University of Genoa

EmaroLab

Author: Buoncompagni Luca.
Supervisor: Mastrogiovanni Fulvio.

Date: November 3, 2016.
Topic: Technical report *I*.
PhD Curriculum in: Robotics & Autonomous Systems.

Primitive shapes segmentation from point clouds.

Abstract: This document shows a ROS implementation that takes a point cloud data from a Kinect in order to perform: plane, sphere cone and cylinder segmentation. Moreover, the objects are supposed to be placed on top of tables and the acquisition is done without any memory states. Also, a procedure to detach supports and objects on top of it are discussed as well. Finally, also experiments results are shown with the purpose to evaluate recognition rates and computational latencies.

Contents

1 *Introduction & requirements*

Here we spend some words to provide the preliminary knowledge required to build a semantic description of objects in the environment. The final aim is to describe the geometrical properties of the objects as well as their absolute location and relative relationship among them[1]. Anyway, the work done so far has the purpose to successfully recognise objects and assign to them a theoretic primitive shape, obtaining also geometrical information about their coefficients.

For a practical point of view let us stress, in this paragraph, the requirements of the system. First of all, it has been implemented using: ROS[Quigley et al., 2009] *Hydro*[2], secondly, the algorithm is already configured to pick data either from stimulator or from real device (not at the same time) since both of them stream data into the ROS node:

```
"camera/depth/points"
```

The simulation has been tested with *Gazebo* 1.9.6-1-precise (ROS interface package version: 2.3.7-0precise). Furthermore, in the simulator the default *turtle-bot* world can be used in order to get a virtual Kinect, which streams data for the system (ROS turtle-bot package version 2.1.1-1precise). Furthermore, real experiment has been performed using a real Kinect (360) which has been connected to ROS through the package *Openni*[Guide, 2011] (openni-camera version 1.9.2-oprecise, openni-launch 1.9.5-oprecise)[3].

[1] e.g.: object *A* is behind the object *B*. Object *A* is on the left of object *B*. Object *A* has the most right position. And even: *A* was on the left of *B*.

Also, consider that the system is based on the Point Cloud Library (PCL) [Rusu and Cousins, 2011] which collects useful procedures to process point clouds.

[2] with catkin workspace loaded from eclipse in Ubuntu 12.04.05-64bit.

Required package for the software:

roscpp	:	1.10.12-oprecise[1]
std_msg	:	0.5.8-oprecise
sensor_msg	:	1.10.6-oprecise
message_generation	:	0.2.10-oprecise
pcl_conversions	:	0.1.6-oprecise
pcl_ros	:	1.1.11-oprecise
pcl_msgs	:	0.1.0-oprecise

Listed in *CMaketLsits.txt*

[3] calibration has not been done yet.

2 *ROS architecture structure*

As it is possible to see from Figure 1, the system is designed to have a central node (9) which retrieves input data in order to make computations and assess object's primitive features, if any. Such computational steps may require external services that would be called in

Where used messages are located in the default folder *srv*, with files:

- $\mathscr{A}$: PointCloud2Ptr
- $\mathscr{B}$: DeepFilter.srv
- $\mathscr{C}$: SupportSegmentation.srv
- $\mathscr{D}$: ClusterSegmentation.srv
- $\mathscr{E}$: PrimitiveSegmentation.srv
- $\mathscr{F}$: (not implemented jet)

where the service C uses also a custom message: *InlierSupport.msg*. While D, is based also in: *InlierCluster.msg*; contained in the *msg* folder.

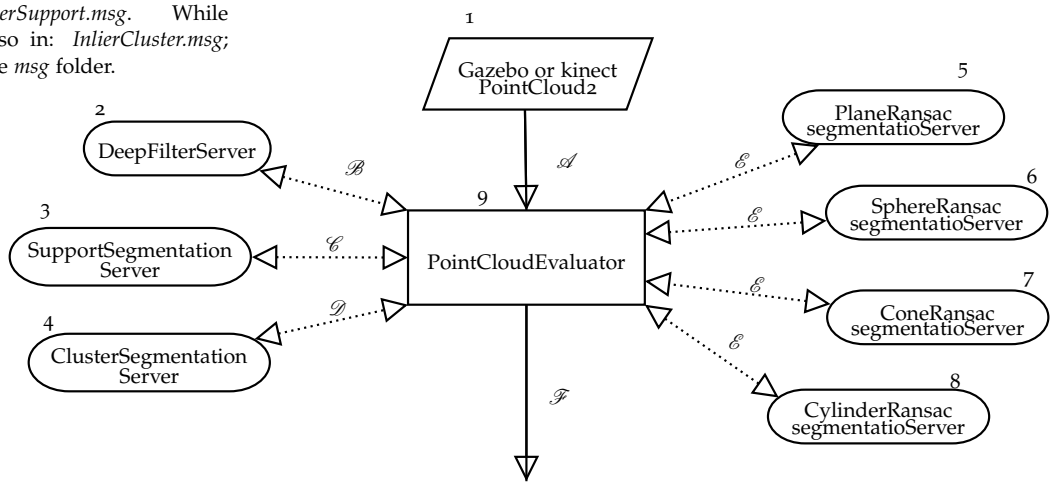


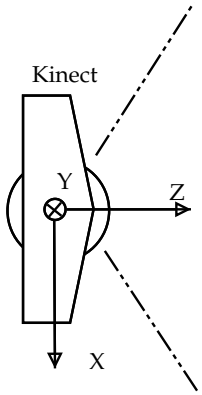
Figure 1: the ROS architecture. Squared are node while ellipse are services and arrow messages.

order to pass through all the stages of the algorithm and then loop forever. Important to notice that the output message of the system has not been implemented yet.

The type of input into the system is a point cloud[4] $\mathcal{P}$. Then, those are mapped into the PCL standard format[5] that are an array of points described through the spatial components $[x_i, y_i, z_i]$ $\forall i \in \mathcal{P}$ with respect to the referencing system shown in Figure 2. Last but not the least, the other messages of the architecture are specifically designed for this task and listed in the following paragraphs of this section.

Let us analyse, in the following sections of this documents, all the components shown in Figure 1. More specifically, for each of them, the inputs and outputs will be addressed in details. Moreover, an algorithm for every components is proposed with the purposes to clarify: the role of all the parameters, the experiments and the results. In fact, last part of this document will be focused on a real experience and the conclusion that are possible to draw from there.

Figure 2: The used kinetic reference system.



[4] in the standard ROS format sensor_msgs/PointCloud2.msg.

[5] pcl::*PointXYZ*. Note that this feature has been parametrised for easy changes. Moreover, consider that it describes coordinates in meters.

2.1 *Deep filtering server*

This server is always called to compute the first manipulation on the complete point cloud. Its purposes is to eliminate all the points that have a deep distance, considered to be on the Z axis, further than an input parameter: *deepThreshold* ($\bar{d}$)[6]. In practice, it just divides an input cloud $\mathcal{P}$ in two different output clouds: $\mathcal{P}^{>}$,$\mathcal{P}^{<}$; following the Algorithm 1.

This algorithm uses the message $\mathscr{B}$ to communicate with the main note of the system; follow the description of such a message:

- **Inputs**
  - *sensor_msgs/PointCloud2* ***cloud***: ($\mathcal{P}$) is the input cloud to the service
  - *float32* ***deepThreshold***: ($\bar{d}$) is the input parameter (in meters) to the service. Default value will be considered if less than zero.
- **Outputs**
  - *sensor_msgs/PointCloud2* ***cloudClose***: ($\mathcal{P}^{<}$) are the points that have a z coordinates less than the threshold $\bar{d}$.
  - *sensor_msgs/PointCloud2* ***cloudFar***: ($\mathcal{P}^{>}$) are the points that have a z coordinates grater than the threshold $\bar{d}$.
  - *float32* ***usedDeepThreshold***: ($\bar{d}$) is the used threshold value. It can be different from the input value if the default limit has been chosen; so if it was less than zero.

[6] If $\bar{d} \leqslant 0$ than the default value is considered (3 meters).
Also, consider that in general all the services use this convention to set their parameter, which are also given back as output to always known which value has been used.

Algorithm 1: Computation steps for deep filtering server. Component number 2 in Figure 1

Input: $\mathcal{P}$.
Parameter: $\bar{d}$.
$\bar{d} \leftarrow$ default or incoming threshold
foreach $p_i \in \mathcal{P}$ **do**
  $z_i \leftarrow {}^{z}p_i$
  if $z_i \leqslant \bar{d}$ **then**
    $\mathcal{P}^{<} \leftarrow$ add p_i
  else
    $\mathcal{P}^{>} \leftarrow$ add p_i
return $\mathcal{P}^{<}$, $\mathcal{P}^{>}$ and $\bar{d}$.

2.2 *Support segmentation server*

This server uses RANSAC[7] algorithm to detect main planes into the cloud, with the purpose to segment all the supporting planes, i.e. tables, that are supposed to do not be one on top of the other. More in detail, it uses an iterative process to find different planes and relate

[7] plc RANSAC implementation details are available here. Further informations are available in Section 2.4.4

them with respect to the indexes of the input cloud, while discarding non horizontal planes. Now, before analysing the implementation of this service, let us consider the type of message $\mathscr{C}$ requested.

- **Inputs**
  - *sensor_msgs/PointCloud2* ***originalCloud***($\mathcal{P}$) is the input cloud to the services
  - *sensor_msgs/PointCloud2* ***originalNorms***($\mathcal{N}^{\mathcal{P}}$) is the input norm to the services[8]
  - *float32* ***minIterativeCloudPercentualSize***($\bar{s}$) specifies the percentage of number of points ($\bar{s} \in [0,1]$) with respect to $\mathcal{P}$ to stop search for plane recursively. By default, if the remaining points (q) (not already segmented as planes) are less than $(\bar{s} \cdot 100)\%$ of the points of the input cloud (n), then the server stops searching for planes.
  - *float32* ***minPlanePercentualSize***($\bar{p}$) defines the percentage of number of points ($\bar{p} \in [0,1]$) with respect to $\mathcal{P}$ to accept the plane and continuing looking for other planes. Particularly, if the detected plane has a number of inlier (m) less than $(\bar{p} \cdot 100)\%$ of the original cloud size (n), this will not be considered and the service stop looking for further planes.
  - *float32* ***maxVarianceThForHorizontal***: ($\bar{\sigma}$) represents the range in which the cross product between the normal to the detected plane and the normal to the ground (specified through $\bar{H}$) has to fall to suppose that the plane is horizontal to the ground. In other words, a plane is suppose to be a *support* if the cross product between an axis perpendicular to the ground and the normal to the plane is zero. Nevertheless, it is suppose to be true even if it falls in a range $[-\bar{\sigma}, +\bar{\sigma}]$ (so, qualitatively, $\bar{\sigma} > 0$ and small).
  - *float32[]* ***horizontalAxis***: ($\bar{H}$) indicates the normal coordinates of the ground plane. If an array with size different from 3 is given, then the default Y axis is used.
  - *float32* ***ransacMaxIterationTh***: ($\widehat{I}$) sets the maximum number of RANSAC iterations to looking for a plane into the remaining cloud $\mathcal{R}$.
  - *float32* ***ransacThDistancePointShape***: ($\widehat{d}$) sets the threshold (in meters) of distance to the plane model. If big, more separated points of different objects may be considered to be belong to a same primitive. On the other hand, if it is small different points of the same object may be considered to be belonging to different primitive.
  - *float32* ***ransacModelDistanceWeight***: ($\widehat{w}$) the relative weight (between 0 and 1) to give to the angular distance (0 to $\pi/2$) between point normals and the plane normal (the Euclidean distance will have weight $1 - \widehat{w}$).
- **Outputs**
  - *InliersSupport[]* ***supportObject***: ($\Sigma = \{\Psi_1 \dots \Psi_s\}$) the output ar-

If the incoming parameter are setted to be $\leqslant 0$ (or $\bar{H}$ with a size $\neq 3$) then default values are used considering:

$\bar{s}$	=	0.03
$\bar{p}$	=	0.03
$\bar{\sigma}$	=	0.5
$\bar{H}$	=	[0,1,0]
$\widehat{I}$	=	10
$\widehat{d}$	=	0.02
$\widehat{w}$	=	0.9

[8] a three component (n_x, n_y, n_z) vector assigned to all the points of $\mathcal{P}$, computed with the pcl API, for more details see section 2.2.1

Algorithm 2: Computation steps for supports segmentation server. Component number 3 in Figure 1

Input: $\mathcal{P}$ and $\mathcal{N}^{\mathcal{P}}$.
Parameter: $\bar{p}$, $\bar{s}$, $\bar{\sigma}$, $\bar{H}$, $\widehat{I}$, $\widehat{d}$ and $\widehat{w}$.

$\mathcal{R} \leftarrow \mathcal{P}$
$n \leftarrow$ number of points of $\mathcal{P}$
$\mathcal{N}^{\mathcal{R}} \leftarrow \mathcal{N}^{\mathcal{P}}$
${}^0\mathcal{I}_{nl} \leftarrow [0, 1, 2 \dots n]$
$f \leftarrow$ true
$\zeta \leftarrow -2$
$\iota \leftarrow 0$
while f *is true* **do**
  $\mathcal{I}_{nl} \leftarrow$ ransac($\mathcal{R}$,$\mathcal{N}^{\mathcal{R}}$,$\widehat{I}$, $\widehat{d}$,$\widehat{w}$)
  $\Omega \leftarrow$ coefficients of $\mathcal{I}_{nl}$
  $m \leftarrow$ the number of points in $\mathcal{I}_{nl}$
  $q \leftarrow$ the number of points in $\mathcal{R}$
  if $m < \bar{p} \cdot n$ **then**
    $f \leftarrow$ false
  else if $q < \bar{s} \cdot n$ **then**
    $f \leftarrow$ false
  else
    $\mathcal{S} \leftarrow$ cloud from $\mathcal{I}_{nl}$
    $\mathcal{R} \leftarrow$ remove $\mathcal{S}$ from $\mathcal{R}$
    $\mathcal{N}^{\mathcal{R}} \leftarrow$ estimate normals of $\mathcal{R}$
    if *isHorizontal*(Ω, $\bar{\sigma}$,$\bar{H}$) **then**
      ${}^0\mathcal{I}_{nl} \leftarrow$ newIdxMap(${}^0\mathcal{I}_{nl}$, $\mathcal{I}_{nl}$, ζ)
      $\mathcal{O} \leftarrow$ getOnTopCloud($\mathcal{S}$, $\mathcal{P}$)
      $\Psi_\iota \leftarrow$ store: [${}^0\mathcal{I}_{nl}$, $\mathcal{S}$, $\mathcal{O}$, Ω]
      $\iota \leftarrow \iota + 1$
      $\zeta \leftarrow \zeta - 1$
    else
      ${}^0\mathcal{I}_{nl} \leftarrow$ newIdxMap(${}^0\mathcal{I}_{nl}$, $\mathcal{I}_{nl}$, -1)
return $\Sigma = \{\Psi_1, \Psi_2, \dots, \Psi_\iota, \dots, \Psi_s\}$.

ray in which each element describes a support with several characteristics (see its type description for more details).
- *float32* ***usedMinIterativeCloudPercentualSize***: ($\bar{s}$) the used value.
- *float32* ***usedMinPlanePercentualSize***: ($\bar{p}$) the used value.
- *float32* ***usedMaxVarianceThForHorizontal***: ($+\bar{\sigma}$) the used value.
- *float32* ***usedMinVarianceThForHorizontal***: ($-\bar{\sigma}$) the used value.
- *float32[]* ***usedHorizontalAxis***: ($\bar{H}$) the used value.
- *float32* ***usedRansaxMaxIterationTh***: ($\widehat{I}$) the used value.
- *float32* ***usedRansacThDistancePointShape***: ($\widehat{d}$) the used value.
- *float32* ***usedRansacModelDistanceWeigth***: ($\widehat{w}$) the used value.

Where the *InlierSupport*, type of the elements of the outputs Ψ_ι, is a custom message that describes a support through the following features[9]

- *int32[]* ***inliers***: ($\mathcal{I}^{\Psi_\iota}$) an array containing the index of the inlier of a particular plane with respect to the original point cloud $\mathcal{P}$.
- *sensor_msgs/PointCloud2* ***planeCloud***: ($\mathcal{S}^{\Psi_\iota}$) a point cloud containing only the horizontal plane.
- *sensor_msgs/PointCloud2* ***pointOnPlaneCloud***: ($\mathcal{O}^{\Psi_\iota}$) a point cloud containing all the points belong to the objects on top of the plane $\mathcal{S}^{\Psi_\iota}$.
- *float32* ***coeff_a***: (α) the x coefficient[10] of the support $\mathcal{S}^{\Psi_\iota}$.
- *float32* ***coeff_b***: (β) the y coefficient of $\mathcal{S}^{\Psi_\iota}$.
- *float32* ***coeff_c***: (γ) the z coefficient of $\mathcal{S}^{\Psi_\iota}$.
- *float32* ***coeff_d***: (δ) the offset coefficient of $\mathcal{S}^{\Psi_\iota}$.

Algorithm 2 shows how supports are computed iteratively starting from the above inputs and parameters. In particular, given a point cloud $\mathcal{P}$ and a 3-dimensional normal vector for each points $\mathcal{N}^{\mathcal{P}}$. The algorithm use RANSAC[11] estimation to identify the inlier vector $\mathcal{I}_{nl}$ that are supposed to contains all the indexes of the points that belongs to a plane, which it is also defined through a certain set of coefficients Ω. Then, if the dimension of the segmented plane is big enough (based on $\bar{p}$) or if this happen for the size of the remaining cloud $\mathcal{R}$ (based on $\bar{s}$). Then data is saved, in order to be returned later as outputs. Finally, the algorithm iterates looking for another plane. In particular, a plane is accepted, and then saved, only if it is horizontal, namely if it has norm components parallel to the given axis $\bar{H}$[12] so:

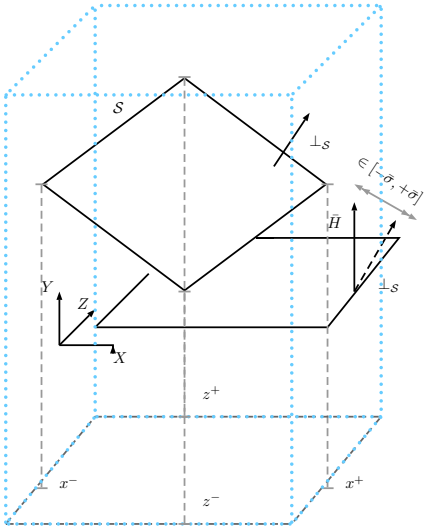
$$\mathcal{S} \text{ is horizontal } \Leftrightarrow \perp_{\mathcal{S}} \times \bar{H} \in (-\bar{\sigma}, +\bar{\sigma}) \longrightarrow 0 \tag{2}$$

where $\mathcal{S}$ represents the point cloud of the detected plane, defined by the indices $\mathcal{I}_{nl}$ (in turn described with respect to $\mathcal{R}$). While, $\perp_{\mathcal{S}}$ represent the normal to the plane, that is possible to compute from its parameters as:

$$x_{\mathcal{S}}^{\perp} = \frac{\alpha}{\Delta} \quad ; \quad y_{\mathcal{S}}^{\perp} = \frac{\beta}{\Delta} \quad ; \quad z_{\mathcal{S}}^{\perp} = \frac{\gamma}{\Delta} \tag{3}$$

where: $\Delta = \sqrt{\alpha^2 + \beta^2 + \gamma^2}$ and $\perp_{\mathcal{S}} = [x_{\mathcal{S}}^{\perp} \;\; y_{\mathcal{S}}^{\perp} \;\; z_{\mathcal{S}}^{\perp}]$.

Figure 3: Support ($\mathcal{S}$) representation with respect to ground plane (with normal $\bar{H}$). Blue box represent the area in which all the other $\mathcal{P}$ pints are represented to be belonging to the $\mathcal{O}$ extrapolated cloud.



[9] considered to be the ι-Th element of the output array Σ.

[10] For planes, RANSAC returns the coefficients of the model:

$$\alpha x + \beta y + \gamma z + \delta = 0 \tag{1}$$

as an array: $\Omega = [\alpha, \beta, \gamma, \delta]$.

[11] RANdom SAmple Consensus shape estimator from point cloud, more addressed in section 2.4.4.

[12] due to this considerations, the algorithm defines a function *isHorizzontal()* which returns true only if the table has norm parallel to the norm of the ground plane $\bar{H}$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	1	2	**-1**	**-1**	**-1**	3	4	5	**-1**	**-1**	6	7	8	**-1**	**-1**	$\mathcal{I}^{\Psi_1}$
0	**-2**	**-2**	-1	-1	-1	1	2	3	-1	-1	4	**-2**	**-2**	-1	-1	$\mathcal{I}^{\Psi_2}$
0	-2	-2	-1	-1	-1	**-3**	**-3**	**-3**	-1	-1	1	-2	-2	-1	-1	$\cdots$
-4	-2	-2	-1	-1	-1	-3	-3	-3	-1	-1	1	-2	-2	-1	-1	$\mathcal{I}^{\Psi_s}$

Table 1: index mapping. To track support points with respect to the input cloud. Example with a cloud of only 16 points where three supports have been identified as well as some non horizontal planes. In particular, the first support has 4 inlier, while the second has 3 and the fourth has only 1 inlier.

Also, let us remark that ζ is called layer of the index map and it is implemented to be efficient. Note that only from the last $\mathcal{I}^{\Psi_s}$ and the original cloud $\mathcal{P}$ it is possible to retrieve all the supports S_i. Nevertheless, this is not implemented yet and all the $\mathcal{I}^{\Psi_i}$ are processed by further algorithms.

To note that during this calculation, Algorithm 2 also manage the reduction of the remaining point on the cloud $\mathcal{R}$, while tracking the horizontal planes with respect to the indices in the original cloud $\mathcal{P}$. In order to do so, the inlier output $\mathcal{I}^{\Psi_i} \triangleq {}^0\mathcal{I}_{nl}(\zeta(\iota))$ is designed to be an array of the same dimension of the original cloud (n) such that it contains tags ζ in the same position of a point in the original cloud (^{0}). The tags have value $\zeta = -1$ for all points labelled as belonging to non horizontal planes. While, if for example $\zeta = -2$ than, all the elements in $\mathcal{I}^{\Psi_i} = -2$ represents (by their indices value into the map) the indices (of $\mathcal{P}$) of the points of the first support found by the algorithm, and so on, as it is possible to see in Table 1. Mathematically, the last support inlier set, computed by the algorithm with respect to the input cloud is:

$$\mathcal{I}^{\Psi_s} = \{\eta_j \forall j \in [0, n-1]\} \text{ where } \begin{cases} \eta_j \geqslant 0, & \text{if } \mathcal{P}_j \notin \Theta \\ \eta_j = -1, & \text{if } \mathcal{P}_j \in \Gamma \\ \eta_j = \zeta(\iota) \leqslant -2 & \text{if } \mathcal{P}_j \in \Sigma \end{cases}$$

where $\mathcal{P}_j$ is the j-Th point of the input cloud and Θ is the set of all the planes segmented by the algorithm. Moreover, it is possible to define the set of non horizontal planes founded Γ as well as the set of supports $\Sigma \equiv \{\mathcal{S}_1, \ldots, \mathcal{S}_s\}$. Hence, $\Theta \equiv \Gamma \cup \Sigma$ where $\Gamma \cap \Sigma \equiv \emptyset$.

Moreover, the Algorithm 2 extrapolates from $\mathcal{P}$ the points supposed to be belong to objects on top of the supports; those points are collected in a cloud called: $\mathcal{O}$. As it is possible to see from Figure 3, the algorithm compute the maximum and minimum coordinate x^+, x^-, z^+ and z^- to estimate a bounding box[13]. Than, $\mathcal{O}$ would simply contains the points of the original cloud $\mathcal{P}$ that falls into the bounding box. Note that, actually, the function *getOnTopCloud* takes as input the last available row of the inlier map (table 1). Important to remark it that there is not limitation on the Y axis by default since objects may have arbitrary height. Moreover, there is not even limitation on the minimum y value to be on top of a plane since it has been considered that there are no paralleled support one on top of the other; e.g. tables not shelves.

[13] note that so far the algorithm is not able to parametrize the on top method with respect to $\bar{H}$. Perhaps, x^+, x^-, z^+ and z^- should be computed in function of the given axis, or the clouds can be rotate to compensate the misalignment with Y and $\bar{H}$.

2.2.1 *Normal estimation*

The used procedure[14] to compute the normals of a surface from a cloud of points are based on [Rusu, 2009]. Which proposes it as an least-square plane fitting estimation problem[15], addressed through Principal Component Analysis. To briefly address this method let define the *covariance* matrix, for each point p_i, as:

$$C = \frac{1}{k} \sum_{i=1}^{k} ((p_1 - \bar{p}) \cdot (p_1 - \bar{p})^T) \tag{4}$$

Where k is the number of points considered to be in the neighbourhood[16] of p_i and $\bar{p}$ represents the centre of mass of the neighbourhoods. So, with respect to eigenvalue λ_j and eigenvector v_j it is

[14] Implemented by PCL, where an interesting description is available here.

[15] Note that this approach does not hold in case of multiple viewpoint.

[16] the library provides also a method to set the radius of a sphere to define the neighbourhoods of a point $\bar{p}$. This, and other used implementations, are documented here. In the following implementation let us just generically address this parameter also with the symbol $\mathcal{N}$ which is the number of points that belongs to such a neighbourhood

possible to write:

$$C \cdot v_j = \lambda_j \cdot v_j \quad ; \quad \forall j \in 1,2,3 \tag{5}$$

Anyway, this method has the problem to do not give the possibility to compute the direction of the normal of a point. Nevertheless, it can be trivially solved if the hypotheses that only one static view point v_p it is considered; in fact, to have a consistent result within this specifications, it must old that[17]: $n_i \cdot (v_p - p_i) > 0$. Note that, thanks to this relation, the direction of the normal can be unambiguously chosen[18]. Moreover, consider that when the eigenvalues of C are computed it also possible to derive the *surface curvature* through the relation:

$$\Xi = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2} \tag{6}$$

Practically, when the neighbourhoods of a point as been fixed the above considerations can be used to derive both the direction of the normal of its centroid as well as the curvature of the relative part of the surface. On the other hand, the magnitude of the normal vector, expressed in coordinates $\{n_x \; n_y \; n_z\}$, associate to a point can be estimate by deriving the coefficients $\{\alpha \; \beta \; \gamma \; \sigma\}$ (as seen in Equation 3) of the plane that best fits on the neighbourhoods points.

Moreover, thanks to the above considerations, let us remark that this normal estimation is based on a parameter: the size of the neighbourhoods k. This parameter is important for the accuracy of system since it is relate to the level of details in the cloud that have to be processed. In fact, if the neighbourhood is small it will be possible to appreciate also small details, but this is true also for the noise that would effecting more the system. On the other hand, big neighbourhood may eliminate details and noise, but the results may be also negatively effected by merging points that are belonging to completely different shapes. Last but not the least, this effects also the efficiency, since there are more points to be processed. Though experiments this parameter has been set for this case study (as seen in Section 2.5) but, with showing purposes it is possible to see the result of different value in Figures 4, 5 and 6. Moreover, from them it is also possible to remark that the output of the normal estimator is a particular type of point cloud, with the same size of the input, which relate a tree dimensional vector for all the points where coordinates are not of the point any more but of the normal vector applied to it.

[17] where all variables are defined in terms of spatial coordinates. Namely they are arrays of three dimensions.

[18] in case in which it is negative the direction of the normal vector would be just flipped.

Figure 4: The real image acquired from the Kinect RGB camera.

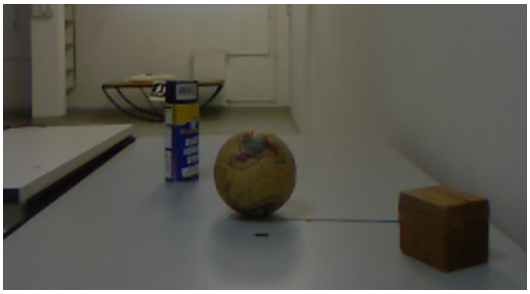


Figure 5: the original cloud computed with normal neighbourhoods of size $k = 5$ points and leaf size of 0.012 meters.

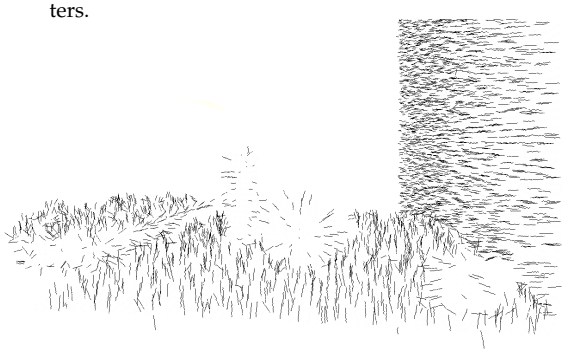
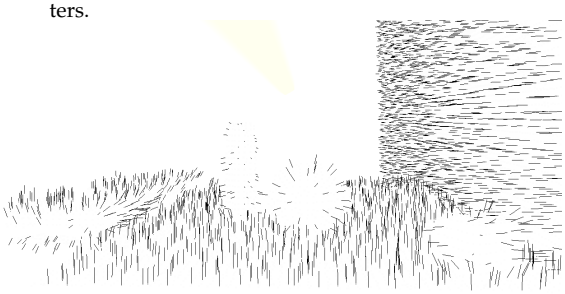


Figure 6: the original cloud computed with normal neighbourhoods of size $k = 50$ points and leaf size of 0.012 meters.



2.2.2 *Down Sampling mechanism*

Moreover, consider that the raw deep input cloud from the Kinect experiences a *Voxel Grid down-sampling*[Whelan et al., 2012] preprocessing[19]. This is done both for efficiency and accuracy reasons, since this process is aimed in reduce the number of points of a cloud. This in turns would reducing the noise, thanks to an averaging mechanism, as well as reducing the computation time of all the system, since less points have to be considered.

[19] This has been implemented though PCL, specially using a particular class.

From a theoretical point of view, the basic idea is to divide the cloud in several parallelepipeds of a constant size. More generically, the points are divided in as much leafs as needed to fulfil all the structure, where the size of the leafs are specified by three parameters: l_x, l_y and l_z[20]. Then, the down sampling mechanism is reduced to find the centroid between the points into a leaf, for all the leafs. Finally, base on that a new cloud must be created with only the computed centroids. Nevertheless, it is important to remark that also those parameters are important and it must be set with respect to the application, similarly to the previous paragraph.

[20] in the following implementation a cube is considered (so $l_x = l_y = l_z \triangleq \bar{j}$). Particularly, those parameters are recalled in Section 2.5.

2.3 *cluster segmentation server*

When the supports have been identified and the *On Top* point cloud is derived, all the points that identify the objects on a support are collected in an unique structure. To separate the objects in small point clouds a cluster server has been implemented. It is in charge to separate the points belong to different objects (contained on the *On Top* cloud) and returns an independent cloud for each object (as it is possible to see in Algorithm 3). Finally, it is important to note that the clustering algorithm does not perform nothing if the input cloud is too small (with respect to the following parameters).

Let know analyse in detail the shape of the message $\mathscr{D}$ shown in Figure 1.

- **Inputs**
  - *sensor_msgs/PointCloud2* ***cloud***: ($\mathcal{P}$) the input cloud to the service
  - *float64* ***clusterTolerance***: ($\bar{\tau}$) the radius (in meters) of the euclidean distance to consider two points to belong to the same object (expressed in meters). If this parameter it is too small an single object may results in multiple clusters; on the other hand, if is it too big different objects may be clustered in a single structure.
  - *float64* ***minClusterSizeRate***: ($\bar{w_-}$) it describes the minimum number of points that can constitute a cluster. It is described as a percentage of points, with respect to the input cloud $\mathcal{P}$.
  - *float64* ***maxClusterSizeRate***: ($\bar{w_+}$) it describes the maximum number of points that can constitute a cluster. It is described as a percentage of points, with respect to the input cloud $\mathcal{P}$.
  - *float64* ***minInputSize***: ($\bar{m}$) describes the absolute minimum number of points that the input cloud $\mathcal{P}$ has to have in order to run the clustering method. Otherwise the will be no returning cluster clouds.
- **Outputs**
  - *InliersCluster[]* ***clusterObjs***: ($\Delta = \{\Pi_1, \ldots, \Pi_c\}$) the output of the server.
  - *float64* ***usedClusterTolerance***: ($\bar{\tau}$) the used parameter. It can be different from the input id it is < 0, in this case default value is

2.3.3 *Clusterize apprauch*

The implemented algorithm uses Euclidean Cluster Extraction [Rusu, 2009] through PCL API. Which uses a classification based on testing if a point is within a sphere, of given radius τ, from the neighborhood points; with the purpose to build an octree representation that divides points belong to different objects. So, practically, given a point p_i, if there are other point p_j within a sphere centered in p_i with radios τ, then the leaf of the octree increases its volume, in order to include also p_j, and this test is repeated for all the points. When the iterations finish every cell of the octree would represent a cloud that contains a single object. Finally, note that this approach result to be very efficient in terms of computation time (as it is possible to see from Section 4.3).

Default value are used if the respective incoming parameter is < 0.

$\bar{\tau}$	=	0.03
$\bar{w_-}$	=	0.1
$\bar{w_+}$	=	1.0
$\bar{m}$	=	15

Algorithm 3: Computation steps for cluster segmentation server. Component number 4 in Figure 1

Input: $\mathcal{P}$.
Parameter: $\bar{\tau}$, $\bar{w_+}$, $\bar{w_-}$.

$\bar{\tau} \leftarrow$ default or incoming threshold
$n \leftarrow$ the number of points $\in \mathcal{P}$
if $n \geqslant \bar{m}$ **then**
  $w_+ \leftarrow n \cdot \bar{w_+}$
  $w_- \leftarrow n \cdot \bar{w_-}$
  $\eth \leftarrow$ EuclideanClusterise($\mathcal{P}$, $\bar{\tau}$, w_+, w_-)
  foreach $c_\ell \in \eth$ **do**
    $\mathcal{I} \leftarrow$ inlier of c_ℓ w.r.t. $\mathcal{P}$
    $\mathcal{C} \leftarrow$ create new cloud for cluster c_ℓ
    $\Pi_\ell \leftarrow$ store $\mathcal{I}$ and $\mathcal{C}$
  return $\Pi_1, \Pi_2, \ldots, \Pi_\ell, \ldots, \Pi_c$.
return $\emptyset$.

used and returned through this parameter.
- *float64* ***usedMinClusterSizeRate***: ($\bar{w_-}$) the used parameter.
- *float64* ***usedMaxClusterSizeRate***: ($\bar{w_+}$) the used parameter.
- *float64* ***usedMinInputSize***: ($\bar{m}$) the used parameter.

Moreover, the generic output object of the service, Π_ℓ, is of the type *InliersCluster* defined as:

- *sensor_msgs/PointCloud2* ***cloud***: ($\mathcal{C}^{\Pi_\ell}$) the cluster, that represents an object, as an independent point cloud.
- *int32[]* ***inliers***: ($\mathcal{I}^{\Pi_\ell}$) the point belong to the i-Th cluster, expressed in terms of index with respect to the input $\mathcal{P}$.

2.4 *Primitive segmentation server*

The primitive segmentation server is a fuzzy server that describe generically the remaining services shown in Figure 1 . Those services are aimed to look for a particular primitive into the input cloud; by using the RANSAC method[21]. In particular, those services are designed to process a cluster that, preferably, contains a single object.[22]

More specifically, the considered primitive shapes are: *Plane*, *Sphere*, *Cone* and *Cylinder*. At every input scan, each of them are guessed and the segmented sub clouds tested against each other, in order to identify the most possible primitive for that cluster. More practically, such sub clouds that defines a particular primitive into a class, have been described through an inlier vector of points and a set of coefficient $\Bbbk \equiv \{\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta\}$[23]. Follows the definition of such coefficients for each primitive[24]:

[25] We already sow in Equation 1 how a *Plane* is parametrised with respect to $\Bbbk$ coefficients. Also note that other elements of $\Bbbk$ are just not used.

[26] The parameters of a *Sphere* are described through the equation:

$$(x-\alpha)^2 + (y-\beta)^2 + (z-\gamma)^2 = \delta^2 \tag{7}$$

that describes all points of a sphere of radios δ. Also, note that this RANSAC implementation provides the centre of the sphere through the coefficients: $\{\alpha, \beta, \gamma\}$. Also in this case other $\Bbbk$ elements are not used.

[27] The description of the *cone* is based on the description of the vertex coordinate $\{\alpha,\ \beta,\ \gamma\}$ (for X, Y, Z components respectively), as well as, the coordinate of a point of the axis (through which it is possible to describe its direction) $\{\delta,\ \epsilon,\ \zeta\}$ and finally the open angle η.

[28] The parameters of a *Cylinder* are described through the coordinate of a point belong to the axis of the cylinder $\{\alpha,\ \beta,\ \gamma\}$ as well as another point coordinate, thanks to which define the axis direction $\{\delta,\ \epsilon,\ \zeta\}$ and, finally, its radios η

Moreover, as for the other implementations, let now analyse the type of the exchange message $\mathscr{E}$ between two servers, when a primitive segmentation server (sketched in Algorithm 4) is involved.

Algorithm 4: Computation steps for the primitive segmentation servers. Components number 5, 6, 7, 8 in Figure 1

Input: $\mathcal{P}$ and $\mathcal{N}^{\mathcal{P}}$.
Parameter: $\widehat{w}$, $\widehat{d}$, $\widehat{I}$, $\widehat{\epsilon}$, $\widehat{\rho_-}$, $\widehat{\rho_+}$, $\widehat{\alpha_-}$ and $\widehat{\alpha_+}$.
1 set input parameter or use default value
2 create model $\mathscr{M}$ for RANSAC primitive segmentation
3 $\mathscr{M} \leftarrow$ set necessary parameters $\widehat{w}$, $\widehat{d}$, $\widehat{I}$, $\widehat{\epsilon}$, $\widehat{\rho_-}$, $\widehat{\rho_+}$, $\widehat{\alpha_-}$ and $\widehat{\alpha_+}$
4 $\mathcal{W} \leftarrow$ RANSAC($\mathcal{P}$, $\mathcal{N}^{\mathcal{P}}$, $\mathscr{M}$)
5 $\mathcal{I} \leftarrow$ index of points $\in \mathcal{W}$ with respect to $\mathcal{P}$
6 $\Bbbk \leftarrow$ coefficients of $\mathcal{W}$
7 **return** $\mathcal{I}$ *and* $\Bbbk$

[21] more in detail addressed in 2.4.4.

[22] In fact, note that so far this implementation guesses only one primitive (the first detected by RANSAC) for each input clusters, so it is not able yet to segment composition of primitives.

[23] in the code an array of coefficient of size 6.

[24] as well as the value of the parameters (introduced in the above list 2.4) of the primitive server, for each shapes.

[25] Plane $(\widehat{P})$

$\widehat{w}$	=	0.0010	$\widehat{d}$	=	0.0070
$\widehat{I}$	=	1000	$\widehat{\epsilon}$	=	0
$\widehat{\alpha_-}$	=	0	$\widehat{\alpha_+}$	=	10.0

[26] Sphere $(\widehat{S})$

$\widehat{w}$	=	0.0010	$\widehat{d}$	=	0.0070
$\widehat{I}$	=	1000	$\widehat{\epsilon}$	=	0.5
$\widehat{\rho_-}$	=	0.005	$\widehat{\rho_+}$	=	0.500
$\widehat{\alpha_-}$	=	100.0	$\widehat{\alpha_+}$	=	180.0

[27] Cone $(\widehat{C})$

$\widehat{w}$	=	0.0006	$\widehat{d}$	=	0.0065
$\widehat{I}$	=	1000	$\widehat{\epsilon}$	=	0.4
$\widehat{\rho_-}$	=	0.001	$\widehat{\rho_+}$	=	0.500
$\widehat{\alpha_-}$	=	10,0	$\widehat{\alpha_+}$	=	170.0

[28] Cylinder $(\widehat{R})$

$\widehat{w}$	=	0.0010	$\widehat{d}$	=	0.0070
$\widehat{I}$	=	1000	$\widehat{\epsilon}$	=	0
$\widehat{\rho_-}$	=	0.005	$\widehat{\rho_+}$	=	0.500
$\widehat{\alpha_-}$	=	50.0	$\widehat{\alpha_+}$	=	180.0

2.4.4 *RANSAC segmentation approach*

The RANdom SAmple Consensus (RANSAC) algorithm [Schnabel et al., 2007] is a very efficient e robust implementation that randomly evaluate a model against noisy point cloud data. In particular it uses a recursively approach to sample some key point (with respect to the specified model: plane, cone, cylinder, sphere but also line and torus). Than, those key points are tested against the others, in order to compute a score of the fitted shape and guess the primitive coefficients end inlier.

The efficiency of this algorithm is strongly effected by the sampling strategies. There are several mechanism but most of them are based on octree decomposition from which randomly chose the level of the cells and retrieve samples within it. This is base on the fact that points belonging to a primitive shapes are relatively close between each other.

Also, consider that this algorithm is not deterministic and it could not end for geometrical reasons but only from the limit on the maximum number of iterations. Moreover, it is important to remark that it is not know a priory the order in which shapes would be segmented; this may create issues for debugging and evaluating procedures. Nevertheless, note that it is very probable that the algorithm finds the primitive with the highest number of inlier first, then the one smaller and so on.

In particular the used pcl implementation requires an input cloud and its norms to return a set of inlier of the primitive and the identified coefficients. Moreover, it needs also parameters (shown in the List (2.4) on this page) that are strongly depending from the objects and the environment of an application.

This is a strong limitation of this algorithm since, for every application, all the listed parameters must be setted and so it is difficult to find a generalization for an arbitrary application. Nevertheless, if RANSAC is applied to cloud that only contain an object it is possible to relax the parameter thresholds in order to fit a bigger case study. On the other hand this improve also the computation time with respect to the case in which all the data set is used to detached primitive. Anyway, this improvement is small with respect to the previous benefits in terms of generality.

- **Inputs**
  - *sensor_msgs/PointCloud2* ***cloud***: ($\mathcal{P}$) is the input cloud to the server. It is suppose to contain a single object representation.
  - *sensor_msgs/PointCloud2* ***normals***: ($\mathcal{N}^{\mathcal{P}}$) is the cloud describing the normal three dimensional vectors compute for all points of $\mathcal{P}$, (Section 2.2.1).
  - *float64* ***normalDistanceWeight***: ($\widehat{w}$) defines the relative influence ($\widehat{w} \in [0, 1]$) of the normals to the surface (within an angle of $[0, \pi/2]$) to concur for model evaluation.
  - *float64* ***distanceThreshold***: ($\widehat{d}$) determinates how close (in meters) a point must be to the model for be considered an inlier.
  - *int64* ***maxIterations***: ($\widehat{I}$) describes the maximum number of iterations of the RANSAC segmentation which limited the possibility to further test an hypothesis against data. It may strongly affect the service accuracy, since it is considered to segment only the first RANSAC solution, not looking for further tentative.
  - *float64* ***minRadiusLimit***: ($\widehat{\rho_{-}}$) describes the minimum allowable radius limits (in rad) for the primitive model; applicable to models that estimate a radius.
  - *float64* ***maxRadiusLimit***: ($\widehat{\rho_{+}}$) describes the maximum allowable radius limits (in rad) for the primitive model; applicable to models that estimate a radius.
  - *float64* ***epsAngleTh***: ($\widehat{\epsilon}$) is the maximum threshold angle (in rad) between the normal to the model and to the surface that makes a point acceptable. This is usually used to segment models perpendicular or parallel to an axis $\bar{H}$, specified by the setAxis pcl method. Note that this is not implemented since no given axes are available. Perhaps it can be removed.
  - *float64* ***minOpeningAngleDegree***: ($\widehat{\alpha_{-}}$) set the minimum opening angle (in grad). Used only for the cone segmentation server.
  - *float64* ***maxOpeningAngleDegree***: ($\widehat{\alpha_{+}}$) set the maximum opening angle (in grad). Used only for the cone segmentation server.
- **Outputs**
  - *int32[]* ***inliers***: ($\mathcal{I}$) the first set of points, detected from RANSAC, to be belonging to the relative primitive model ($\widehat{P}$, $\widehat{S}$, $\widehat{C}$ or $\widehat{R}$). Such points are described through an array of indices that refer directly to the samples of the input cloud $\mathcal{P}$.
  - *float32[]* ***coefficients***: ($\mathbb{k}$) the already analysed (in paragraphs 25, 26, 27, 28) vector of coefficients.
  - *float64* ***usedNormalDistanceWeight***: ($\widehat{w}$) the value actually used in the server. It is different from the input parameters only if a number less than zero is given. If such a case occurs default parameter are used, as specified in notes 25, 26, 27, 28, where not listed parameters for each model are considered to do not be used.
  - *float64* ***usedDistanceThreshold***: ($\widehat{d}$) same as the above item (2.4).

- *int64* ***usedMaxIterations***: ($\widehat{I}$) same as the above item (2.4).
- *float64* ***usedMinRadiusLimit***: ($\widehat{\rho_-}$) same as the item (2.4).
- *float64* ***usedMaxRadiusLimit***: ($\widehat{\rho_+}$) same as the item (2.4).
- *float64* ***usedEpsAngleTh***: ($\widehat{\epsilon}$) same as the item (2.4 of this list).
- *float64* ***usedMinOpeningAngleDegree*** ($\widehat{\alpha_-}$) same as the item (2.4 of this list).
- *float64* ***usedMaxOpeningAngleDegree***: ($\widehat{\alpha_+}$) same as the item (2.4 of this list).

2.5 *Point cloud Evaluator*

The *PointCloudEvaluator* is the only node of the system and it is in charge to guess the primitive of the objects posed in horizontal supports. In order to do that, it uses the above services (through theirs messages shown above) in a looping manner. In particular it is designed to listen the standard point cloud topic and process particular steps when a new cloud is available. Anyway, the buffer of its topics is of size 1, for memory issues. This means that the node does its best (basically never stops between loops since is late), but data published while it is computing are definitely lost.

Moreover, note (in Figure 1) that the type of the output of this algorithm as not been defined yet, so its purposes so far is only to visualize data on screen and log results for developing purposes. On the other hand, the input of the note is well defined and has been already cited in Section 2 and Note 4.

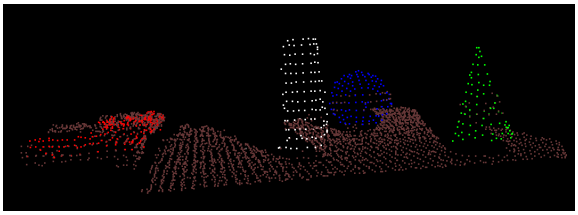
Furthermore, this node contains the computational steps listed in Algorithm 5 and also it introduces some other parameters[29] to the overall system. In particular those are:[30]

- *MIN_POINT_IN_ORIGINAL_CLOUD*: ($\bar{n}$) defines the minimum number of points such that an input cloud ${}^0\mathcal{P}$, which is considered to be already down sampled (as seen in Section 2.2.2), is discarded and the algorithm not performed.
- *DOWN_SAMPLING_RATE*: ($\bar{j}$) defines the leaf size (Note 20) for the down sampling procedure (in meters), seen in Section 2.2.2.
- *ESTIMATE_NORMAL_SPAN*: ($\bar{\mathcal{N}}$) defines the parameter (Note 16) to consider the neighbourhoods to compute the norms, as seen in Section 2.2.1. This is expressed as the number of points to defines the neighbourhoods.
- *CONE_TO_CYLINDER_PRIORITY*: ($\widehat{C}\bar{p}_{\widehat{R}}$) defines the penalty (in terms of percentage of inlier points) of a cylinder with respect to a cone. In particular, this means that for choose which primitive best fit an object, the number of inlier of a cone are more important than the number of inlier of a cylinder. So, practically, since a cone is lucky to be detected as a cylinder, this parameters make more possible to choose a cone with respect to a cylinder; even if the number of inlier of both segmentations are the same.

Moreover, from Algorithm 5 it is possible to see how the best primitive is chosen based on the number of inlier that are segmented from

Figure 7: A real case scene.

Figure 8: The segmented primitive. Supports are shown as brown, while planes are red, cone green, sphere blue and cylinder white. Note that in this case the wall has been removed so, only two different planes are considered to be supports.



[29] Of course the algorithm propagate the definition of all the parameters seen on the above services. Those are not considered in detail in this section but remark that every call may used different values. Nevertheless, in this report are used only the defaults values for all the parameters, specifically described in tables on 6, 2.2, 2.3, 25, 26, 27, 28 and 30.

[30] by default set to:

$\bar{n}$	=	100	$\bar{j}$	=	0.015
$\bar{\mathcal{N}}$	=	20	$\widehat{C}\bar{p}_{\widehat{R}}$	=	0.9

The implementation relies on a class: *PCManager* which collects common procedures and acts as a static library. Mainly, it is in focused in Point Cloud Library interface. Finally, consider that the class *PCPrimitive* is not used any more.

Since this node as the main purpose to log results other useful parameters has been defined as well:

- to filter data.
  - Int: *primitive*_MIN_INLIERS, where a variable is dedicated for all shape ($\widehat{P}$, $\widehat{S}$, $\widehat{C}$, $\widehat{R}$). If the number of the point of a cluster is less than this threshold, then the relative primitive server is not called and the number of inlier is 0). By default it is set to 40 points for all shapes.
- to show point on screen.
  - Bool: *SHOW_ORIGINAL_CLOUD*. Point colour: [grey]
  - Bool: *SHOW_SUPPORTS*. Point colour: [random]
  - Bool: *SHOW_OBJECT_ON_SUPPORT*. Point colour: [brown]
  - Bool: *SHOW_PRIMITIVE*, and clusters. Point colour: [Plane=red], [Sphere=blue], [cone=green], [cylinder=white], [unknown=dark grey].
- to log results on screen and on a csv textual file.
  - Bool: *CREATE_TXT_FOR_TEST*.
  - String: *FILE_PATH*.
  - Bool: *WRITE_ON_CONSOLE*.
- to perform experiments
  - Int: *TEST_COUNT_LIMIT*.
  - Int: *REAL_PRIMITIVE_TAG*, which must be one of the values: unknown(0), plane(1), sphere(2), cone(3) and cylinder(4).

Algorithm 5: Computation steps for the point cloud evaluator node. Component number 9 in Figure 1.

Input: ${}^0\mathcal{P}$.
Parameter: $\bar{n}$, $\bar{j}$, $\bar{\mathcal{N}}$ and ${}_{\widehat{C}}\bar{p}_{\widehat{R}}$.
${}^0\mathcal{P} \leftarrow$ get point cloud from ROS topic
${}^0\mathcal{P} \leftarrow$ downSampling(${}^0\mathcal{P}$, $\bar{j}$)
${}^d\mathcal{P} \leftarrow$ **callDeepFilter**(${}^0\mathcal{P}$, $\bar{d}$)
$n \leftarrow$ number of points of ${}^d\mathcal{P}$
if $(n > \bar{n})$ **then**
  ${\mathcal{N}}^{{}^d\mathcal{P}} \leftarrow$ estimateNormal(${}^d\mathcal{P}$, $\bar{\mathcal{N}}$)
  $[\Psi_1 \dots \Psi_s] \leftarrow$ **callSupportServer**(${}^d\mathcal{P}$,${\mathcal{N}}^{{}^d\mathcal{P}}$, $\bar{p}$,$\bar{s}$, $\bar{\sigma}$,$\bar{H}$, $\widehat{l}$,$\widehat{d}$,$\widehat{w}$)
  foreach $\iota \in [1, s]$ **do**
    $\mathcal{O} \leftarrow \mathcal{O}^{\Psi_\iota}$
    $[\Pi_1 \dots \Pi_c] \leftarrow$ **callClusterServer**($\mathcal{O}$, τ,$\bar{w_+}$,$\bar{w_-}$,$\bar{m}$)
    log *supports* and *cluster* data and computation time.
    foreach $\ell \in [1, c]$ **do**
      $\mathcal{C} \leftarrow \mathcal{C}^{\Pi_\ell}$
      $\mathcal{N}^{\mathcal{C}} \leftarrow$ estimateNormal($\mathcal{C}$, $\bar{\mathcal{N}}$)
      ${}_{\widehat{P}}\mathcal{I}^{\mathcal{C}} \leftarrow$ **callPlaneServer**($\mathcal{C}$,$\mathcal{N}^{\mathcal{C}}$, $\widehat{w}$, $\widehat{d}$, $\widehat{I}$, $\widehat{\epsilon}$, $\widehat{\rho_-}$,$\widehat{\rho_+}$, $\widehat{\alpha_-}$,$\widehat{\alpha_+}$)
      $n_{\widehat{P}} \leftarrow$ get the number of points of ${}_{\widehat{P}}\mathcal{I}^{\mathcal{C}}$
      $\Omega_{\widehat{P}} \leftarrow$ coefficients of ${}_{\widehat{P}}\mathcal{I}^{\mathcal{C}}$
      ${}_{\widehat{S}}\mathcal{I}^{\mathcal{C}} \leftarrow$ **callSphereServer**($\mathcal{C}$,$\mathcal{N}^{\mathcal{C}}$, $\widehat{w}$, $\widehat{d}$, $\widehat{I}$,$\widehat{\epsilon}$, $\widehat{\rho_-}$,$\widehat{\rho_+}$, $\widehat{\alpha_-}$,$\widehat{\alpha_+}$)
      $n_{\widehat{S}} \leftarrow$ get the number of points of ${}_{\widehat{S}}\mathcal{I}^{\mathcal{C}}$
      $\Omega_{\widehat{S}} \leftarrow$ coefficients of ${}_{\widehat{S}}\mathcal{I}^{\mathcal{C}}$
      ${}_{\widehat{C}}\mathcal{I}^{\mathcal{C}} \leftarrow$ **callConeServer**($\mathcal{C}$,$\mathcal{N}^{\mathcal{C}}$, $\widehat{w}$, $\widehat{d}$, $\widehat{I}$, $\widehat{\epsilon}$, $\widehat{\rho_-}$,$\widehat{\rho_+}$, $\widehat{\alpha_-}$,$\widehat{\alpha_+}$)
      $n_{\widehat{C}} \leftarrow$ get the number of points of ${}_{\widehat{C}}\mathcal{I}^{\mathcal{C}}$
      $\Omega_{\widehat{C}} \leftarrow$ coefficients of ${}_{\widehat{C}}\mathcal{I}^{\mathcal{C}}$
      ${}_{\widehat{R}}\mathcal{I}^{\mathcal{C}} \leftarrow$ **callCylinderServer**($\mathcal{C}$,$\mathcal{N}^{\mathcal{C}}$, $\widehat{w}$,$\widehat{d}$,$\widehat{I}$,$\widehat{\epsilon}$,$\widehat{\rho_-}$,$\widehat{\rho_+}$, $\widehat{\alpha_-}$,$\widehat{\alpha_+}$)
      $n_{\widehat{R}} \leftarrow$ get the number of points of ${}_{\widehat{R}}\mathcal{I}^{\mathcal{C}}$
      $\Omega_{\widehat{R}} \leftarrow$ coefficients of ${}_{\widehat{R}}\mathcal{I}^{\mathcal{C}}$
      if $(n_{\widehat{P}} = 0) \wedge (n_{\widehat{S}} = 0) \wedge (n_{\widehat{C}} = 0) \wedge (n_{\widehat{R}} = 0)$ **then**
        tag: $\mathcal{C}$ as *UNKNOWN* without any coefficients.
      else if $(n_{\widehat{P}} \geqslant n_{\widehat{S}}) \wedge (n_{\widehat{P}} \geqslant n_{\widehat{C}}) \wedge (n_{\widehat{P}} \geqslant n_{\widehat{R}})$ **then**
        tag: $\mathcal{C}$ as a *PLANE* with coefficients $\Omega_{\widehat{P}}$
      else if $(n_{\widehat{S}} \geqslant n_{\widehat{P}}) \wedge (n_{\widehat{S}} \geqslant n_{\widehat{C}}) \wedge (n_{\widehat{S}} \geqslant n_{\widehat{R}})$ **then**
        tag: $\mathcal{C}$ as a *SPHERE* with coefficients $\Omega_{\widehat{S}}$
      else if $(n_{\widehat{C}} \geqslant n_{\widehat{P}}) \wedge (n_{\widehat{C}} \geqslant n_{\widehat{S}}) \wedge (n_{\widehat{C}} \geqslant {}_{\widehat{C}}\bar{p}_{\widehat{R}} \cdot n_{\widehat{R}})$ **then**
        tag: $\mathcal{C}$ as a *CONE* with coefficients $\Omega_{\widehat{C}}$
      else if $(n_{\widehat{R}} \geqslant n_{\widehat{P}}) \wedge (n_{\widehat{R}} \geqslant n_{\widehat{S}}) \wedge (n_{\widehat{R}} \geqslant n_{\widehat{C}})$ **then**
        tag: $\mathcal{C}$ as a *CYLINDER* with coefficients $\Omega_{\widehat{R}}$
      else
        tag: $\mathcal{C}$ as *UNKNOWN* without any coefficients
      log *RANSAC* coefficients, number of inlier and computation times.
return nothing.

[5].***line1***: as introduced in Section 1.

[5].***line2***: smooth cloud to remove noise and reduce the number of points, as seen in Section 2.2.2.

[5].***line3***: call to deep filter server, shown in Algorithm 1. In this algorithm the only returning value considered is $\mathcal{P}^<$

[5].***line6-13***: seen in Section 2.2.1. Note that it uses the parameter $\bar{\mathcal{N}}$.

[5].***line7***: call the support server, seen in Algorithm 2 in order to iterate over all the horizontal planes.

[5].***line9***: retrieve the cloud contains the *onTop* (Figure 3) points of a given support.

[5].***line10***: call the cluster server, seen in Algorithm 3, in order to divide the cloud of the top point to a supports in clouds containing objects.

[5].***line15***: call the Primitive server (Algorithm 4) specialised in Plane segmentation (note 25).

[5].***line18***: call the Primitive server (Algorithm 4) specialised in Sphere segmentation (note 26).

[5].***line21***: call the Primitive server (Algorithm 4) specialised in Cone segmentation (note 27).

[5].***line24***: call the Primitive server (Algorithm 4) specialised in Cylinder segmentation (note 28).

[5].***line27-29-31-33-35-37***: The selection low to detect which is the more probable primitive to guess.

[5].***line11-40***: The action of the algorithm to save result is not sketched in this algorithm. More information on Section 3.1.

[5].***line40***: This algorithm is used within ROS framework so, the algorithm restart again as soon as a new cloud is available.

RANSAC, applied to all the primitive at every cloud scan. Particularly, it is possible to see how a prioritized mechanism has been implement. Also, note that since cylinder and cone are similar shape, the algorithm results to better assess primitives if the number of inlier of the cylinder are penalized when a cone hypothesis is tested, as already seen for parameter ${}_{\widehat{C}}\bar{p}_{\widehat{R}}$. Finally, note that since the structure is of type if-else, the algorithms does not test for other guesses so, it is not possible to check if the object is composed from more than one primitive.

Finally, consider Figure 7 and 8 which show an environment and

the segmented results of the algorithm in a real case. From there, it is possible to notice how the wall as been removed, since it is not an horizontal plane, while two supports have been detected since two different planes are present in the scene. Particularly, in one support a book is placed, while on the other table there are: a cylinder and a cone. Than the system compute the clusters for all the objects and, for each of them, it computes the inlier of all the primitive models using RANSAC. Then, the low (defined in Algorithm 5) to choose the most probably primitive is applied ant the cluster cloud coloured with a particular colour for each shape. To note that the system is able to detect shapes also when they are partially occluded. Finally, consider that only a plane has been segmented from the book instead of two, due to the intrinsic implementation of this note that does not look for more than one primitive in every cluster.

Figure 9: The theoretical environmental set up for the experiments. It shows the absolute measurements as well as the Kinect coordinate frame and the point in which object will be placed proximately in the middle table.

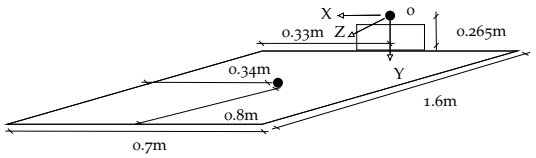


Figure 10: The real environmental set up for the experiments. Primitive objects (shown in Figure 7) will be posed on the centre of the table in correspondence to the brown sign.

3 *Experiment set-up*

The following experiment set up has been designed in order to estimate the recognition rates as well as the computational time for static primitive detection. More in detail, let us consider that, since the methods used both in Algorithm 2 and 3 does not allow to know in advance the index ι and ℓ even for a static known case[31]. Practically, this means that even if a scene is static and it is well known that contains a single support with primitive objects (or a single object with several supports) it is not trivial to test efficiently (and automatically) all the data iterating through $\iota \in [1, s]$ and $\ell \in [1, c]$. For this reasons, the experiment is design to be used against an environment having only a support and a primitive object. This reduces $s = 1$ and $c = 1$, making clear the results. This is useful to estimate quantitatively the performances of the algorithm but qualitative tests, with several objects and supports have been performed as well, as it is possible to see in Figure 8.

In particular, the environment (shown theoretically and really in figures 9 and 10 respectively) it is compose by two main planes: one is a wall while the other is a table[32]. More in detail, the table dimensions are: heigh 0.72m, length 1.6m and width 0.7m. On it, sequentially, has been performed five tests and, for each of them, data is collected as described in Section 3.1. In the first experiment the table is empty, while on the second in the table there is a book[33] with two visible planes. Than for a third experiment only a ball[34] is placed on the table, than a cone[35] and finally a cylinder[36]. All the tests are considered to old until 1000 scans are processed. For all the test the relative object is posed in the centre of the table highlighted both in figures 9 and 10. Finally, consider that tests have been performed with a laptop with 4GB of memory, and processor *i5- 460M- 2.53GHz- 2core*; where only one core it is dedicated to the ROS computation[37].

[31] since support are based on RANSAC (Section 2.4.4), in turn based on randomised approach, it is not know a priori the sequence in which planes are processed. This, inevitably, will effect the succession of results given by clustering, making ι and ℓ indeterminable with respect to the real objects. So it is impossible to relate a real object with respect to the index over time

[32] Also note that the kinect is placed over this table, on top of a box which is heigh 0.25m.

[33] plane of heigh 0.02m, length 0.245m and width 0.165m

[34] sphere with a radius of 0.065m

[35] cone with height 0.175m and a radius of 0.05m

[36] cylinder with height 0.194m and a radius of 0.03m

[37] This is done by setting the ROS_PARALLEL_JOBS variable to '-j1 -l1'.

3.1 *Log CSV structure*

To systematically evaluate experiments, the cloud evaluator node is designed to create a textual file[38]. This file is update at every scan and summarise a set of useful states of the system using a *csv* format. Particularly, initial and final time stamps in micro-seconds are logged for every service calls shown in Algorithm 4. Moreover, the system logs also data about RANSAC segmentation as the number of inlier, the coefficient as well as the selected primitive. From a practical point of view, to do so the algorithm logs two different types of messages: the first is written when the line 11 is computed and contains info about the supports and clusters identified by the services. While the other message is more focused on the performances of the primitive shape segmentation and it is appended into the file when the line 39 of Algorithm 5 is performed. Thanks to this clear sketch, it is possible to easily draw the main structure of the logging file. So, if the first type of message is called $[1]$ while the second $[2]$, for every lines of the first type there would be $s \cdot c$ messages of type $[2]$[39].

[38] with the name based on data, time and the name of the real primitive placed on the table

[39] actually messages are lines in the file that start with the symbols: [1] or [2].

Let us analyse now in detail the data collected into the logging file[40]. Particularly, its lines are in accord with the previous description and its name can be used, in the most basic case, to retrieve two different structure: **SC** and **RA**; where the first contains only messages of type $[1]$ while the second of type $[2]$. Furthermore, the support-cluster structure **SC** contains f messages[41] $\mathbf{sc}_j$, which shape is:

[40] Let us remark that the output file contains an herder which summarize all the parameters used by the system, exhaustively enough to describe the experiment from a computational point of view. So, the actual csv file starts with the line after a separator (###...).

[41] where f is the number of total scans, in this report set to 1000

$$\mathbf{sc}_j \triangleq \left[\{s\ {}^0t_s\ {}^1t_s\ n_s\} \quad \{c_1\ {}^0t_{c_1}\ {}^1t_{c_1}\}\ \{c_2\ {}^0t_{c_2}\ {}^1t_{c_2}\}\{\ldots\}\{c_s\ {}^0t_{c_s}\ {}^1t_{c_s}\}\right] \tag{8}$$

where n_s is the number of point in the input cloud to the support server. While, s represents the number of supports found by the server, while c_ι is the number of cluster segmented from the ι-Th support. Also, it is possible to point out that with the symbols ${}^0t$ and ${}^1t$ the initial and final time stamp are described respectively for each service calls. So, for example, $\left({}^1t_s - {}^0t_s\right)$ is the computational time spent identify all the supports at the j-Th[42] scan. while, $\left({}^1t_{c_3} - {}^0t_{c_3}\right)$ is the time spent to segment in clusters the points on top of the third support.

[42] Note that j is not directly linked to the index ι and ℓ, but it is just the number of line of the csv file

On the other hand, let us analyse the structure representing assessments based on RANSAC (**RA**), which contains $\left(f \cdot (s \cdot c)|_\iota^\ell\right)$ messages $\mathbf{ra}_j$ which, in turn, have the following shape.

$$\mathbf{ra}_i \triangleq \left[\{\iota\ \ell\ n_c\}\ \{n_{\hat{p}}\ {}^0t_{n_{\hat{p}}}\ {}^1t_{n_{\hat{p}}}\}\ \{n_{\hat{S}}\ {}^0t_{n_{\hat{S}}}\ {}^1t_{n_{\hat{S}}}\}\ \{n_{\hat{C}}\ {}^0t_{n_{\hat{C}}}\ {}^1t_{n_{\hat{C}}}\}\ \{n_{\hat{R}}\ {}^0t_{n_{\hat{R}}}\ {}^1t_{n_{\hat{R}}}\}\ \{\bar{\chi}\ \chi\}\ \{\alpha^*\ \beta^*\ \gamma^*\ \delta^*\ \epsilon^*\ \zeta^*\ \eta^*\}\right] \tag{9}$$

So, it contains: the indices ι and ℓ, as well as the number of the inlier ($n_{\hat{\cdot}}$) and the computation time for all primitive segmentation servers. Last but not less important, it contains also the number of points of the particular cluster analysed in this iteration: n_c; this is the size of the input cloud to the primitive services. Also, note that even in this case the time spent in computations can be relatively derived as in the previous case. So, for example: $\left({}^1t_{n_{\hat{p}}} - {}^0t_{n_{\hat{p}}}\right)$ would be the

time spent by the plane segmentation server on the ι-Th support and ℓ-Th cluster. Moreover, it contains also the tag[43] of the real primitive object on the table: χ; and the final estimation of the algorithm: $\bar{\chi}$[44]. Finally, the massage would also contains the available coefficients of the chosen primitive $\mathbb{k}$.

In order to clarify those particular logs, let us analyse the example shown in Table 2. From here it is possible to see that the structure **SC** is not a matrix, since its number of columns in not constant but equal to $(3s+4)$; where s may change for every scans since different set of supports can be found in different iterations (due to the concepts seen in Section 2.4.4). On the other hand this is not true for **RA** which is a matrix with 23 columns but, unfortunately, its number of rows it is not known a priory since the number of cluster and supports may slightly change during iterations, due to the noise.

Moreover, note that the computational time has been logged as a set of time stamp instead of a set of temporal intervals. This has been decided because such a structure allows to derive more information. For example, the overall computation time of the algorithm can be computed as the differences between the final instant of last primitive server call, with $\iota = s$ and $\ell = c$, and the initial time of the support server call. So for a generic j-Th scan it would be:

$$T_j = \left({}^0t_{\widehat{R}}(\iota = s, \ell = c) \; - \; {}^1t_s(j) \right) \tag{10}$$

Or, if also the timing to acquire a new scan[45] would be considered:

$$T_j^+ = \left({}^0t_s(j+1) \; - \; {}^0t_s(j) \right) \tag{11}$$

[43] In particular primitive tags are:

·	unknown	0
$\widehat{P}$	plane	1
$\widehat{S}$	sphere	2
$\widehat{C}$	cone	3
$\widehat{R}$	cylinder	4

[44] computed on lines 28, 30, 32, 34 or 38 of Algorithm 5.

⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
[1]	2	$\{{}^0t, {}^1t, n_s\}$	1	$\{{}^0t, {}^1t\}$	2	$\{{}^0t, {}^1t\}$	
[2]	0 0	n_c	$\{n,{}^0t,{}^1t\}_{\widehat{P}}$	$\{n,{}^0t,{}^1t\}_{\widehat{S}}$	$\{n,{}^0t,{}^1t\}_{\widehat{C}}$	$\{n,{}^0t,{}^1t\}_{\widehat{R}}$	$\{\bar{\chi},\chi\}$	$\{\mathbb{k}^*\}$
[2]	1 0	…	…	…	…	…	…	
[2]	1 1	…	…	…	…	…	…	
[1]	1	$\{{}^0t, {}^1t, n_s\}$	3	$\{{}^0t, {}^1t\}$				
[2]	0 0	n_c	$\{n,{}^0t,{}^1t\}_{\widehat{P}}$	$\{n,{}^0t,{}^1t\}_{\widehat{S}}$	$\{n,{}^0t,{}^1t\}_{\widehat{C}}$	$\{n,{}^0t,{}^1t\}_{\widehat{R}}$	$\{\bar{\chi},\chi\}$	$\{\mathbb{k}^*\}$
[2]	0 1	…	…	…	…	…	…	
[2]	0 2	…	…	…	…	…	…	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Table 2: an example of the logs. Table **SC** labelled with: [1], while the structure **RA** with the tag [2].

[45] Also, consider that it is possible to compute only the acquisition time for the j-Th scan as: $\left(T_j^+ - T_j\right)$, from which it is also possible to derive the acquisition rate.

4 *Results* (Very Draft)

Let us now analyse the the results obtained through the experiments described in Section 3. Particularly, the logged data have been analysed using MATLAB[MATLAB, 2010] in order to visualize the recognition rates for all the primitive shapes (shown in Section 4.1). Moreover, for all the services of the system[46], the computation time is proposed as well, both in terms of variance and number of points (shown in sections 4.2, 4.3, 4.4 and **??**). Finally, the Section **??** is appended in order to analyse the quality of the estimation of the primitive coefficients provided by RANSAC.

Furthermore, let us consider the dimensionality of this particular case of study. Particularly, the number of acquired points when the environment is empty (as shown in Figure 10) are 113226.6, in an average of 1000 scans. Then, thanks to the down sampling procedure (introduced in Section 2.2.2) those are limited to 8107.3 points; which is equal to the number of estimated normals vectors, as shown in Section 2.2.1.

[46] This report comes with the ROS package *RansacServiceSegmentation*, which contains the implementation of this system. Also, the package contains, in the *file* folder, the logs of the experiments as well as the MATLAB functions to compute the below results.

4.1 *Recognition Rates*

Table 3 shows the recognition rates of the primitives, organized as a confusion matrix[Stehman, 1997]. In particular, the data collected in each experiments has been shown by columns while the rows gives an direct view of the miss classification rates.

%	*Empty*	*Plane*	*Sphere*	*Cone*	*Cylinder*
Empty	87.40	11.65	0.20	1.06	1.60
Plane	12.50	82.54	6.63	7.22	8.70
Sphere	0.00	0.00	93.17	0.00	0.00
Cone	0.00	0.32	0.00	66.04	0.11
Cylinder	0.10	5.49	0.00	25.68	89.59

Table 3: the confusion matrix to show recognition and miss segmentation rates. Kinect view point placed in accord with the frame shown in Figure 9

Moreover, consider that if an error on the support server occurs (which may happen if the support is not found or if more than one are detected) then the number of total scans is not equal to the number of acquisitions (from which this table has been computed). In particular, this happened 128 times, when the support was empty, 253 times when only the book (plane) was present, 44 times for the sphere, 43 for the cone and 41 for the cylinder. Also, consider that this data comes from 1000 scans for each primitives, so the errors can be respectively expressed as the: 12.8%, 25.3%, 4.4% and 4.3%.

Furthermore, from the confusion matrix it is possible to see that there are not important miss classification between shapes. Nevertheless, it may luckily happen that a cone is misclassified as a cylinder, but it is less likely than the vice versa occurs. So it could be reasonable[47] to further tune the algorithm in order to compensate this not symmetric behavior and assure more reliable results.

[47] This has not been done yet. Further improvement of this algorithm would be to compensate those error using this assessments on Algorithm 5, especially at lines 27-29-31-33-35-37.

4.2 *Support service performances*

Figure 11: the time spent to compute the support at every scan for different object posed in the table (empty scenario is considered as well), as seen in Section 3.

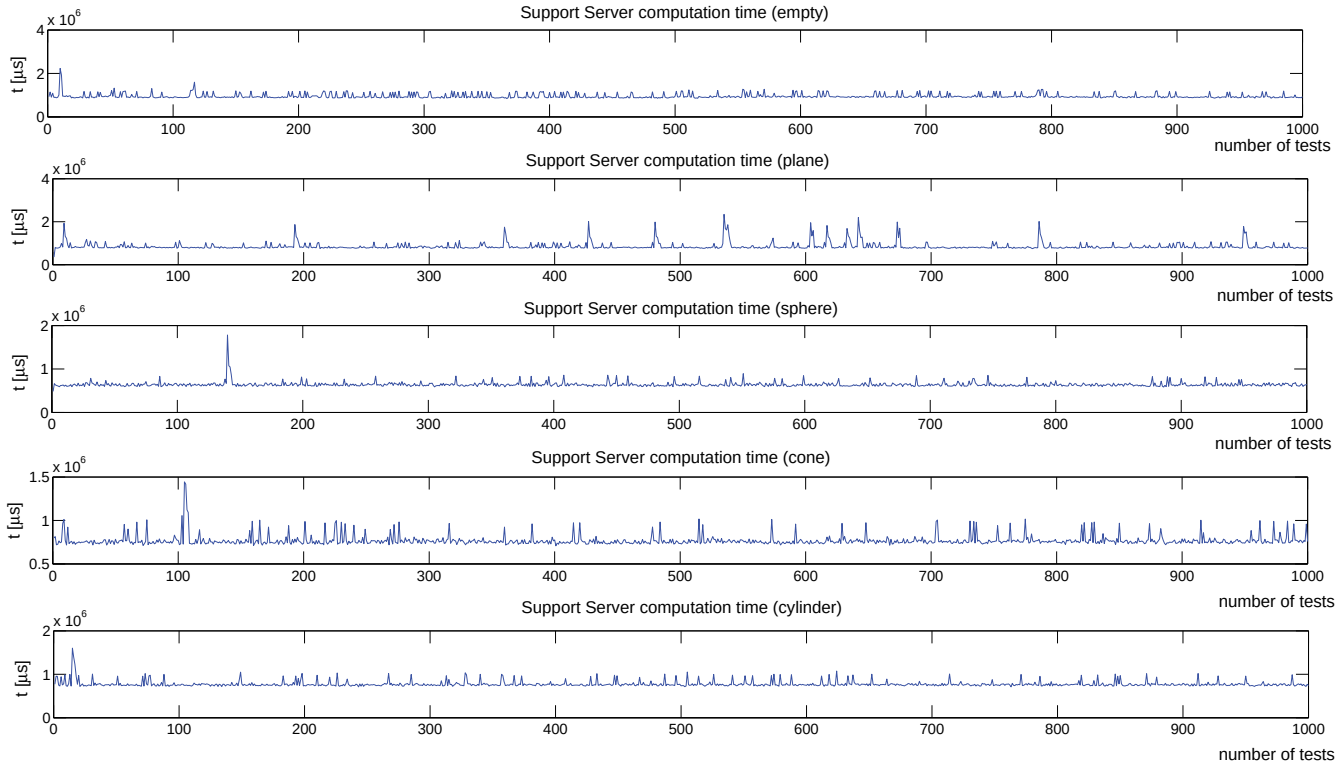
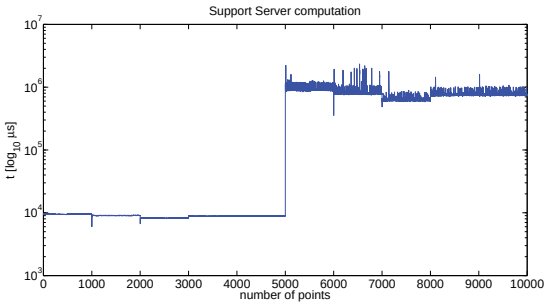


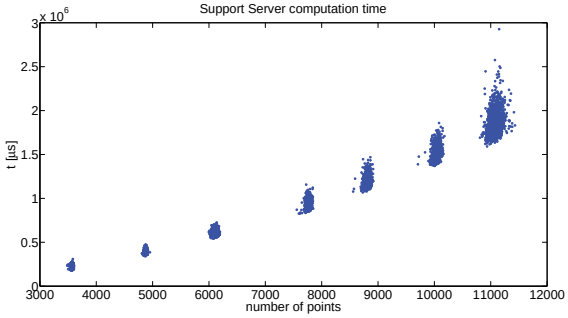
Figure 11, shows the amount of time spent to process the support server (Section 2.2) in order to segment the table and the points of the objects in top of it. From here, it is possible to consider that the time is affected by a white noise due to other laptop computations but it does not change strongly the overall behavior of the server. In particular, it is possible to appreciate that the average delay introduced by this server is between 0,6 and 0,9 seconds; which makes this server the slowest of all the system. Furthermore, it is interesting to see the distribution of the same data organized with respect to the number of points of the input cloud to the support server. In particular, this result is proposed in Figure 12 and shows the internal behaviour of the PCL API, where it is important to remark that the Y axis is in logarithmic scale and that the left part of the line it is not flat, but is affected by a small white noise. Moreover, it is interesting to notice how the computational time strongly changes at number of points equal to: 2001, 3001, 5001, 7001 and 8001 (but small changes can be seen periodically except for 4001 and 9001 points). This behaviour, seems to be linked to the resources allocated by the Point Cloud Library, but why this happen is not clear even after an open question on the official forum. Finally, it is important to consider that this behaviour holds also after test done with different laptops and environments.

Figure 12: the time spent to segment the support with respect to the number of points of the input cloud.



Moreover, Figure 13 shows the computation time spent by the support server with respect to table with different sizes. In particular, to simulate that, on top of the table of Figure 3 have been placed a vertical plane in order to hide the points of the supports behind it. Whit this set up[48] it has been possible to obtain such a graph though 8 different experiments. For each of them, still 1000 scans are acquired where, in the first experiment limits the support to $z \leqslant 0.8m$, the second to $z \leqslant 0.9m$, then $z \leqslant 1m$, $z \leqslant 1.1m$, $z \leqslant 1.2m$, $z \leqslant 1.3m$, $z \leqslant 1.4m$ and $z \leqslant 1.5m$. Particularly, each cluster (where the fourth and the fifth are so close to look the same) of Figure 13 is denoted by an experiments and the computation time is shown with respect to the number of points of the input cloud given to the support server. This results, makes clear that the computation time is not linear with respect to the increase of the number of points but, at the same time, it does not increase very rapidly. Also, from the same graph it is possible to see that the variance is limited even if it tends to increase when the number of points are increasing.

Figure 13: the time spent to segment the support with respect to the number of points of the table acquired by the Kinect.



[48] it has been also possible to measure the minimum span of places in front of the Kinect which is not read by the sensor. In particular this is described for all the points of the support that have the z coordinate less than 0.68m (with respect to figures 9 10).

4.3 *Cluster service performances*

The cluster server described in Section 2.3 is the components which requires less resources of all the system. In fact, from Figure 14, it is possible to see that the computation time for every scans is at most $1\mu s$, which is at the same order of magnitude of the time stamp resolution used to log data. So it is possible to consider the latency introduced from this server negligible with respect to the other components. Last but not the least, it is important to remark that this is

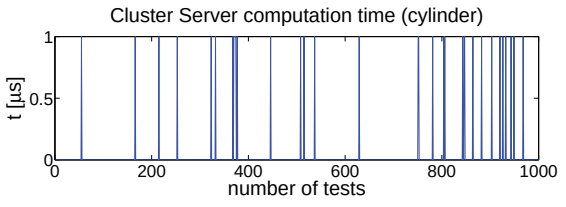


Figure 14: The computation time of the cluster server for all scans in which a cylinder was placed in the support.

true for all the shapes, in fact Figure 14 is only an example, but this behaviour is the same for all the experiments.

4.4 *Primitive service performances*

Figure 15: the time spent to compute the RANSAC models at every scan for different object posed in the table (empty scenario is considered as well), as seen in Section 3.

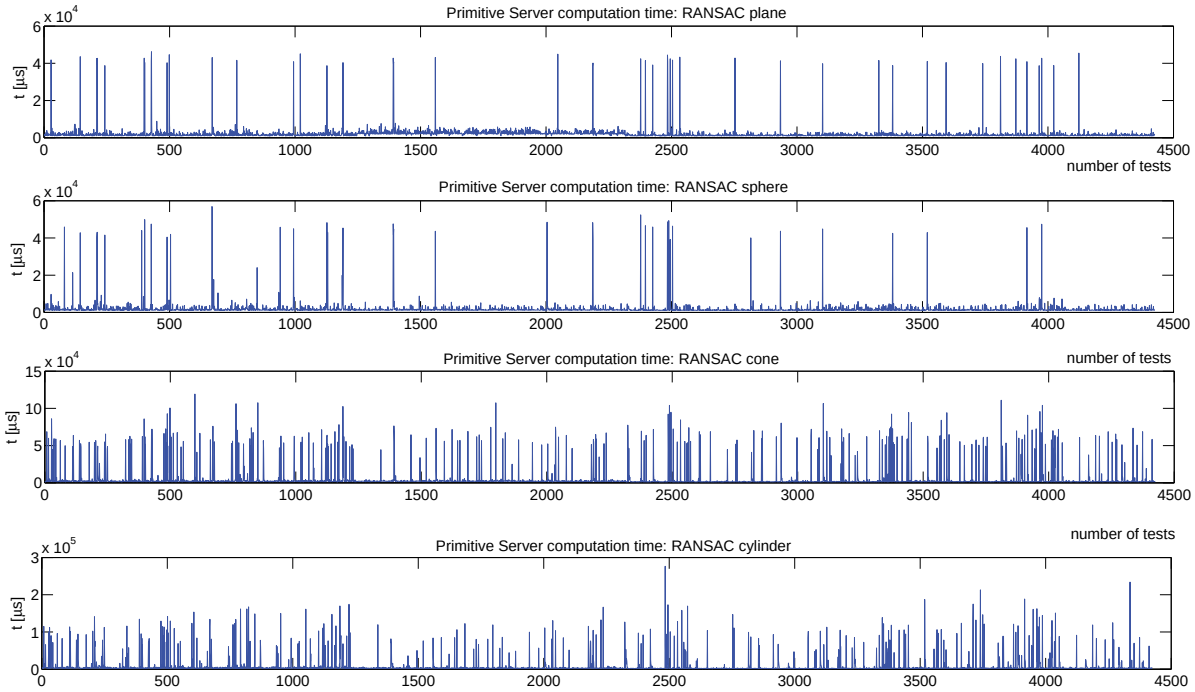


Figure 16: time spent to segment the plane w.r.t. number of points.

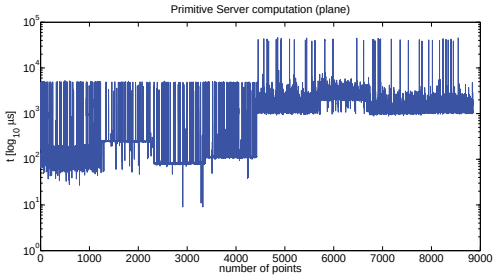


Figure 17: time spent to segment the sphere w.r.t. number of points.

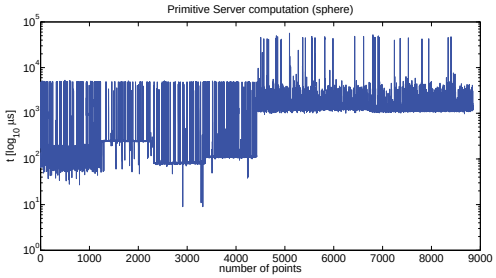


Figure 18: time spent to segment the cone w.r.t. number of points.

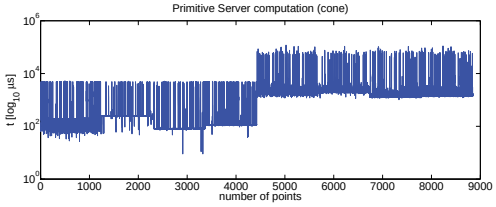


Figure 20 shows the computation time of the primitive servers with respect to the different scans in all the experiments performed for different object shapes. From here it is possible to appreciate that the temporal performances are stable and even if affected by a white noise. In particular, the computational old in average for less than 10ms with sporadic peaks which, in average, are around of 40ms. This, results makes clear that this server has good performances especially thanks to the fact that RANSAC is applied to small cluster instead of to the all original point cloud.

Furthermore, figures 16, 17, 18 and 19 present the same data of Figure 20 with respect to the number of points available in the input cloud to the primitive services. Those graphs have the same char-

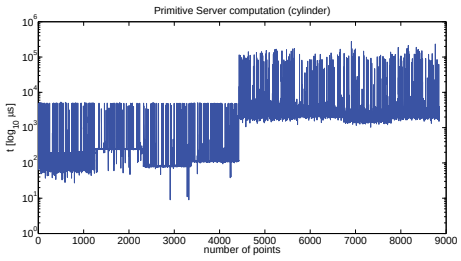
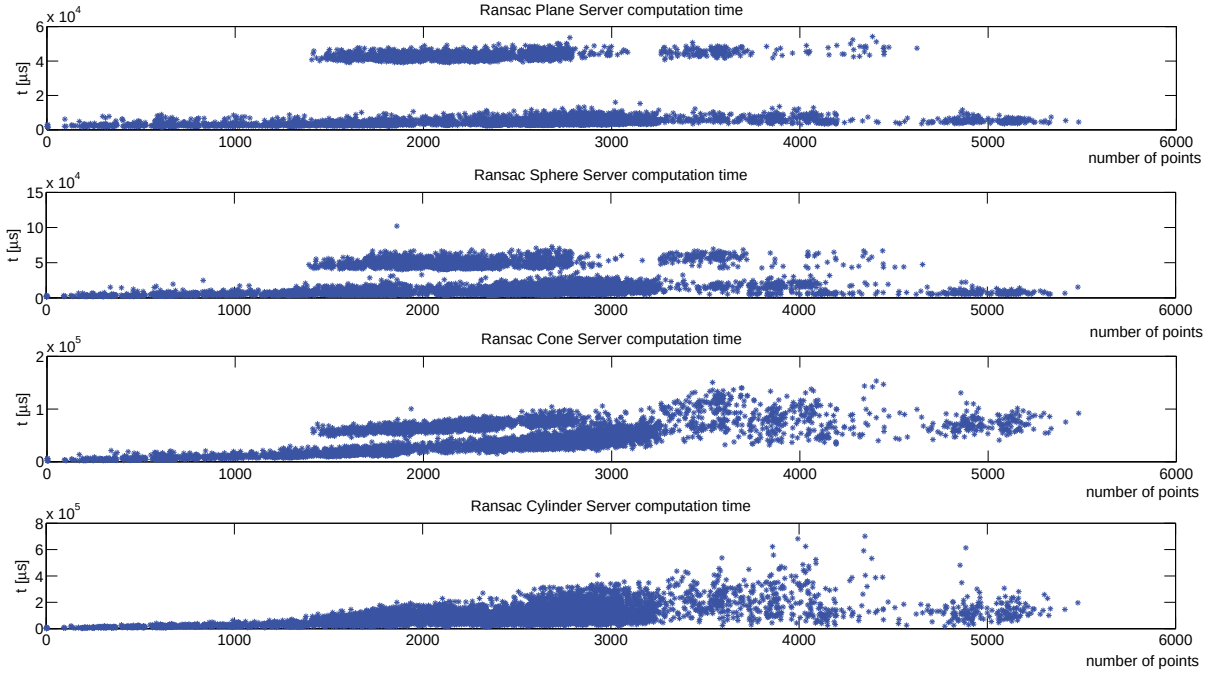


Figure 19: time spent to segment the cylinder w.r.t. number of points.

acteristics as the Figure 12 and they presents the same type of behaviour, where the critical points are for: 2346, 3365 and 4836 points, for all the shapes.

Figure 20: the time spent to compute the RANSAC models at every scan for different object posed in the table (empty scenario is considered as well), as seen in Section 3.



References

Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011. (page)

Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009. (page)

OpenNI User Guide. Openni organization, november 2010. *Last viewed*, 18:15, 2011. (page)

Radu Bogdan Rusu. *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Computer Science department, Technische Universitaet Muenchen, Germany, October 2009. (page)

Thomas Whelan, Michael Kaess, Maurice Fallon, Hordur Johannsson, John Leonard, and John McDonald. Kintinuous: Spatially extended kinectfusion. 2012. (page)

Ruwen Schnabel, Roland Wahl, and Reinhard Klein. Efficient ransac for point-cloud shape detection. In *Computer graphics forum*, volume 26, pages 214–226. Wiley Online Library, 2007. (page)

MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010. (page)

Stephen V Stehman. Selecting and interpreting measures of thematic classification accuracy. *Remote sensing of Environment*, 62(1):77–89, 1997. (page)