# Perception and Mapping –

# Assignment 3: "Collision Avoidance using 3D LiDAR"

Group: José Pedro dos Santos Rodrigues (201806386)

Marina Rodrigues Brilhante (201806677)

## Answers to the queries

*P1) Develop a program (a node called detector) in C++.*

For this part, we started with creating a subscriber that receives the Point Clouds contained in the provided rosbag. To detect the closest object to the vehicle and compute its centroid, we followed the following steps:

1. *Filtering*
   In order to increase the processing speed and reduce the points in the Cloud that are not important (the background, mostly) we first downsampled the points using Voxel Grid. This resulted in fewer number of points in the Point Cloud and in a voxelized grid, where each voxel is approximated with its centroid.

```
pcl::VoxelGrid<pcl::PointXYZ> vg;
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered(new
pcl::PointCloud<pcl::PointXYZ>);
vg.setInputCloud(cloud);
vg.setLeafSize(0.15f, 0.15f, 0.15f);
vg.filter(*cloud_filtered);
```

2. *Clustering*
   After, we used clustering to help us identify the object in the scene, in this case, the boat. For this, we apply Euclidean Clustering. To find and segment the individual object we use a Kd-tree structure for finding the nearest neighbourhood. First, a point is chosen from the Cloud and all the points that are within the specified distance to this point are identified. Do this for every neighbour until no more points satisfy the distance criteria. All the identified points form a cluster. The algorithm terminates when all the points have been processed.

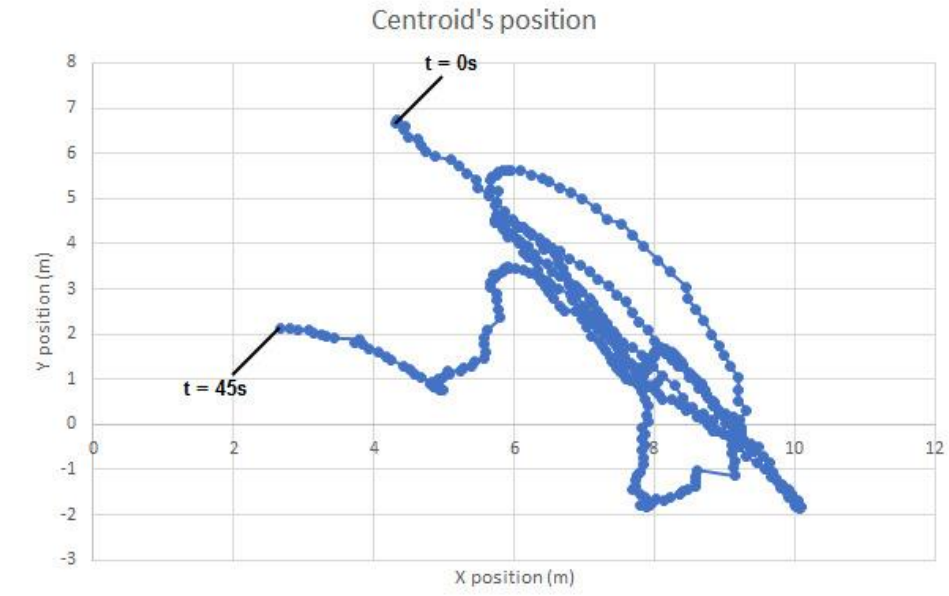
```
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new
pcl::search::KdTree<pcl::PointXYZ>);
tree->setInputCloud(cloud_filtered);

std::vector<pcl::PointIndices> cluster_indices;
pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
ec.setClusterTolerance(1);
```

```
ec.setMinClusterSize(100);
ec.setMaxClusterSize(25000);
ec.setSearchMethod(tree);
ec.setInputCloud(cloud_filtered);
ec.extract(cluster_indices);
```

3. *Bounding Box*
   For each cluster, a bounding box is fitted. In this, we used the *pcl::MomentOfInertiaEstimation* class. This class allows the extraction of the oriented bounding box of the cloud.
   As requested a video was made showing the evolution of the bounding box with the Point Cloud. We noticed a delay with the bounding box, this delay is due to the low computational power of the Virtual Machine, only working when added more RAM.

4. *Plotting the object's centroid in the xy chart*
   For the plot, we used Excel and the values of the closest centroid.




*P2) Develop a program (a node called risk_assessor) in C++.*

In part two, we started creating a subscriber that receives information about the IMU and the odometry, and the obstacle Point Cloud defined in the previous part. For this, it was assumed that the segmented object is static. To calculate the Time to Collison (TTC) we followed these steps:

1. *Estimating angular and linear velocity, position and acceleration in the inertial frame*
   For this, we first converted the values of orientation, provided by the IMU, into a 3x3 Rotation Matrix. With this matrix, we estimated the values of angular

velocity and acceleration. For the linear velocity and the position, we used the values provided by the Odometry.

```
tf::Matrix3x3 rot(tf::Quaternion(orientation.x, orientation.y, orientation.z,
orientation.w));
tf::Vector3 angular_velocity_inertial = rot * tf::Vector3(angular_velocity.x,
angular_velocity.y, angular_velocity.z);
tf::Vector3 position_inertial(position.x, position.y, position.z);
tf::Vector3 velocity_inertial(velocity.x, velocity.y, velocity.z);
tf::Vector3 acceleration_inertial = rot *
tf::Vector3(linear_acceleration.x,linear_acceleration.y,
linear_acceleration.z) - angular_velocity_inertial.cross(velocity_inertial);
```

2. *Calculating the Time to Collision (TTC)*

   The TTC is calculated by dividing the relative position by the relative velocity (both computed in step 2).

   To this, we only considered a possible collision when the object is in front of the vehicle, otherwise is considered TTC = 0 seconds.

```
float ang1 = atan(marker.points.at(0).y/marker.points.at(0).x);
float ang2 = atan(marker.points.at(0).y/marker.points.at(10).x);
float ang3 = atan(marker.points.at(10).y/marker.points.at(0).x);
float ang4 = atan(marker.points.at(10).y/marker.points.at(10).x);

float ang_max = std::max({ang1,ang2,ang3,ang4});
float ang_min = std::min({ang1,ang2,ang3,ang4});

if (vel_ang <= ang_max && vel_ang >= ang_min) {
  TTC = (tf::Vector3(center.x,center.y,center.z)).length() /
tf::Vector3(velocity.x,velocity.y,0).length();
}
  else {
    //ROS_INFO("No collision");
    TTC = 0;
  }
```

3. *Plotting the computed TTC over time*

   Here, we also used Excel to create the plot and used the values of TTC in seconds.