaml file
    send_to_yaml('output_%s.yaml' %TEST_SCENE, 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)

        # 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
```