RBE450X - VISION-BASED ROBOTIC MANIPULATION

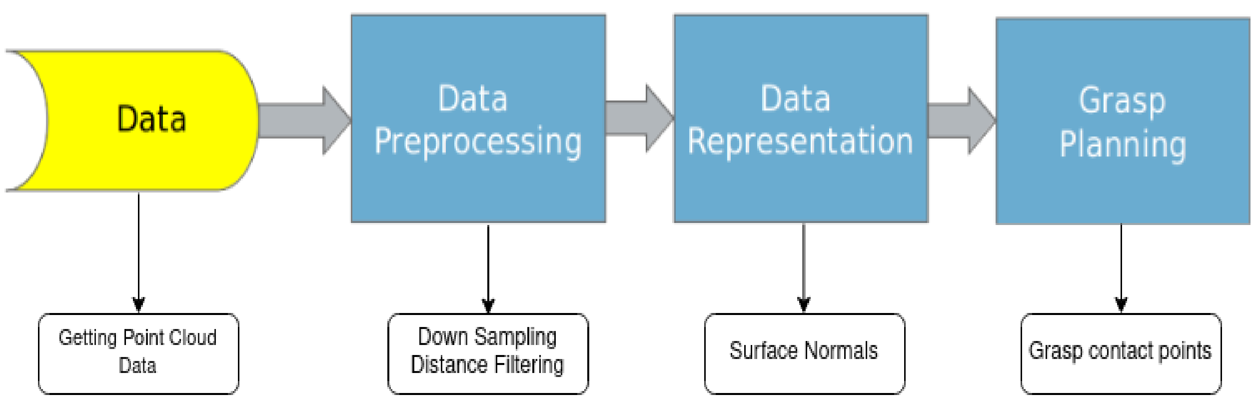
ACTIVE VISION FOR GRASPING

GROUP 4
Sri Lakshmi Hasitha Bachimanchi, Saurabh Kashid, Yuen Lam Leung,
Sai Ramana Kiran Pinnama Raju, Tript Sharma, Prarthana Sigedar

OBJECTIVE

Implementation of Active Vision by moving the camera to different locations, calculating the best grasp locations in every viewpoint, and finding the overall best grasp in the respective viewpoints. In this project, we follow the grasping pipeline which consists of Data Preprocessing with downsampling and segmentation, Data Representation with normal calculations, and Grasp Synthesis using force vector formulation. We then visualize the resulting Point Cloud object and the contact points of the best grasp in RViz.

FLOWCHART



APPROACH

1. SETTING UP THE WORLD AND ENVIRONMENT

In this step, the required world and environment are set up in the Gazebo simulator by spawning a table for the major plane, an object for grasping, a camera, and light. Furthermore, we had to add extra plugins in ROS2 to get the pose and orientation of our entity (camera).

2. ACQUIRING AND PREPROCESSING THE POINT CLOUD DATA

The segmentation node first acquires the Point Cloud data by subscribing to the topic `/realsense/points`. We then applied a downsampling VoxelGrid Filter with a leaf size of 0.8 cm. Downsampling is necessary because initially, our Point Cloud data had 50,000 points and this made our computation extremely slow and heavy, thereby we decided to downsample our data.

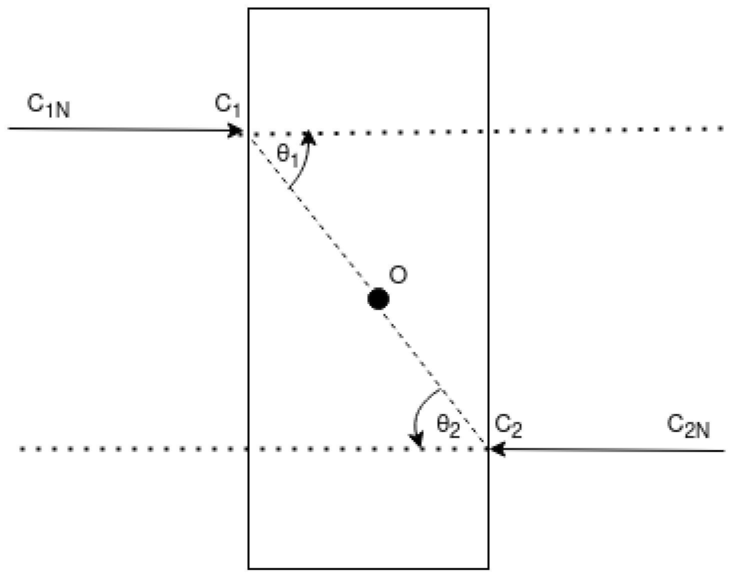
3. SEGMENTATION OF POINT CLOUD DATA FROM A PARTICULAR VIEW

After downsampling our point cloud data, we applied distance thresholding to eliminate the ground plane. Depending on the angle, it is possible that the ground plane is the major plane detected. Therefore, we set the distance threshold to 1 meter to filter out the ground plane before applying the RANSAC algorithm. Then we used the RANSAC algorithm on the filtered Point Cloud data to identify the major plane, in this case, the major plane will be the table plane as the ground plane has been filtered. Then, we utilized the inliers derived from the RANSAC algorithm to identify the table plane, and the outliers to identify the object surface plane. For the purpose of visualization, we published the Point Cloud data of the table plane to the topic `/tablePoints`, and the object surface plane to the topic `/objectPoints`.

4. FINDING THE NORMAL VECTORS

Once we have our segmented object, the normal vectors are found using the `Normal Estimation` class with a `KDTree` and a search radius of 0.5 cm. Since the point cloud is not aware of the full object, it does not know which direction construes outwards or inwards of the object. To make sure that estimated normals are pointed outward we use the centroid of the object as a heuristic. We estimated the centroid using `compute3DCentroid`, and passed it as a viewpoint into the `flipNormalTowardsViewpoint` function. This flipped all the normals towards the centroid ensuring that normals pointed inwards of the object.

5. FINDING CONTACT PAIRS



Where,

C_1 = Contact Point 1
C_2 = Contact Point 2
O = Centroid of the Point Cloud in Viewpoint
C_{1N} = Normal for Contact Point 1
C_{2N} = Normal for Contact Point 2
Θ_1 = Angle between C_1OC_2 and C_{1N}
Θ_2 = Angle between C_1OC_2 and C_{2N}

We computed a grasp quality metric that computes normals and enforces the idea, "Opposing contact points passing through the centroid are stable". This hypothesis is based on the fact that if the center of mass of an object is covered within the grasp then the grasp should be stable enough to prevent object drops.

Hence, this signifies that C_1, O, and C_2 must be collinear. Considering all points and normals as vectors, the condition for collinearity can be formulated using dot product for vectors given as:

$$cos^{-1}\left(\frac{C_1O \,.\, C_2O}{||C_1O||\,||C_2O||} \right) \;==\; 0$$

However, since we have downsampled the point cloud, it's not necessary that we will always obtain directly opposing contact points. Hence the constraint equation is updated to:

$$cos^{-1}\left(\frac{C_1O \cdot C_2O}{||C_1O||\,||C_2O||}\right) \in [-\ \phi, \phi]$$

where ϕ defines the range for the angle between the two vectors to check collinearity.

After obtaining directly opposing points, we need to compute the angles between the C_1OC_2 with C_{1N} and C_{2N} respectively. The dot product of the vectors was computed to get Θ_1 and Θ_2.

Θ_1 and Θ_2 are necessary for computing the grasp quality metric condition given by:

$$GQA_{12} = (\theta_1 + \theta_2)\ if\ (\theta_1 + \theta_2) \in [180^{\circ} \pm \lambda]$$
$$GQA = max(GQA,\ GQA_{12})$$

where GQA refers to Grasp Quality Angle i.e. the quality metric we are trying to calculate and GQA_{12} refers to GQA for the corresponding contact point pair.

The process is repeated for all the points pairs and their corresponding normals to get the best GQA. Contact point pairs with summed angles closest to 180 degrees are stored as the optimal contact point pairs for that specific viewpoint.

6. REPEAT STEPS 1-6 FOR THE NEXT LOCATION

After getting the grasp point from one angle we move to the next viewpoint and repeat the process of getting the contact point from a different location.

RESULTS

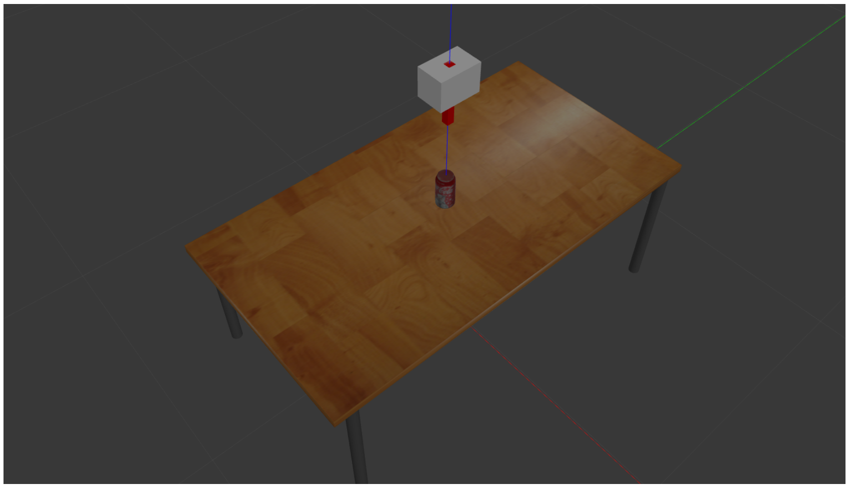


Fig 1: Gazebo Environment

To prepare the environment for the project, we generated a table, an object, a camera, and the light in Gazebo. The object we are using for this project is a coke can. We initialized the camera pointing downwards at the object, and we used a Gazebo or set entity to move and rotate the camera to a different viewpoint.

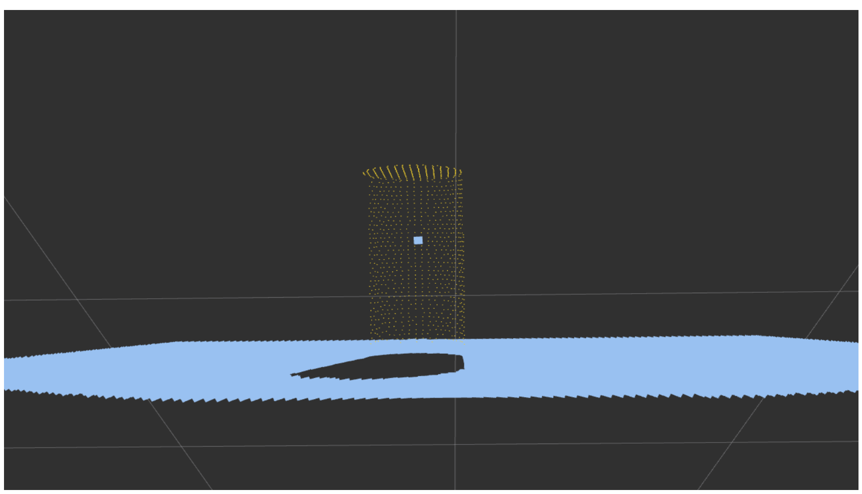
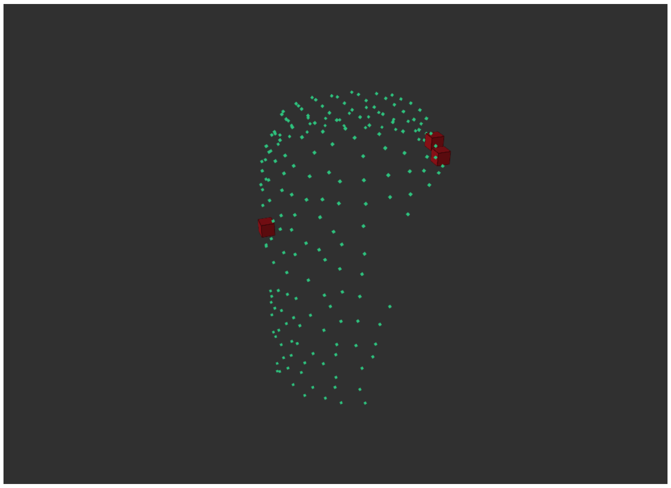
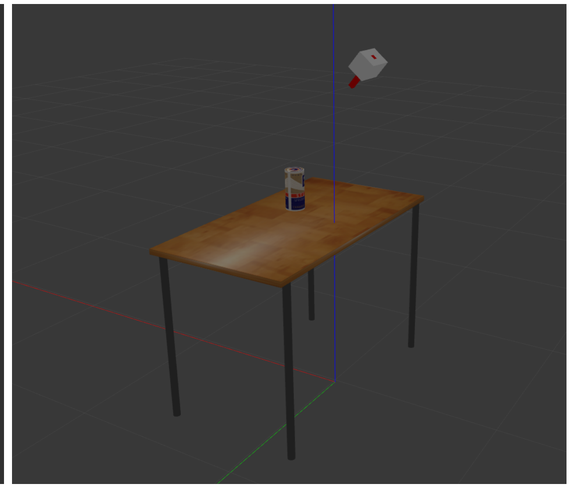


Fig 2: Segmented object with centroid

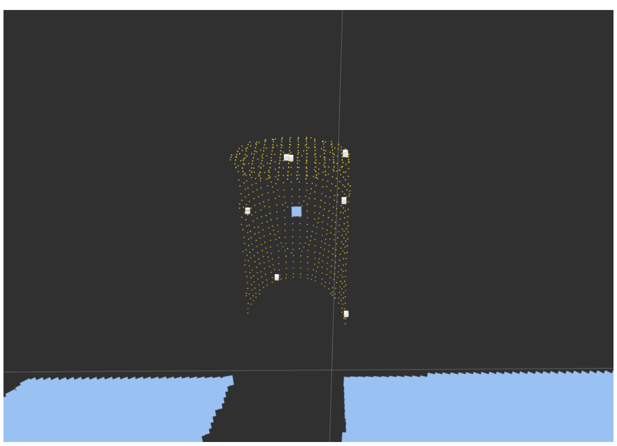
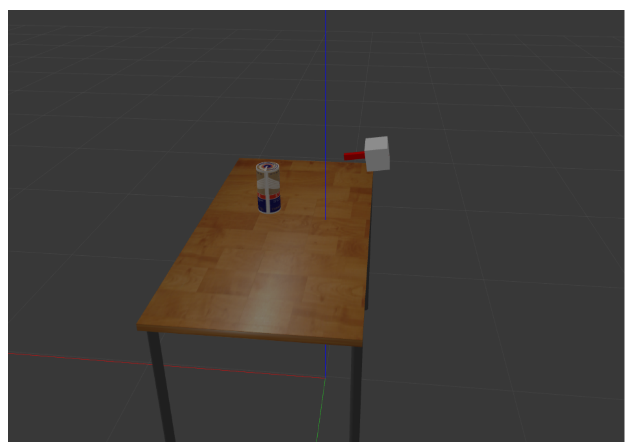
To preprocess the data for normal calculation, we segmented the object from the table plane. As shown in figure 2, the segmented object is represented in yellow and the table plane is represented in blue. The centroid is also calculated, which is also represented in blue.

(a) Contact points of the best grasp (b) Configuration of viewpoint 1

Fig 3: Segmented Point Cloud object and the result of best grasp in viewpoint 1

After data preprocessing, we used the filtered Point Cloud object to calculate the normals. As we will be using force vector calculation to calculate the best grasp, in which we are finding the collinear pair of contact points that crosses the centroid or the Center of Mass, we flipped all the normals towards the centroid. This allows us to be consistent in our force vector calculations using the dot products of the pair of normals. Brute force search is used to find the best grasp, and as seen in figure 2, the contact points of the best grasp are indicated with red cubes. In this case, there are two pairs of best grasp.

(a) Contact points of the best grasp (b) Configuration of viewpoint 2

Fig 4: Segmented Point Cloud object and the result of best grasp in viewpoint 2

After changing the viewpoint we went through the same process from acquiring and preprocessing the point cloud data to finding the contact points. The above image shows the resulting contact point giving the best grasp. At a lower angle, we are able to see more Point Cloud data on the side of the coke can, enabling us to formulate more good grasps.

SNAPSHOT OF THE CODE

```cpp
// Collinearty check function
bool isCollinear(const Eigen::Vector3f& vec1, const Eigen::Vector3f& vec2, double eps = 0.1) const
{
  // Normalize the functions before calculation dot product
  const auto& vec1_norm = vec1.normalized();
  const auto& vec2_norm = vec2.normalized();

  // derived dot product
  auto dot_product_val = vec1_norm.dot(vec2_norm);

  // making sure that dot product value is around -1 (angle as 180)
  //  since there can be numerical accuracies not giving us perfect -1
  if ((-1 - eps <= dot_product_val) && (dot_product_val <= -1 + eps))
  {
    return true;
  }
  return false;
}
```

```cpp
/* Function to calculate the best grasp contact pairs in the segmented point cloud */
std::vector<std::pair<Eigen::Vector3f, Eigen::Vector3f>>
   getBestGraspContactPair(const Eigen::Matrix3Xf& normals,
                           const Eigen::Matrix3Xf& contact_points,
                           const Eigen::Vector3f& centroid) const
 {
   // Initialize the containers to store the contact pairs
   std::vector<std::pair<Eigen::Vector3f,Eigen::Vector3f>> cp_pairs;
   pcl::PointCloud<pcl::PointXYZ>::Ptr grasp_point_cloud (new pcl::PointCloud<pcl::PointXYZ>);

   // ideal best grasp angle value
   double best_grasp_angle = 0;

   // define angle threshold
   float angle_threshold_degree = 10;
   float angle_threshold = angle_threshold_degree * (M_PI / 180);

   // Matrix of Vectors between contact points and centroid
   auto CX0 = contact_points.colwise() - centroid;

   /* Check against all the contact points iteratively */
   for (size_t i=0; i < normals.cols(); i++)
   {
     // 1st contact point that is considered
     const auto& C1 = contact_points(all, i);

     // 1st contact point normal
     const auto& C1N = normals(all, i);

     // vector between 1st contact point and centroid
     const auto& C10 = CX0(all, i);

     for (size_t j=0; j<normals.cols(); j++)
     {
       // exclude comparing between same contact points
       if (i==j)
       {
         continue;
       }

       // 2nd contact point that is considered
       const auto& C2 = contact_points(all, j);

       // vector between 2nd contact point and centroid
       const auto& C20 = CX0(all, j);

       // check if vectors between
       //      (1st contact point and centroid)
       //   and (2nd contact point and centroid)
       //   are collinear
       auto is_collinear = isCollinear(C10,C20);

       // if they are collinear check if they satisfy the necessary
       //   force vector grasp formulation
       if (is_collinear)
       {

         // 1st contact point normal
```
segmentation.cpp

```
        //  and (2nd contact point and centroid)
        //  are collinear
        auto is_collinear = isCollinear(C10,C20);

        // if they are collinear check if they satisfy the necessary
        //  force vector grasp formulation
        if (is_collinear)
        {

          // 1st contact point normal
          const auto& C2N = normals(all, j);

          // vector between contact points
          auto C1C2 = (C1-C2).normalized();

          // calculate angles between contact points and corresponding normals
          auto angle1 = acos(C1N.dot(C1C2));
          auto angle2 = acos(C2N.dot(C1C2));
          double grasp_angle = angle1 + angle2;

          // Check if corresponding grasp angle is falling within the threshold
          if(M_PI - angle_threshold < grasp_angle && grasp_angle < M_PI + angle_threshold)
          {
             // Check if the corresponding grasp angle is better
             // than previous candidate grasp angles
             if (grasp_angle >= best_grasp_angle)
             {
                grasp_point_cloud->push_back(pcl::PointXYZ(C1(0),C1(1),C1(2)));
                grasp_point_cloud->push_back(pcl::PointXYZ(C2(0),C2(1),C2(2)));
                cp_pairs.push_back({C1, C2});
                best_grasp_angle = grasp_angle;
             }
          }
        }
      }
    }

    auto output_grasp_points = new sensor_msgs::msg::PointCloud2;
    pcl::PCLPointCloud2::Ptr cloud_grasp_points(new pcl::PCLPointCloud2);
    pcl::toPCLPointCloud2(*grasp_point_cloud,*cloud_grasp_points);
    pcl_conversions::fromPCL(*cloud_grasp_points, *output_grasp_points);
    output_grasp_points->header.frame_id = "camera_link";
    grasp_points_pub_->publish(*output_grasp_points);

    return cp_pairs;
  }
```

```cpp
void topic_callback(const sensor_msgs::msg::PointCloud2::SharedPtr msg)
{
  // Convert sensor_msg::PointCloud2 to pcl::PCLPointCloud2
  pcl::PCLPointCloud2::Ptr cloudPtr(new pcl::PCLPointCloud2); // container for pcl::PCLPointCloud
  pcl_conversions::toPCL(*msg, *cloudPtr); // convert to PCLPointCloud2 data type

  // 1. Downsample
  pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
  sor.setInputCloud (cloudPtr);
  sor.setLeafSize (0.008f, 0.008f, 0.008f);
  sor.filter (*cloudPtr);

  // Convert pcl::PCLPointCloud2 to PointXYZ data type
  pcl::PointCloud<pcl::PointXYZ>::Ptr XYZcloudPtr(new pcl::PointCloud<pcl::PointXYZ>);
  pcl::fromPCLPointCloud2(*cloudPtr,*XYZcloudPtr);

  // 2. Distance Thresholding: Filter out points that are too far away, e.g. the floor
  auto plength = XYZcloudPtr->size();   // Size of the point cloud
  pcl::PointIndices::Ptr farpoints(new pcl::PointIndices());  // Container for the indices
  for (int p = 0; p < plength; p++)
  {
      // Calculate the distance from the origin/camera
      float distance = (XYZcloudPtr->points[p].x * XYZcloudPtr->points[p].x) +
                       (XYZcloudPtr->points[p].y * XYZcloudPtr->points[p].y) +
                       (XYZcloudPtr->points[p].z * XYZcloudPtr->points[p].z);

      if (distance > 1) // Threshold = 1
      {
        farpoints->indices.push_back(p);    // Store the points that should be filtered out
      }
  }

  // 3. Extract the filtered point cloud
  pcl::ExtractIndices<pcl::PointXYZ> extract;
  extract.setInputCloud(XYZcloudPtr);
  extract.setIndices(farpoints);          // Filter out the far points
  extract.setNegative(true);
  extract.filter(*XYZcloudPtr);

  // 4. RANSAC; Plane model segmentation from pcl
  pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
  pcl::PointIndices::Ptr inliers (new pcl::PointIndices);

  // 5. Create the segmentation object
  pcl::SACSegmentation<pcl::PointXYZ> seg;
  seg.setOptimizeCoefficients (true);
  seg.setModelType (pcl::SACMODEL_PLANE);
  seg.setMethodType (pcl::SAC_RANSAC);
  seg.setDistanceThreshold (0.01);
  seg.setInputCloud (XYZcloudPtr);
  seg.segment (*inliers, *coefficients);

  if (inliers->indices.size () == 0)
  {
    PCL_ERROR ("Could not estimate a planar model for the given dataset.\n");
  }

  // 6. Extract the inliers
```

segmentation.cpp

```cpp
if (inliers->indices.size () == 0)
{
  PCL_ERROR ("Could not estimate a planar model for the given dataset.\n");
}

// 6. Extract the inliers
pcl::PointCloud<pcl::PointXYZ>::Ptr XYZcloud_filtered(new pcl::PointCloud<pcl::PointXYZ>); // container for pcl::PointXYZ
pcl::PointCloud<pcl::PointXYZ>::Ptr XYZcloud_filtered_table(new pcl::PointCloud<pcl::PointXYZ>); // container for pcl::PointXYZ
extract.setInputCloud (XYZcloudPtr);
extract.setIndices (inliers);
extract.setNegative (false);  // false -> major plane, true -> object
extract.filter (*XYZcloud_filtered_table);

extract.setInputCloud (XYZcloudPtr);
extract.setIndices (inliers);
extract.setNegative (true);  // false -> major plane, true -> object
extract.filter (*XYZcloud_filtered);

// 7. NORMAL ESTIMATION
// Create the normal estimation class, and pass the input dataset to it
pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
ne.setInputCloud (XYZcloud_filtered);

// Create an empty KDTree representation, and pass it to the normal estimation object.
// Its content will be filled inside the object, based on the given input dataset (as no other search surface is given).
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ> ());
ne.setSearchMethod (tree);

// Output datasets
pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new pcl::PointCloud<pcl::Normal>);

// Use all neighbors in a sphere of radius 3cm
ne.setRadiusSearch (0.005);
ne.useSensorOriginAsViewPoint();

// 7.1 Compute the features
ne.compute (*cloud_normals);
RCLCPP_INFO_STREAM(this->get_logger(), "# of normals: " << cloud_normals->size ());

// 8. CENTROID
// 16-bytes aligned placeholder for the XYZ centroid of a surface patch
Eigen::Vector4f xyz_centroid;

// 9. Estimate the XYZ centroid
pcl::compute3DCentroid (*XYZcloud_filtered, xyz_centroid);
auto centroid_point = new sensor_msgs::msg::PointCloud2;
pcl::PCLPointCloud2::Ptr centroid_cloud_point(new pcl::PCLPointCloud2);
pcl::PointCloud<pcl::PointXYZ> centroid_point_cloud;
centroid_point_cloud.push_back(pcl::PointXYZ(xyz_centroid(0),xyz_centroid(1),xyz_centroid(2)));
pcl::toPCLPointCloud2(centroid_point_cloud,*centroid_cloud_point);
pcl_conversions::fromPCL(*centroid_cloud_point, *centroid_point);
centroid_pub_->publish(*centroid_point);


 // Table
XYZcloud_filtered_table->push_back(pcl::PointXYZ(xyz_centroid[0], xyz_centroid[1], xyz_centroid[2]));
auto output_table = new sensor_msgs::msg::PointCloud2;                  // TABLE: container for sensor_msgs::msg::PointCloud2
pcl::PCLPointCloud2::Ptr cloud_filtered_table(new pcl::PCLPointCloud2); // TABLE: container for pcl::PCLPointCloud2
```

segmentation.cpp

```
    pcl_conversions::fromPCL(*centroid_cloud_point, *centroid_point);
    centroid_pub_->publish(*centroid_point);


     // Table
    XYZcloud_filtered_table->push_back(pcl::PointXYZ(xyz_centroid[0], xyz_centroid[1], xyz_centroid[2]));
    auto output_table = new sensor_msgs::msg::PointCloud2;                  // TABLE: container for sensor_msgs::msg::PointCloud2
    pcl::PCLPointCloud2::Ptr cloud_filtered_table(new pcl::PCLPointCloud2); // TABLE: container for pcl::PCLPointCloud2
    pcl::toPCLPointCloud2(*XYZcloud_filtered_table,*cloud_filtered_table);  // TABLE: convert pcl::PointXYZ to pcl::PCLPointCloud2
    pcl_conversions::fromPCL(*cloud_filtered_table, *output_table);         // TABLE: convert PCLPointCloud2 to sensor_msgs::msg::PointCloud2

    // Object
    auto output = new sensor_msgs::msg::PointCloud2;                        // OBJ: container for sensor_msgs::msg::PointCloud2
    pcl::PCLPointCloud2::Ptr cloud_filtered(new pcl::PCLPointCloud2);       // OBJ: container for pcl::PCLPointCloud2
    pcl::toPCLPointCloud2(*XYZcloud_filtered,*cloud_filtered);              // OBJ: convert pcl::PointXYZ to pcl::PCLPointCloud2
    pcl_conversions::fromPCL(*cloud_filtered, *output);                     // OBJ: convert PCLPointCloud2 to sensor_msgs::msg::PointCloud2

    segmented_pub_->publish(*output);                                        // publish OBJECT plane to /objectPoints
    table_pub_->publish(*output_table);                                      // publish TABLE plane to /tablePoints

    pcl::PointXYZ centroidXYZ(xyz_centroid[0], xyz_centroid[1], xyz_centroid[2]);

    // 10. FLIPPING NORMALS ACCORIDNG TO CENTROID
    Eigen::Matrix3Xf normal_vector_matrix(3,cloud_normals->size());
    Eigen::Matrix3Xf point_cloud(3,cloud_normals->size());
    for(size_t i = 0; i < cloud_normals->size(); i++)
    {
      Eigen::Vector3f normal = cloud_normals->at(i).getNormalVector4fMap().head(3);
      Eigen::Vector3f normal_dup = cloud_normals->at(i).getNormalVector4fMap().head(3);

      //pcl::flipNormalTowardsViewpoint(centroidXYZ, 0, 0, 0, normal);
      pcl::flipNormalTowardsViewpoint(XYZcloud_filtered->at(i), xyz_centroid[0], xyz_centroid[1], xyz_centroid[2], normal);
      normal_vector_matrix(0,i) = normal[0];
      normal_vector_matrix(1,i) = normal[1];
      normal_vector_matrix(2,i) = normal[2];

      //const auto& pointMatrix = XYZcloud_filtered->at(i);
      point_cloud(0,i) = XYZcloud_filtered->points[i].x;
      point_cloud(1,i) = XYZcloud_filtered->points[i].y;
      point_cloud(2,i) = XYZcloud_filtered->points[i].z;
    }

    const auto& data = getBestGraspContactPair(normal_vector_matrix, point_cloud,xyz_centroid.head(3));
    RCLCPP_INFO_STREAM(this->get_logger(), "Size of data: " << data.size());

  };
```