Worksheet 6 – PCL 3D Processing

This script covers the most basic tools of 3-D processing within the ROS environment. This usually means dealing with dedicated message types enclosing range sensory information and processing them through the Point Cloud Library (PCL).

1. 3-D Acquisition

There are a lot of sensors capable of acquiring range information. There are passive sensors like stereo camera pairs and active sensors like lasers, sonars, time-of-flight cameras or structured-light scanners, just to state the most common.

These devices' integration with ROS through dedicated drivers have the ultimate purpose of having a ROS process publish each sensed measurement under a certain topic message type. The most commonly used message types can be found under the *sensor_msgs* package (http://wiki.ros.org/sensor_msgs). From all these sensor messages, the ones more relevant to transmit 3-D information are *sensor_msgs/Range.msg*, *sensor_msgs/LaserScan.msg* and *sensor_msgs/PointCloud2.msg*.

The first is used to transmit a single range measurement, normally from a device with only one active range sensor. It encloses such information as the span of valid ranges (*range_min* and *range_max*), the type of emitted radiation (ultrasound or infrared) and, among others, the actual perceived range (*range*).

The second is used to hold a whole scan from a 2-D laser range-finder. On this type of message the planar scan has its range measurements equally distributed (*angle_increment*) in a determined window (*angle_min*, *angle_max*) through an array of *ranges* and *intensities*.

The *PointCloud2* message, as its name suggests, takes clouds of points, or range readings, in the form of their Cartesian coordinates, while the other two message types have them in the polar coordinate system. It can be properly set to have *width* and *height* and thus have all points mapped in a serialized 2-D array (*data*).

2. The ROS Bagfile and what's to come…

The ROS Bagfile issued with this worksheet was recorded during an experiment where a 2-D laser scanner, mounted on a platform, was made to tilt by a dedicated servo motor (see Figure 1). The whole scheme is, in its turn, mounted on a tripod, coupled with a camera to, among other uses, provide the necessary offline visual feedback of what is being perceived by the tilting laser.

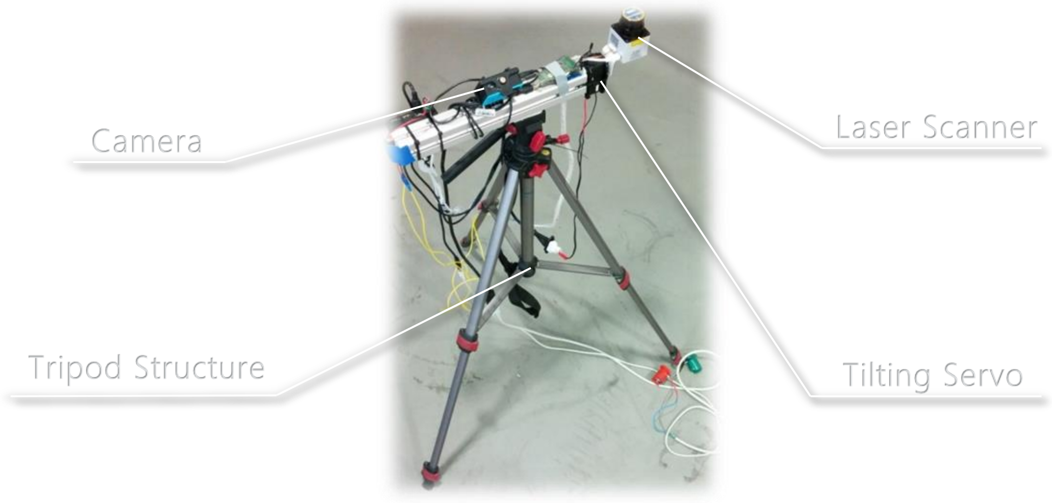


Figure 1 – Tripod Sensor Mount.

The tilting mechanism enables the whole 3-D space to be swept as the laser moves cyclically up and down.

In this particular exercise, the tripod setup was set to gather the 3-D occupancy of a shelf containing some eurocontainer type boxes (see Figure 2). The goal is then to process the incoming 2-D laser readings and, as the tilting occurs, reconstruct the 3-D shapes of the perceived objects, register them in a concise way, and, finally, assess the information to separate the boxes through a simple segmentation process.

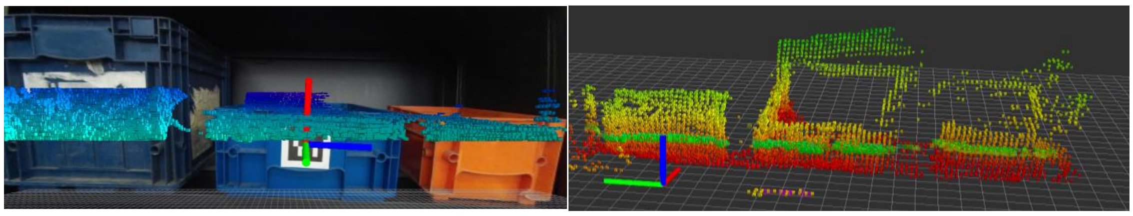


Figure 2 – Main topics included in the provided ROS Bagfile.

For now, play the Bagfile, setting the *use_sim_time* param *true* for the bagfile node to act as the clock server (http://wiki.ros.org/Clock):

```
$ rosparam set use_sim_time true
$ rosbag play --clock lesson6_bag_pcl.bag
```

and use the *rostopic list* command to see which topics are available.

```
$ rostopic list
```

```
/camera_image
/clock
/rosout
/rosout_agg
/scan
/tilt_servo_angle
```

The */scan* topic has *sensor_msgs::LaserScan* messages, with ranges spanning 240º, equally spaced by 1/4º, published at 10 Hz. The /tilt_angle is a *std_msgs::Float64* message, implicitly published by the servo motor (20 Hz), indicating the angle (in radians) the laser is rotated. Also a *sensor_msgs::Image* message topic, */camera_image*, is available to let you see, live, the space before the tripod.

Before we can interpret the 3-D occupancy information leading to the boxes's segmentation, through more complex PCL library functionalities, a few steps has to be taken:

1) Every *sensor_msgs::LaserScan* message has to be transported to the *sensor_msgs::PointCloud2* format, and then transformed according to the incoming rotation angle, */tilt_angle*.
2) The clouds of points should be coherently integrated both in time and space, onto a concise structure, like a voxel-grid or octree.

Only then, a *PointCloud2* concerning the whole perceived occupancy, i.e., the boxes and the shelves, can be filtered, matched to a model, and clustered into 2 main groups, thus pinpointing both boxes' positions.

3. Managing the Laser's Scan Readings

On this first step we will start by just transferring the hit measures from its original polar coordinates, within the *LaserScan* message, to the *PointCloud2*'s Cartesian setup. The ROS package *laser_geometry* already gathers a multitude of tools to deal with such conversions.

Let's begin by creating the package enclosing the ROS nodes for, not only this step, but also for all that will follow. The package will depend necessarily upon the *sensor_msgs* package, but also on *laser_geometry*.

```
$ cd ~/trsa_ws/src
$ catkin_create_pkg lesson6_package_pcl roscpp rospy sensor_msgs laser_geometry
```

Next create the source code file, ***laser_2_cloud.cpp***, for this step's conversions. There, you'll add the necessary include entries:

```
#include "ros/ros.h"
#include "sensor_msgs/LaserScan.h"
#include "sensor_msgs/PointCloud.h"
#include "laser_geometry/laser_geometry.h"
```

Then, create a class named ***LaserScan_2_PointCloud*** with the following members:

```
  ros::NodeHandle n_;
  laser_geometry::LaserProjection projector_;
  ros::Subscriber scan_sub_;
  ros::Publisher cloud_pub_;
```

The first refers to the *NodeHandle* belonging to the node which the class will be instantiated to, the second is the *LaserProjection* object dealing with the coordinate system conversion, and the last two are the associated with the *LaserScan* topic subscriber and *PointCloud2* message publisher, respectively.

The constructor should initialize these two members, ***scan_pub_*** and ***cloud_pub_***, as such:

```
LaserScan_2_PointCloud(ros::NodeHandle n) :
n_(n)
{
  scan_sub_ = n_.subscribe("/laser_scan", 1000, &LaserScan_2_PointCloud::scanCallback, this);
  cloud_pub_ = n_.advertise<sensor_msgs::PointCloud2>("/laser_cloud",1);
}
```

Thus assigning topic names and message buffers' sizes to both of them, as well as setting the subscriber's callback function to be the last remaining class's member, ***scanCallback***:

```
void scanCallback (const sensor_msgs::LaserScan::ConstPtr& scan_in)
{
  sensor_msgs::PointCloud2 cloud;
  projector_.projectLaser(*scan_in, cloud);

  cloud.header.stamp = ros::Time::now();
  cloud.header.frame_id = scan_in->header.frame_id;
  cloud_pub_.publish(cloud);
}
```

A *PointCloud2* object, ***cloud***, is declared to receive, through the *laser_geometry::LaserProjection* object's, ***projector_***, *projectLaser* call, the set $\{x, y, z\}^T$ points referring to the incoming *LaserScan* message, ***scan_in***, range readings, removing a few invalid (out of bound) ones along the way.

Note here, that usually this type of conversion is undertaken in a more careful manner. Normally, an evaluation of the transform backbone supporting the frame to which the *LaserScan* is referenced takes place. Especially, it is verified if the publishing of the whole transform branch, within the */tf* topic, from this frame to another (base reference) is up to date, considering the laser's message time stamp.

This */tf* topic is responsible for holding every transform between all frames within the ROS environment. As the servo-related rotation would be included among those transforms, dealt with by another process entirely, the projection from polar to Cartesian coordinates here explained would implicitly transform the cloud of points according to the servo's rotation.

But this is a subject for later worksheets, and so the projection is, for now, kept simple and the point cloud's tilting servo-driven rotation is explicitly handled further on this node.

In this callback though, the remaining lines are just configuring the rest of the *PointCloud2* message and publishing it via *cloud_pub_ publish* call. Next, to set up the node the main function should initialize the ROS node and instantiate the ***LaserScan_2_PointCloud*** object:

```
int main(int argc, char** argv)
{
  ros::init(argc, argv, "LaserScanToPointCloud");
  ros::NodeHandle n;
  LaserScan_2_PointCloud ls2pc(n);
  ros::spin();
  return 0;
}
```

To build the node, remember to add the appropriate *CMakeLists.txt* calls for the node to be created:

```
find_package(catkin REQUIRED COMPONENTS
  laser_geometry
  roscpp
  rospy
  sensor_msgs
)

(…)

catkin_package(
  INCLUDE_DIRS include
  LIBRARIES lesson6_package_pcl
  CATKIN_DEPENDS laser_geometry roscpp rospy sensor_msgs
  DEPENDS system_lib
)

(…)


add_executable(lesson6_laser_2_cloud_node src/laser_2_cloud.cpp)
target_link_libraries(lesson6_laser_2_cloud_node  ${catkin_LIBRARIES})
```

```
$ cd ~/trsa_ws
$ catkin_make
```

You can now create a launch file, *laser_2_cloud.launch* on a launch folder, to run your node and remap the *LaserScan* topic to match the name of the one published on the Bagfile playback:

```
<launch>
   <node pkg="lesson6_package_pcl" type="lesson6_laser_2_cloud_node" name="laser_2_cloud_node" output="screen"
           clear_params="true">
      <remap from="/laser_scan" to="/scan" />
   </node>
</launch>
```

Hence, you will just have to launch your converter node:

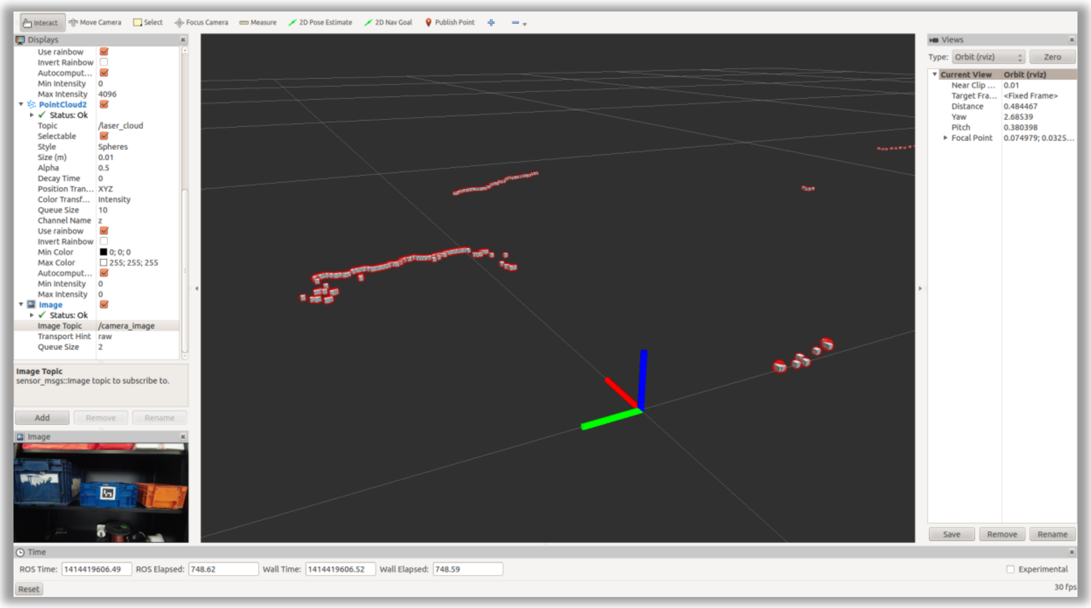
```
$ roslaunch lesson6_package_pcl laser_2_cloud.launch
```

Now, before you playback the Bagfile and have the node *laser_2_cloud_node* publishing the *PointCloud2* version of the laser's output, take some time to run and configure the ROS visualization tool RViz. To do that, run the RViz node follow the instructions to setup the displays:

```
$ rosrun rviz rviz
```

- Set the Global Options > Fixed Frame to */hokuyo_laser_link* as there is no other reference frame.

- **Add** a new Display: **Axes** to show the laser's viewpoint fixed frame. Change the **Length** to *0.05*, **Radius** to *0.005*.
- **Add** a new Display: **LaserScan** to show the node's input *LaserScan* message. Change the **Style** to *Boxes*, **Box Size** to *0.005*, and the **Topic** to */scan*.
- **Add** a new Display: **PointCloud2**, name it ***LaserCloud***, to show the node's output *PointCloud2* message. Change the **Style** to *Spheres*, **Sphere Size** to *0.01*, the ***alpha*** to *0.5*, the **Channel Name** to *z*, uncheck **AutoCompute**, typing *-0.1* and *0.4* for the **Min** and **Max Intensity**, and, finally, the **Topic** to */laser_cloud*. Note: you will only be able to set some of these attributes after a message has been received.
- **Add** a new Display: **Image** subscribing the **Topic** */camera_image* for you to see the live feed from the camera (dock the display to the RViz window's left bottom corner).



Use rainbow	✓
Invert Rainbow	☐
Autocomput...	✓
Min Intensity	0
Max Intensity	4096
PointCloud2	✓
Status: Ok	
Topic	/laser_cloud
Selectable	✓
Style	Spheres
Size (m)	0.01
Alpha	0.5
Decay Time	0
Position Tran...	XYZ
Color Transf...	Intensity
Queue Size	10
Channel Name	z
Use rainbow	✓
Invert Rainbow	☐
Min Color	■ 0; 0; 0
Max Color	☐ 255; 255; 255
Autocomput...	✓
Min Intensity	0
Max Intensity	0
Image	✓
Status: Ok	
Image Topic	/camera_image
Transport Hint	raw
Queue Size	2

Image Topic
sensor_msgs::Image topic to subscribe to.

Type: Orbit (rviz) Zero

Current View	Orbit (rviz)
Near Clip ...	0.01
Target Fra...	<Fixed Frame>
Distance	0.484467
Yaw	2.68539
Pitch	0.380398
Focal Point	0.074979; 0.0325...

ROS Time: 1414419606.49 ROS Elapsed: 748.62 Wall Time: 1414419606.52 Wall Elapsed: 748.59 ☐ Experimental

30 fps

Figure 3 – RViz displays the 2D Laser Scan and untransformed Point Cloud message.

This way it will be possible to see the messages being passed through ROS environment in a more friendly way. For more information on ROS's visualization tool check http://wiki.ros.org/rviz.

Now playback the Bagfile and you should see something similar to Figure 3. You can see *laser_2_cloud_node* is converting the *LaserScan* to a *PointCloud2* message correctly if all Laser Scan boxes are enveloped by a Point Cloud sphere.

The next is to rotate the conversion's output cloud considering the available tilting angle, provided through the */tilt_angle* topic. Some modifications have to be inducted to the whole package:

- The package.xml file should state the dependencies on *std_msgs*, as the servo's angle comes as a *std_msgs::Float64*, and also pcl_conversions and pcl_ros to, respectively, access these packages' ROS message / PCL types conversion and the transform functionality upon a whole PCL point cloud.

```
<build_depend>std_msgs</build_depend>
<build_depend>pcl_conversions</build_depend>
<build_depend>pcl_ros</build_depend>

<run_depend>std_msgs</run_depend>
<run_depend>pcl_conversions</run_depend>
<run_depend>pcl_ros</run_depend>
```

- The *CMakeLists.txt* file's *find_package* and *catkin_package* calls should also be updated with the references to *std_msgs* and *pcl_ros*:

```
find_package(catkin REQUIRED COMPONENTS
  laser_geometry
  roscpp
  rospy
  sensor_msgs
  std_msgs
  pcl_ros
)

(...)

catkin_package(
  INCLUDE_DIRS include
  LIBRARIES lesson6_package_pcl
  CATKIN_DEPENDS laser_geometry roscpp rospy sensor_msgs std_msgs pcl_ros
  DEPENDS system_lib
)
```

- The source file, *laser_2_cloud.cpp*, should now:
  - Include the respective headers:

```
#include "std_msgs/Float64.h"
#include <pcl_conversions/pcl_conversions.h>
#include <pcl_ros/transforms.h>
```

  - Add to the ***LaserScan_2_PointCloud*** class two new members: a new subscriber, to deal with the servo's incoming *Float64* tilt angle message, and a *Eigen::Matrix4f* transform object:

```
  ros::Subscriber scan_sub_, servo_sub_;

(...)

  Eigen::Matrix4f transform;
```

  - Configure the subscriber and initialize the transform matrix on the class's constructor:

```
    servo_sub_ = n_.subscribe("/servo_tilt_angle", 1000, &LaserScan_2_PointCloud::servoCallback, this);
    transform = Eigen::Matrix4f::Identity();
```

  - Have the member ***servoCallback*** defined to adapt the transform matrix to include the desired rotation:

```
  void servoCallback(const std_msgs::Float64::ConstPtr& tilt_angle){
    transform (0,0) = cos (-M_PI + tilt_angle->data);
    transform (0,2) = sin(-M_PI + tilt_angle->data);
    transform (2,0) = -sin (-M_PI + tilt_angle->data);
    transform (2,2) = cos (-M_PI + tilt_angle->data);
  }
```

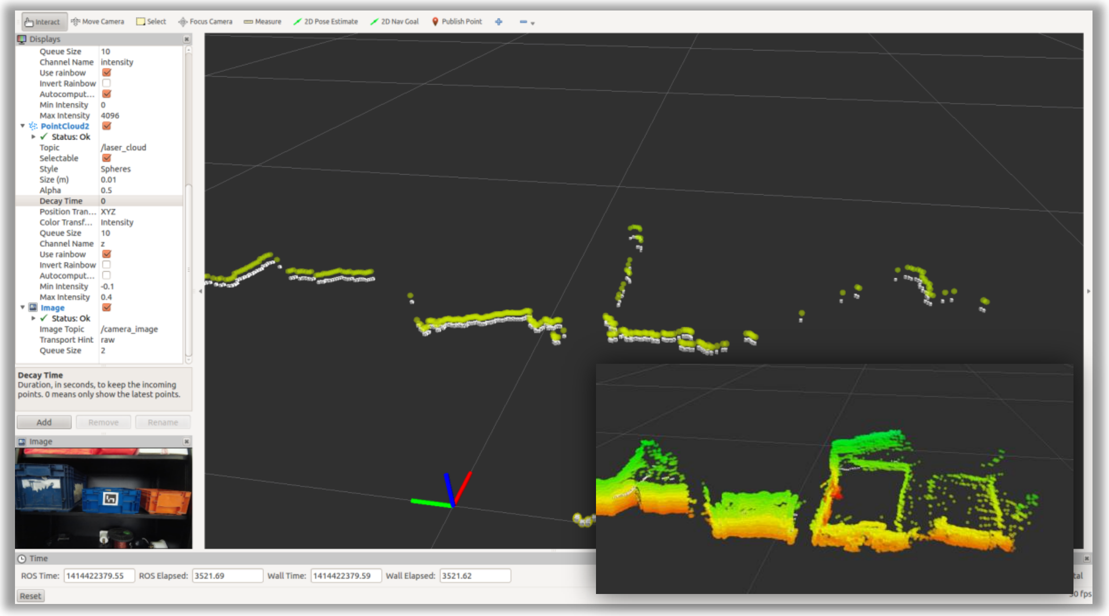
To recall on transform matrixes, the rotation component in the y-axis of θ, here ***tilt_angle***, is defined by $R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}$, where the whole transform $T_y(\theta) = \begin{bmatrix} R_y(\theta) & 0_{3x1} \\ 0_{1x3} & 1 \end{bmatrix}$. The *–M_PI* is just to compensate the servo's incoming tilt angle offset.

- Modify the *LaserScan* callback, to submit the converter's *PointCloud2* object to the current transform, through the call on PCL's *pcl::transformPointCloud* functionality:

```
void scanCallback (const sensor_msgs::LaserScan::ConstPtr& scan_in)
{
  sensor_msgs::PointCloud2 cloud;
  projector_.projectLaser(*scan_in, cloud);

  pcl::PointCloud<pcl::PointXYZ>::Ptr pcl_cloud (new pcl::PointCloud<pcl::PointXYZ> ());
  pcl::fromROSMsg(cloud, *pcl_cloud);
  pcl::transformPointCloud(*pcl_cloud, *pcl_cloud, transform);
  pcl::toROSMsg(*pcl_cloud, cloud);

  cloud.header.stamp = ros::Time::now();
  cloud.header.frame_id = scan_in->header.frame_id;
  cloud_pub_.publish(cloud);
}
```

  In this code segment, you can see the transportation between ROS *sensor_msgs::PointCloud2* to PCL's *pcl::PointCloud*, by calling *pcl::fromROSMsg*, the ***transform*** object being applied to each the PointXYZ cloud point through the use of *pcl::transformPointCloud*, terminated by the point cloud's reverse conversion from PCL to ROS type, *pcl::toROSMsg*.

Run *catkin_make* and, after you remap the servo's tilt angle topic on your launch file, as it was done for the laser scan's, re-launch the node. On RViz you will be able to see now the *LaserCloud* display's cloud moving up and down, like in Figure 4.



Queue Size	10
Channel Name	intensity
Use rainbow	✓
Invert Rainbow	☐
Autocomput...	✓
Min Intensity	0
Max Intensity	4096
PointCloud2	✓
✓ Status: Ok	
Topic	/laser_cloud
Selectable	✓
Style	Spheres
Size (m)	0.01
Alpha	0.5
Decay Time	0
Position Tran...	XYZ
Color Transf...	Intensity
Queue Size	10
Channel Name	z
Use rainbow	✓
Invert Rainbow	☐
Autocomput...	☐
Min Intensity	-0.1
Max Intensity	0.4
Image	✓
✓ Status: Ok	
Image Topic	/camera_image
Transport Hint	raw
Queue Size	2

Decay Time
Duration, in seconds, to keep the incoming points. 0 means only show the latest points.

ROS Time: 1414422379.55 ROS Elapsed: 3521.69 Wall Time: 1414422379.59 Wall Elapsed: 3521.62

Figure 4 – Point Cloud transformed by the servo's tilt angle.

Change the **Decay Time** feature on the ***LaserCloud*** display to ***20*** (seconds), for example, and the consecutive point clouds will accumulate for that period, showing a virtual reconstruction of the scene (see Figure 4's bottom left window).

4. Point Cloud Integration on an Octree Structure

There are two main approaches to storing three-dimensional occupancy. One, point-based, accumulates the sensors' continuous measures on an ever-extending point cloud. The other, voxel-based, partitions space into volumetric cubic elements, having each of those store a single occupancy measure to be somehow representative of enclosed sensor readings. Each solution has its merits and its faults, and the decision on either one affects the whole subsequent processing concept and paradigm.

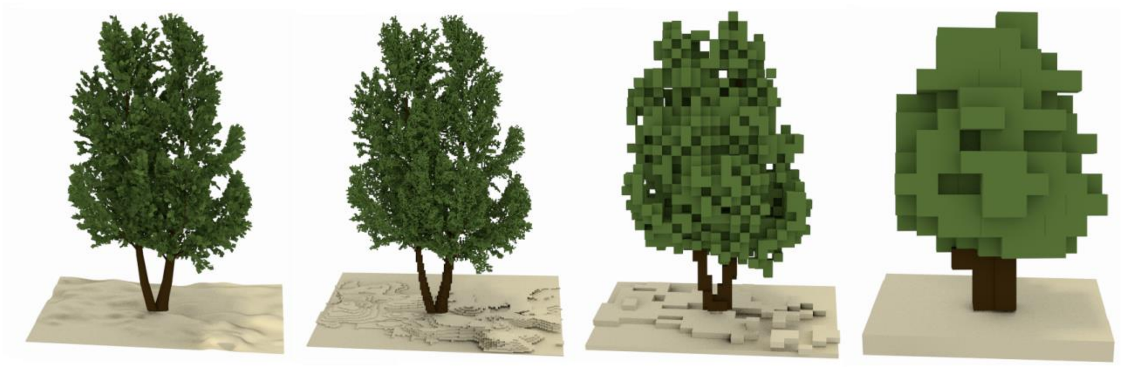


Figure 5 - Objects distortion caused by undersampling space. The point cloud (left) becomes distorted and inflated as its representation's resolution lessens, i.e., as the voxel's size is increased (from left to right).

As the former offers the maximum possible detail, it also requires a whole lot more memory to be allocated just to store information. Consequently, all algorithms around that information will entail heavier processing. The latter is bound to carry some loss of information resulting from spatial quantization errors (see Figure 5), no matter high its resolution. Nonetheless, its voxels' occupancy upkeep mechanism may provide an answer to a problem the point-based approach is particularly susceptible to: ill registered occupancy readings.

Here we'll opt for a voxel-based approach to store those occupancy *PointCloud2* readings. The OctoMap is an octree-based 3-D mapping structure with its own algorithm-wrapping C++ library specifically designed for robotics, already integrated within ROS and PCL. One of its most important features is the capability to deal with sensor noise and dynamic environment changes by probabilistically modeling free and occupied space.

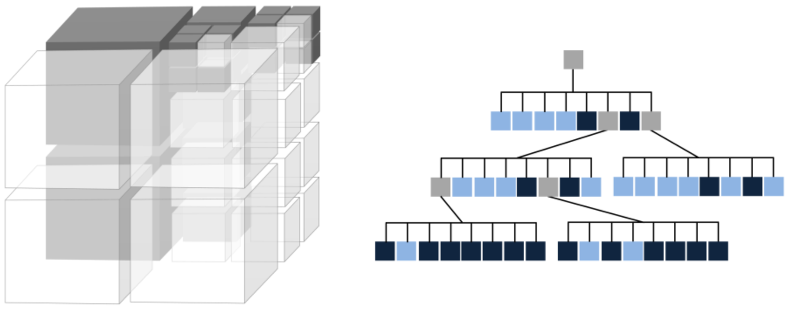


Figure 6 - Octree space division. (Left) Example tree representation of structure with occupied (black) and free nodes (white/transparent). (Right) Corresponding tree with free (light blue), occupied (dark blue) and unpruned nodes (grey). Voxels' occupancy states are fully represented through free, occupied and unknown leaf nodes. Free and occupied states are rendered by a probabilistic value, leaving unmapped volumes to be represented through uninitialized nodes.

In simple terms, a range reading is submitted to raycasting to assess which voxels in the structure should be affected free, i.e., the ones corresponding to the clear line of sight between sensor and hit object, and which voxel the object belongs to, and so affected occupied. Then each voxel is hinted either free or occupied with a certain weight, which should reflect the confidence credited to each measurement. The voxel's probabilistic measure is updated through a log-odd sum, and the confidence on its value is determined by how close it comes to either free or occupied stable clamping thresholds, l_{min} and l_{max}. The confidence in each measurement itself is dimensioned against these thresholds, and how its values for free, l_{miss}, and occupied, l_{hit}, bring a voxel to being stable. For more detailed information see OctoMap.

Let's then start to install all required dependencies:

```
$ sudo apt-get install ros-indigo-octo*
```

Then clone the *octomap_mapping* package group (http://wiki.ros.org/octomap_mapping) enclosing the provided *octomap_server* package, including the node source files.

```
$ git clone https://github.com/OctoMap/octomap_mapping.git
```

You will now create custom *launch* (*octomap_mapping.launch*) and *yaml* (*octomap_mapping.yaml*) files to launch the *octomap_server* node in a way that suits the application.

../lesson6_package_pcl/launch/octomap_mapping.launch

```
<launch>
  <node pkg="octomap_server" type="octomap_server_node" name="octomap_server" output="screen"
        clear_params="true">
    <rosparam command="load" file="$(find lesson6_package_pcl)/params/octomap_mapping.yaml" />
    <remap from="/cloud_in" to="/laser_cloud" />
    <remap from="/octomap_point_cloud_centers" to="/octomap_cloud" />
  </node>
</launch>
```

../lesson6_package_pcl/params/octomap_mapping.yaml

```
resolution: 0.01
frame_id: hokuyo_laser_link
base_frame_id: hokuyo_laser_link
sensor_model/max_range: 4.0
```

On this worksheet we'll just use the OctoMap in a simple way, using most of its default configuration, it will be up to you to explore its features on your own time. Here, only the laser range specification, the hokuyo's 4 m reach, and the desired minimum leaf size, 1 cm, are submitted in this parameter file.

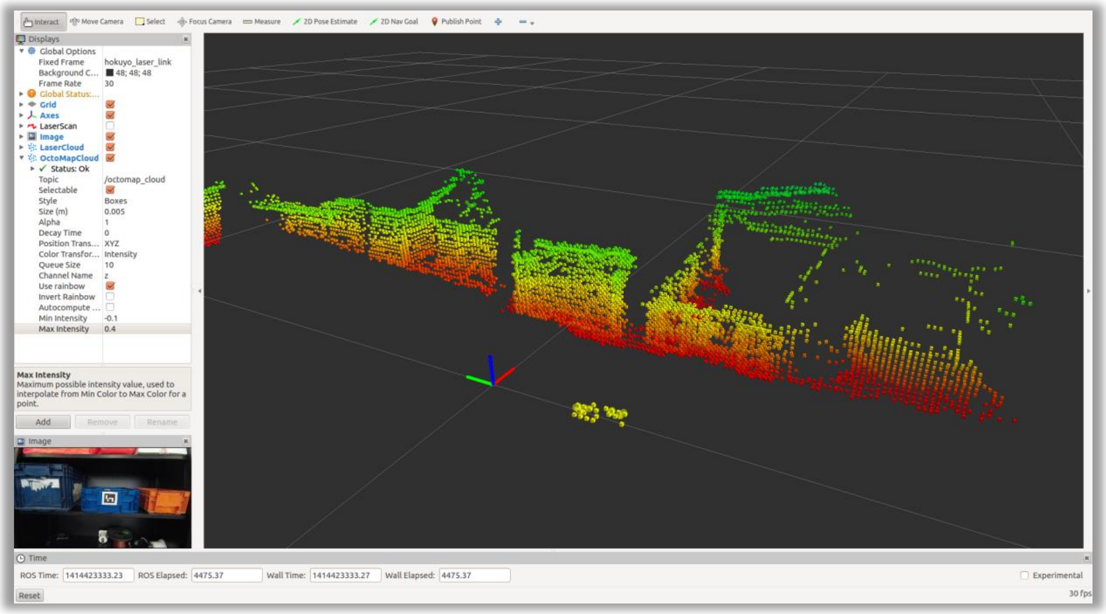
Setting the node's *frame_id* and *base_frame_id* the same is just a tweak to overrun the inner *tf* transforms the *octomap_server* node evaluates when aggregating all sensory information on a single reference frame. Here, the absence of transforms under the */tf* topic, makes it necessary to make such an adjustment.

Furthermore, the default probability values for this *octomap_server* are $p_{min} = 0.12$, $p_{max} = 0.97$, $p_{miss} = 0.4$, $p_{hit} = 0.7$. This was set for large outdoor mapping as the stable thresholds were set very far from the hit and miss probabilities, and the unknown default value, 0.5. Those also establish a higher belief on hits over free space, due to the fact that clearances are usually not to be so easily trusted. Although not optimal for this indoor problem, where far less noise and registration problems are to be expected, they offer a decent parameterization. You are invited to change these parameters and see the effects on your own time.

So, run *catkin_make* and roslaunch the *octomap_server* and, assuming the *laser_2_cloud_node* is still active, replay the Bagfile.

```
$ roslaunch lesson6_package_pcl octomap_mapping.launch
```

Now, on the RViz window, add a new Display for a *PointCloud2*, *name it OctoMapCloud*. Set the ***Topic*** ***/octomap_cloud***, to subscribe the *octomap_server* node's fully integrated output cloud. Also, set the **Style** to ***Boxes*** with **Size** ***0.005***, **Channel Name** ***z***, uncheck the **AutoCompute** and set the same limit values for the **Intensity** ***-0.1*** and ***0.4***, for **Min** and **Max** respectively. Finally, uncheck the ***LaserScan*** display and reset the ***LaserCloud***'s **Decay Time** to ***0***. You will be able to see something like Figure 7.



```
Global Options
  Fixed Frame        hokuyo_laser_link
  Background C...    48; 48; 48
  Frame Rate         30
Global Status:...
Grid                 ✓
Axes                 ✓
LaserScan            ☐
Image                ✓
LaserCloud           ✓
OctoMapCloud         ✓
  Status: Ok
  Topic              /octomap_cloud
  Selectable         ✓
  Style              Boxes
  Size (m)           0.005
  Alpha              1
  Decay Time         0
  Position Trans...  XYZ
  Color Transfor...  Intensity
  Queue Size         10
  Channel Name       z
  Use rainbow        ✓
  Invert Rainbow     ☐
  Autocompute ...    ☐
  Min Intensity      -0.1
  Max Intensity      0.4

Max Intensity
Maximum possible intensity value, used to interpolate from Min Color to Max Color for a point.

ROS Time: 1414423333.23   ROS Elapsed: 4475.37   Wall Time: 1414423333.27   Wall Elapsed: 4475.37
```

Figure 7 – Octomap integrating the transformed laser's cloud.

5. Identifying the Boxes

In this final step the OctoMap's output cloud will be processed through filtering, sample consensus model fitting and Euclidean clustering to separate the point cloud into two groups, each corresponding to the two bigger boxes. A new node, *lesson6_box_segmentation_node*, will be created, subscribing the */octomap_cloud* topic, and publishing a new *PointCloud2* message, where only the points identified as belonging to boxes will remain. Also, the message's *intensity* channel will indicate which points belong to one box and the other.

First let's overview the header files included in this node's source file, *box_segmentation.cpp*:

```
#include "ros/ros.h"
#include "sensor_msgs/PointCloud.h"
#include <pcl_conversions/pcl_conversions.h>
#include <pcl_ros/transforms.h>

#include <pcl/filters/passthrough.h>
#include <pcl/filters/radius_outlier_removal.h>
#include <pcl/filters/extract_indices.h>

#include <pcl/ModelCoefficients.h>
#include <pcl/point_types.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/kdtree/kdtree.h>
#include <pcl/segmentation/extract_clusters.h>
```

Then define the main class for this node with the following members:

```
class Box_Segmentation{

public:

   ros::NodeHandle n_;
   ros::Subscriber cloud_sub_;
   ros::Publisher treated_cloud_pub_;

   Box_Segmentation(ros::NodeHandle n) :
   n_(n)
   {
      cloud_sub_ = n_.subscribe("/octomap_cloud", 1000, &Box_Segmentation::cloudCallback, this);
      treated_cloud_pub_ = n_.advertise<sensor_msgs::PointCloud2>("/box_cloud",1);
   }

   void cloudCallback (const sensor_msgs::PointCloud2::ConstPtr& cloud_in)
   {

    (...)

   }
};
```

And the main function as such:

```
int main(int argc, char** argv)
{
  ros::init(argc, argv, "BoxSegmentation");
  ros::NodeHandle n;
  Box_Segmentation bs(n);
  ros::spin();
  return 0;
}
```

Now, the major processes will be included whenever an OctoMap's *PointCloud2* message is received and the ***cloudCallback*** is called into action. So let's examine carefully the contents on this function:

- Starting by converting the *sensor_msgs::PointCloud2* into *pcl::PointCloud* type:

```
    pcl::PointCloud<pcl::PointXYZ>::Ptr pcl_cloud (new pcl::PointCloud<pcl::PointXYZ> ());
    pcl::fromROSMsg(*cloud_in, *pcl_cloud);
```

- Next, to restrain the region of interest to a small area right in front of the laser scanner, a *pcl::PassThrough* filter is applied to eliminate all points farther than 0.5 m distance from the sensor in the *y*-axis.

```
pcl::PassThrough<pcl::PointXYZ> pass;
pass.setFilterFieldName("y");
pass.setFilterLimits(-0.5, 0.5);
pass.setInputCloud(pcl_cloud);
pass.filter(*pcl_cloud);
```

- Followed by a model fitting process to a plane perpendicular to the *x*-axis (***setAxis(Eigen::Vector3f(1, 0, 0))***). The sample consensus method of choice is RAndom SAmple Consensus (*SAC_RANSAC*), configured to accept inliers up to 3 cm distanced to the hypothetical plane model (***setDistanceThreshold (0.03)***), which can vary up to 0.3 radians from the default upright plane (***setEpsAngle (0.02)***). The number of iterations for the RANSAC method to find the desired plane model amongst the point samples is maintained at 50 (default value). Only the inliers are then transported as the output ***treated_cloud***. For more detail on PCL's sample_consensus and segmentation modules refer to http://pointclouds.org/ documentation/tutorials.

```
pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
pcl::SACSegmentation<pcl::PointXYZ> seg;
seg.setOptimizeCoefficients (true);
seg.setModelType (pcl::SACMODEL_PERPENDICULAR_PLANE);
seg.setMethodType (pcl::SAC_RANSAC);
seg.setDistanceThreshold (0.03);
seg.setAxis (Eigen::Vector3f(1, 0, 0));
seg.setEpsAngle (0.02);
seg.setInputCloud (pcl_cloud);
seg.segment (*inliers, *coefficients);

if (inliers->indices.size () == 0)
  return;
pcl::PointCloud<pcl::PointXYZ>::Ptr treated_cloud (new pcl::PointCloud<pcl::PointXYZ> ());
for (std::vector<int>::const_iterator it = inliers->indices.begin(); it != inliers->indices.end (); ++it)
  treated_cloud->push_back(pcl_cloud->points[*it]);
```

- A radial outlier filter is applied to remove any points linking the two boxes' front side planes.

```
pcl::RadiusOutlierRemoval<pcl::PointXYZ> radialFilter;
radialFilter.setInputCloud(treated_cloud);
radialFilter.setRadiusSearch(0.03);
radialFilter.setMinNeighborsInRadius (20);
radialFilter.filter (*treated_cloud);
```

- Finally, a clustering algorithm based on the Euclidean distance between the points is applied, to distinguish the two boxes. First, the points are submitted to a K-d Tree to ease the search space for the nearest neighbor search of points. The distance threshold is set to 2 cm (***setClusterTolerance (0.02)***) and the minimum number of points needed to a cluster to be formed is 100 (***setMinClusterSize (100)***).

```
pcl::search::KdTree<pcl::PointXYZ>::Ptr kdtree (new pcl::search::KdTree<pcl::PointXYZ>);
kdtree->setInputCloud (treated_cloud);
std::vector<pcl::PointIndices> cluster_indices;
pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
ec.setClusterTolerance (0.02);
ec.setMinClusterSize (100);
ec.setMaxClusterSize (10000);
ec.setSearchMethod (kdtree);
ec.setInputCloud (treated_cloud);
ec.extract (cluster_indices);
pcl::PointCloud<pcl::PointXYZI>::Ptr cluster_cloud (new pcl::PointCloud<pcl::PointXYZI> ());
pcl::PointXYZI cluster_point;
double cluster_final_average;
int cluster_id=0;
for (std::vector<pcl::PointIndices>::const_iterator it = cluster_indices.begin (); it != cluster_indices.end ();
  ++it, ++cluster_id)
{
  for (std::vector<int>::const_iterator pit = it->indices.begin (); pit != it->indices.end (); pit++)
  {
    cluster_point.x = treated_cloud->points[*pit].x;
    cluster_point.y = treated_cloud->points[*pit].y;
    cluster_point.z = treated_cloud->points[*pit].z;
    cluster_point.intensity = cluster_id;
    cluster_cloud->push_back(cluster_point);
  }
}
```

Note the *pcl::PointCloud* container for the output cloud, ***cluster_cloud***, including point types pcl::PointXYZI. This way there is an intensity channel that can be used to store a point's the cluster identification (***cluster_id***).

The callback function is then finallized by setting up the sensor_msgs::PointCloud2 message with the clustered cloud information, and publishing it.

```
sensor_msgs::PointCloud2 cloud;
pcl::toROSMsg(*cluster_cloud, cloud);
cloud.header.stamp = ros::Time::now();
cloud.header.frame_id = cloud_in->header.frame_id;
treated_cloud_pub_.publish(cloud);
```

Now the node's source code is complete, edit the *CMakeLists.txt* file to create this very node upon package build

```
add_executable(box_segmentation_node src/box_segmentation.cpp)
target_link_libraries(box_segmentation_node ${catkin_LIBRARIES})
```

Run *catkin_make*, and *relaunch/rerun* all tree nodes.

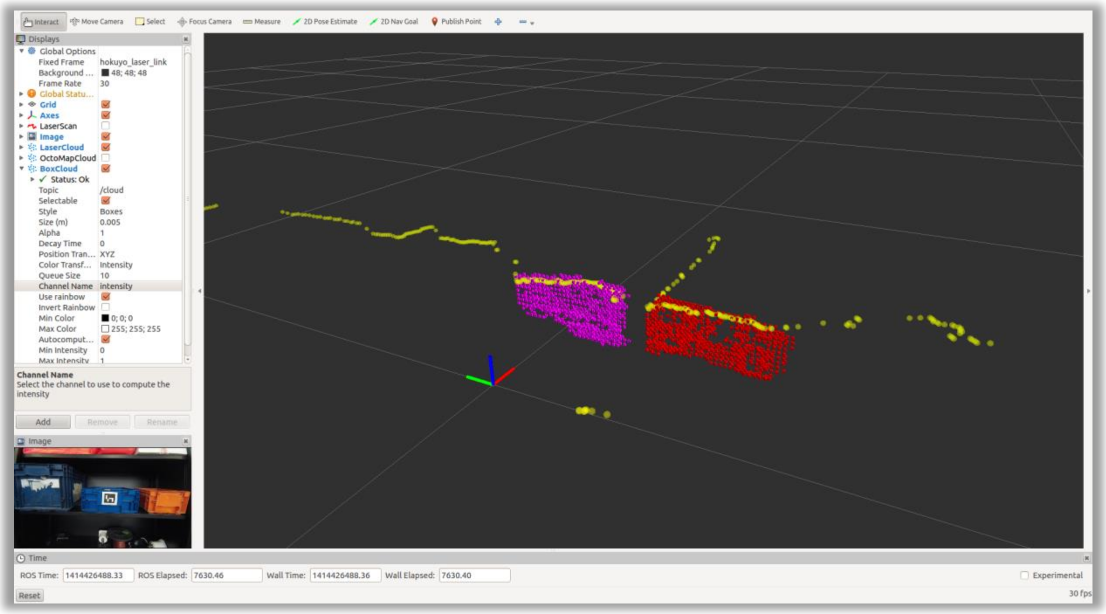
```
$ roslaunch lesson6_package_pcl laser_2_cloud.launch
```

```
$ roslaunch lesson6_package_pcl octomap_mapping.launch
```

```
$ rosrun lesson6_package_pcl box_segmentation_node
```

Add a new **PointCloud2** Display to RViz to subscribe the */box_cloud* topic, set the **Style** to ***Boxes***, **Size** to ***0.005*** and the **Channel Name** to ***intensity***. Also, uncheck the ***OctoMapCloud*** Display.

As you playback the bagfile again, you will be able to see two groups of points discriminated by their color, on most iterations at least, like it is depicted on Figure 8.

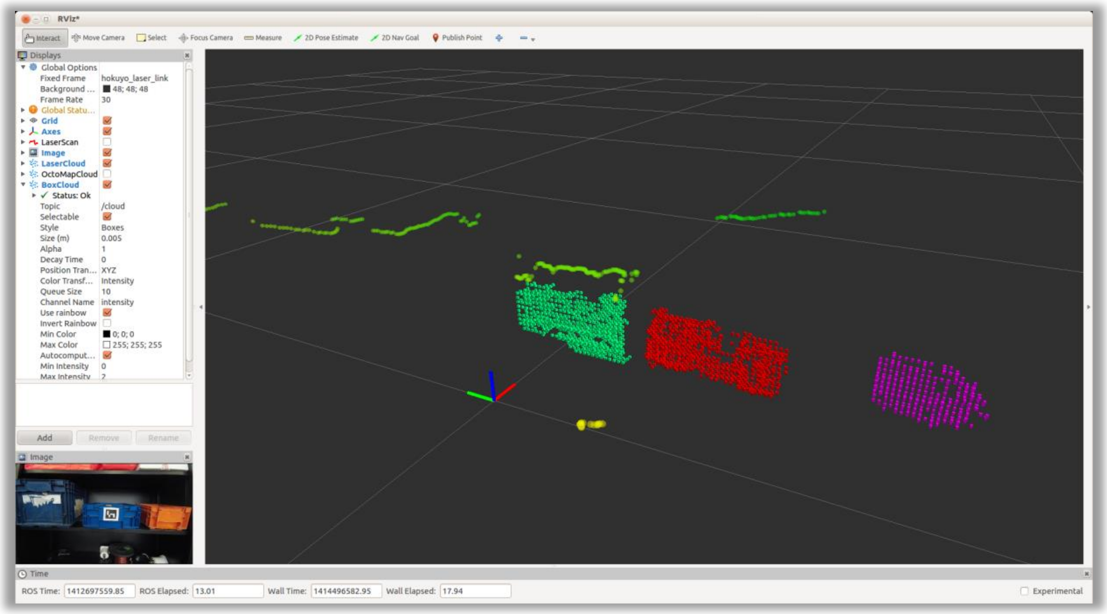


```
Global Options
  Fixed Frame        hokuyo_laser_link
  Background ...     48; 48; 48
  Frame Rate         30
Global Statu...
Grid                 ✓
Axes                 ✓
LaserScan            ☐
Image                ✓
LaserCloud           ✓
OctoMapCloud         ☐
BoxCloud             ✓
  Status: Ok
  Topic              /cloud
  Selectable         ✓
  Style              Boxes
  Size (m)           0.005
  Alpha              1
  Decay Time         0
  Position Tran...   XYZ
  Color Transf...    Intensity
  Queue Size         10
  Channel Name       intensity
  Use rainbow        ✓
  Invert Rainbow     ☐
  Min Color          0; 0; 0
  Max Color          255; 255; 255
  Autocomput...      ✓
  Min Intensity      0
  Max Intensity      1

Channel Name
Select the channel to use to compute the intensity

ROS Time: 1414426488.33   ROS Elapsed: 7630.46   Wall Time: 1414426488.36   Wall Elapsed: 7630.40   ☐ Experimental
```

Figure 8 – Final Box front side discrimination.

6. Final Exercise – Detect the Remaining Box

The goal of this exercise is to consolidate the PCL tools introduced on the last chapters, and invite you to explore and apply any at your disposal on http://pointclouds.org/documentation/. The desired end product is that the rightmost orange box gets detected also, just as the other two were: by identifying the points belonging to the box's front side (see Figure 9).



```
Global Options
  Fixed Frame        hokuyo_laser_link
  Background ...     48; 48; 48
  Frame Rate         30
Global Statu...
Grid                 ✓
Axes                 ✓
LaserScan            ☐
Image                ✓
LaserCloud           ✓
OctoMapCloud         ☐
BoxCloud             ✓
  Status: Ok
  Topic              /cloud
  Selectable         ✓
  Style              Boxes
  Size (m)           0.005
  Alpha              1
  Decay Time         0
  Position Tran...   XYZ
  Color Transf...    Intensity
  Queue Size         10
  Channel Name       intensity
  Use rainbow        ✓
  Invert Rainbow     ☐
  Min Color          0; 0; 0
  Max Color          255; 255; 255
  Autocomput...      ✓
  Min Intensity      0
  Max Intensity      2

ROS Time: 1412697559.85  ROS Elapsed: 13.01  Wall Time: 1414496582.95  Wall Elapsed: 17.94  ☐ Experimental
```

Figure 9 – Box Full Segmentation

The */box_cloud* topic's point cloud should then include the points corresponding to the orange box signaled properly as part of a different cluster (your student's number, for example), on the *intensity* channel.

This, of course, means that you will have to make some minor adjustments to your *box_segmentation.cpp* PCL processing. Yet, a few restrictions are in order though. You need to keep, at least, all those steps for the new box: filtering, sample consensus and clustering (just to not oversimplify the problem). Needless to say the parameters may very well need to change.

Deliver lesson6_package_pcl package you have been working on, with all files included, on a zip file named T6_PCL_XXXXX.zip (where XXXXX is your student's number) to Moodle platform until November 5th, 23:55.