Laurea Magistrale in Ingegneria Informatica

A Cloud based Reinforcement Learning Framework for humanoid grasping

Laureando:
Alejandro Gatto

Relatore:
Prof. Enrico Pagello

Correlatore:
Dott.ssa Elisa Tosello

Anno Accademico 2015 - 2016

Università degli Studi di Padova
Scuola di Ingegneria

Dipartimento di Ingegneria
dell'Informazione
Laurea Magistrale in Ingegneria Informatica

A Cloud based Reinforcement Learning Framework for humanoid grasping

Laureando:
Alejandro Gatto

Relatore:
Prof. Enrico Pagello

Correlatore:
Dott.ssa Elisa Tosello

Anno Accademico 2015 - 2016

Abstract

This work presents an innovative approach to a common task on robotics: grasping a set of objects.

Grasping is one of the earliest and most common tasks developed in robotics.

Modern-day robots can be carefully hand-programmed to carry out many complex manipulation tasks. However, autonomously grasping a previously unknown object still remains a challenging problem. This Thesis presents a new framework, inspired by the classical ***sense-model-act*** architecture and the knowledge processing of ***Cognitive Robotics***. The framework tries to generalize the grasping task to a generic action, where ***Reinforcement Learning*** and ***Cloud Robotics*** play an important role.

In fact, the ***sense*** module receives the input from *RGB-D* sensors and uses state-of-art vision algorithms for segmentation and detection of objects in point clouds.

The ***model*** module takes the input from the sense node and uses a novel Reinforcement learning algorithm that improves the internal model by using the experience of past actions in a particular manner: it stores the experience information in an ontology database that can also be used by other robots performing the same task. Ontology communication is done by using modern web-application standards in order to generalize and scale better the system. The grasping action itself uses another state-of-art algorithm. It takes the input object and the gripper's geometry and it generates some ***high-level*** grasping poses. These poses are then filtered by the learning algorithm.

Finally, the ***act*** module takes the ***model***'s output and executes the motion planning algorithm to complete the task.

This architecture is very modular and can be easily adapted to perform different tasks or to execute them on other robots.

Contents

List of Figures

1 Introduction

1.1 Objectives

The main objective of this Thesis is the development of a complete pipeline that involves the classical sense-model-act architecture in which the **model** part implements a new algorithm that combines reinforcement learning and ontology reasoning. The specific task to be performed by the robot will be the grasping of a set of objects. Nevertheless the pipeline can be easily adapted to carry out many different tasks. The communication between modules will be implemented using another novel approach that involves **cloud robotics** and uses some common web application's standards like ***Json*** and ***HTTP POST calls***. Figure 1 presents a high level scheme of the pipeline. The **sense module** receives the scene's point cloud data and then it passes it to the segmentation algorithm that gets the list of valid objects. This list is passed to the **model module** that sends every object to the cloud server. The cloud server first extracts different descriptors of the object's point cloud and then it searches the object on the **RDF** database. If the object is found it also retrieves the grasping information and sends it to the **act module**. Otherwise, it passes to the Reinforcement Learning algorithm which learns to grasp the novel object. Finally with the grasping information it calls the **act module** in order to perform the grasping action on the object. For each action the **act module** returns a *reward* that completes the Reinforcement Learning's cycle.

All the experiments will be done using the Aldebaran's humanoid robot Nao.

1.2 Thesis's structure

This thesis is structured as follows:

- Introduction.
- Sense module.
- Model module.
- Act module.
- Experiments.

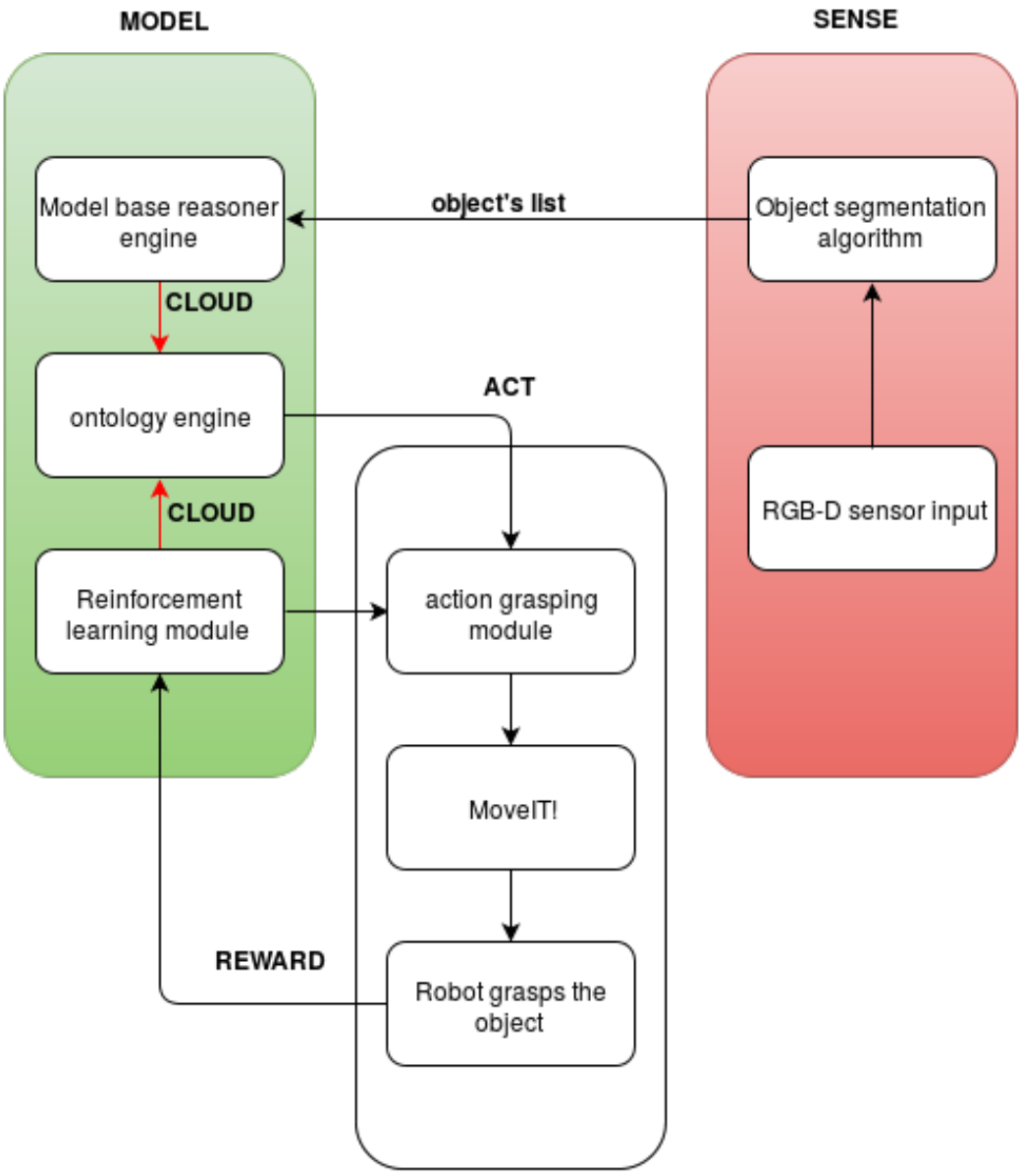


Figure 1: Pipeline's high level scheme reproducing the three components of the **sense-model-act** architecture.

- Conclusions and further work.

The whole pipeline is implemented using the ***ROS middleware*** [2], specifically the ***ROS Indigo*** version. The modules are mainly written in the ***C++*** language.

The first part presents the problem of detecting a set of single objects (or also a set of objects in a clutter) from a RGB-sensor's output. The chapter begins with the description of the ***ASUS PrimeSense***'s hardware (the RGB-D sensor used). The problem is described as a ***tabletop object detection*** problem. A section covers the segmentation algorithms used by the module and some details of the implemented code.

The second part describes the ***Model module***. This is the most important module because it contains the learning algorithm. First there's a brief introduction to Reinforcement learning and then an introduction to ontologies. Some nodes of this module run on an external server that uses the infrastructure developed for the ***Core project*** [27]. So there's an entire section describing the ***cloud*** part's details. After this part, there's a review of 2D and 3D features descriptors, which are important because they are used on the object's classification. The last part of this chapter introduces the ***agile grasp*** algorithm [6] used to retrieve the grasping positions.

The following chapter gives a description of the ***act module***. It also presents a brief introduction to the motion planning library ***MoveIT!*** and a description of the Nao's hardware.

The experiment's chapter shows the results of grasping objects on a *tabletop scenario*. There are two types of experiments: single objects and objects in a clutter. Another interesting test is done using previously grasped objects found on the ontology database. The last test of this chapter is done by asking the robot a specific object's class to demonstrate that the reasoning module works.

The last chapter contains a discussion on the experiment's results and some suggestions on possible future work that could be done in order to extend the analysis proposed by this Thesis.

1.3 Introduction to the sense-model-act framework

A robotic *paradigm* is a mental model of how a robot operates, The ***sense-model-act*** framework is a robotic paradigm composed by three elements that a robot must have in order to operate:

- ***Sense*** - the robot acquires information about the environment using this element. The information is, for example, sensor's feedback, like images, force sensor's feedback, etc.

- ***Model*** - the robot takes the sensed data and have to respond accordingly to it. This element creates an action's sequence (plan) to be executed.

- ***Act*** - this element executes the action's sequence generated by the the Model module.

This paradigm was among the first proposed in the *70's*, it implies a *closed* world model. The external world's representation is an idealized model and the robot is ***not*** part of it. The paradigm is useful if the sensing phase is slow, and the environment is static. The act phase should be fast enough in order to jump immediately to the sense phase again and read new changes on the environment.

The model phase is the real bottleneck of this framework, because the module may have to handle many states produced by the sense module inputs. More states means generating more possible actions and then decide the best action to take at a certain time.

Figure 2 shows the relation between the different phases of the sense-model-act framework. In this case the sensing and action along with the environment are showed together in the lower part of the scheme. This lower part remembers the scheme of another well known machine learning framework: *Reinforcement learning*. The idea behind this Thesis's work is to improve the model phase by using Reinforcement learning and to speed up the entire pipeline by using the accumulated knowledge shared on the database that resides on the *cloud*.

1.4 Software - ROS Middleware

The entire Thesis's pipeline is implemented using ***ROS (Robot Operating System)***.

ROS is a robotics opensource *middleware*. The main vantage of this middleware is the infrastructure layer, that's shared by all platforms using the operating system. In this way there are many of libraries already implemented and of easy integration. The robotics community is very active on developing new libraries for this system, so we can find a lot of algorithms,

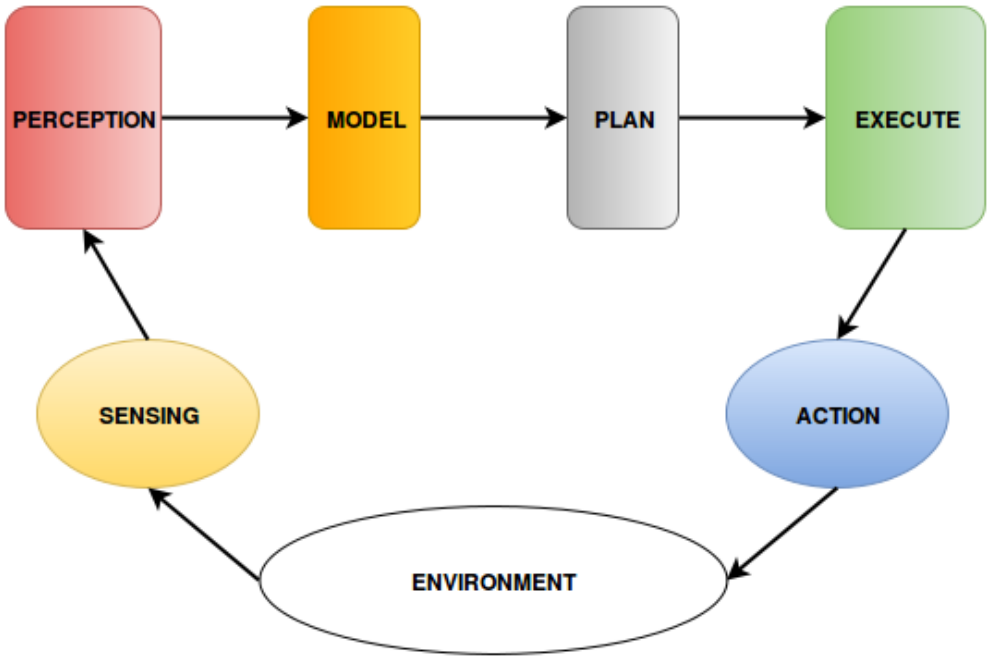


Figure 2: The sense-model-act scheme. The different phases that compose the sense-model-act modules are detailed.

Figure 3: ROS Indigo version's logo. The framework's version used in this Thesis's work.

simulations, etc. For a proper introduction to ROS visit the official website ([2]). The version used in this Thesis is the ***Indigo*** version (figure 3) running on **Ubuntu 14.04 LTS**.

1.5 Software - the Gazebo simulator

Gazebo is an open source simulator for robotics[1]. It offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It has a robust physics engine, high-quality graphics and convenient programmatic and graphical interfaces.

Gazebo is compatible with a particular XML file used in ROS as standard, the *Universal Robotic Description Format* (URDF) that serves to describe all the elements of a robot.

[1] http://gazebosim.org/

1.5.1 Modifying the Nao's URDF model

The Nao's ROS model is implemented by the package ***nao_description***. This package is necessary to run the real Nao and the virtual Nao. This section shows how to modify the *nao_description* package in order to add a virtual kinect that will help later to calculate the distance to objects and to use the simulation on the recognition's pipeline.

Inside this package there are two files:

- *nao_sensors.xacro*
- *naoGazebo.xacro*

Note that the file's extension is not *.urdf* but *.xacro*. *Xacro* is an ***XML Macro language*** which permits to reduce large XML expressions using some macro definitions, see[2] for details.

These two *.xacro* files where modified in order to add the RGB-D sensor (kinect) to the robot.

On the first file the following section was added (at the top of the head joint)

```
<joint name="camera_depth_joint" type="fixed">
    <origin xyz="0 0 0" rpy="0 0 0" />
    <parent link="CameraDepth3_frame" />
    <child link="camera_depth_frame" />
  </joint>

  <link name="camera_depth_frame">
    <inertial>
      <mass value="0.01" />
      <origin xyz="0 0 0" rpy="0.0 0.0 0.0" />
      <inertia ixx="0.001" ixy="0.0" ixz="0.0"
       iyy="0.001" iyz="0.0"
       izz="0.001" />
    </inertial>
  </link>

  <joint name="camera_depth_optical_joint" type="fixed">
    <origin xyz="0 0 0" rpy="0 0 0" />
    <parent link="camera_depth_frame" />
    <child link="camera_depth_optical_frame" />
  </joint>
```

[2] http://wiki.ros.org/xacro

```
<link name="camera_depth_optical_frame">
  <inertial>
    <mass value="0.001" />
    <origin xyz="0 0 0" rpy="0 0 0" />
    <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
     iyy="0.0001" iyz="0.0"
     izz="0.0001" />
  </inertial>
</link>

<joint name="camera_rgb_joint" type="fixed">
  <origin xyz="0 -0.005 0" rpy="0 0 0" />
  <parent link="CameraDepth3_frame" />
  <child link="camera_rgb_frame" />
</joint>

<link name="camera_rgb_frame">
  <inertial>
    <mass value="0.001" />
    <origin xyz="0 0 0" />
    <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
     iyy="0.0001" iyz="0.0"
     izz="0.0001" />
  </inertial>
</link>

<joint name="camera_rgb_optical_joint" type="fixed">
  <origin xyz="0 0 0" rpy="0 0 0" />
  <parent link="camera_rgb_frame" />
  <child link="camera_rgb_optical_frame" />
</joint>

<link name="camera_rgb_optical_frame">
  <inertial>
    <mass value="0.001" />
    <origin xyz="0 0 0" />
    <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
     iyy="0.0001" iyz="0.0"
     izz="0.0001" />
  </inertial>
</link>
```

Note that the origin *TAG* of the ***camera_rgb_joint***

```
<joint name="camera_rgb_joint" type="fixed">
```

```
    <origin xyz="0 -0.005 0" rpy="0 0 0" />
    <parent link="CameraDepth3_frame" />
    <child link="camera_rgb_frame" />
  </joint>
```

defines the relative position of the RGB-D sensor to the robot's head joint.

The second file, *naoGazebo.xacro* add the RGB-D sensor as a virtual device, so the camera takes the images of the virtual scene and elaborates them as real images. The following section was added after the normal RGB camera's section:

```
<gazebo reference="CameraDepth3_frame">
            <sensor type="depth" name="openni_camera_camera_ale">
                <always_on>true</always_on>
                <update_rate>20.0</update_rate>
                <camera>
                    <horizontal_fov>1.2</horizontal_fov>
                    <image>
                        <format>R8G8B8</format>
                        <width>640</width>
                        <height>480</height>
                    </image>
                    <clip>
                        <near>0.05</near>
                        <far>8.0</far>
                    </clip>
                </camera>
                <plugin name="kinect_camera_controller"
                filename="libgazebo_ros_openni_kinect.so">
                    <cameraName>kinect_ale</cameraName>
                    <alwaysOn>true</alwaysOn>
                    <updateRate>10</updateRate>
                    <imageTopicName>
                    /CameraDepth3_frame/rgb/image_raw
                    </imageTopicName>
                    <depthImageTopicName>
                    /CameraDepth3_frame/depth/image_raw
                    </depthImageTopicName>
                    <pointCloudTopicName>
                    /CameraDepth3_frame/depth/points
                    </pointCloudTopicName>
                    <cameraInfoTopicName>
                    /CameraDepth3_frame/rgb/camera_info
                    </cameraInfoTopicName>
                    <depthImageCameraInfoTopicName>
                    /CameraDepth3_frame/depth/camera_info
```

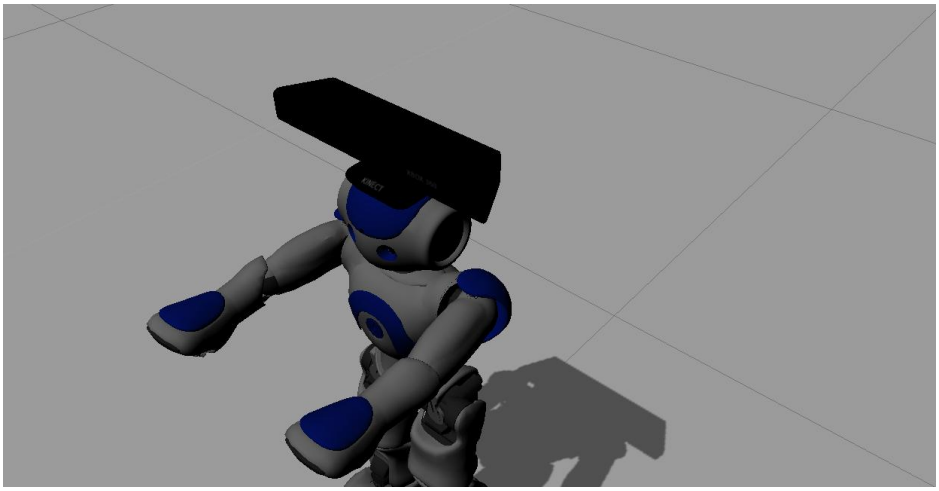


Figure 4: The Nao robot with the RGB-D sensor on Gazebo.

```
                    </depthImageCameraInfoTopicName>
                    <frameName>camera_depth_optical_frame
                    </frameName>
                    <baseline>0.1</baseline>
                    <distortion_k1>0.0</distortion_k1>
                    <distortion_k2>0.0</distortion_k2>
                    <distortion_k3>0.0</distortion_k3>
                    <distortion_t1>0.0</distortion_t1>
                    <distortion_t2>0.0</distortion_t2>
                    <pointCloudCutoff>0.4</pointCloudCutoff>
                  <pointCloudCutoffMax>5.0</pointCloudCutoffMax>
                </plugin>
            </sensor>
        </gazebo>
```

The most interesting tags of this listing are the ones named *TopicName* because the RGB-D sensor's output will be published using these topics.

The final result is showed in figure 4, where the Gazebo's Nao presents the new RGB-D sensor at the head's top.

Figure 5: Laboratory objects prepared for the tabletop detection's problem.

2 The sense module

The perception module is composed by a group of ROS nodes and a RGB-D sensor.

These two components are used to solve the ***Tabletop object detection*** problem that will be presented as the first part of our grasping task.

The entry point for the grasping task is *the sense module*, this task is designed as follows: there are N objects on a table (in a *clutter* manner and not) and the robot must pick up every object and place it on a box that resides near the table's side, this is *the Tabletop detection problem*. The task is designed in such way in order to have the best conditions for the object's perception.

Figure 5 shows some objects disposed for the tabletop detection's problem.

The design of the whole module tries to reach the following constrains:

- There isn't prior knowledge of the scene.
- Not using a database to detect the object and the corresponding complete 3D model.
- Not having an offline learning's phase.

Some popular alternatives to detect objects are the **tabletop_object_detector** module[3], originally created by Willow Garage for the ROS platform. The purpose of this module is to provide a means of recognizing simple household objects placed on a table such that they can be manipulated effectively. This module uses a similar algorithm for the segmentation step but in order to recognize an object, it uses a model database, so the module can't recognize an unknown object and the database is the classical relational *SQL* database, not well suited for sharing knowledge between robots.

Another popular alternative is the ***Object recognition kitchen (ORK)*** framework[4] that's also being developed by the Willow Garage's team. It's a complete framework providing a lot of algorithms for object detection, e.g. for transparent objects, non textured objects, articulated, etc. It also contains an entire pipeline's implementation for object detection, giving the database layer, input/output handling and more. The problem with this library is that not provides the latest state-of-art algorithms for object detection and the database is a classic relational *SQL* database, being hard to scale and handle many objects at the same time. Lastly being a complete framework it's not easy to integrate with ROS (but provides some ROS communication functions).

Two interesting works serve as inspiration to the development of this module, the first is [18], in this Thesis a complete pipeline is implemented, following the objectives of the present module. A ROS node was released containing all the implemented code, ***rail_object_detection***, but the problem is that this node is implemented using an old ROS version and thus parts of the module must be rewritten in order to use it with the *INDIGO* version.

The second work is [19]. This technical report proposes a similar pipeline to that of the first, but using the latest state-of-art algorithms for segmentation (using the implementation present in the *PCL* library). This work will be the starting point of the sense's module implementation.

Before starting with the sense module's description the following three sections will explain the tools needed and used on the development of the module.

[3] http://wiki.ros.org/tabletop_object_detector

[4] http://wg-perception.github.io/object_recognition_core/

2.1 Software - The Point Cloud Library

The Point Cloud Library (*PCL*) is an open source project (*BSD* license) for 3D geometry processing and point cloud processing. This project was originally develop by Willow Garage in 2010 as a ROS package. The library is divided in the following modules:

- filters
- features
- keypoints
- registration
- kdtree
- octree
- segmentation
- sample_consensus
- surface
- recognition
- io
- visualization

The PCL is the tool used in this Thesis to work with point clouds that are the outputs of RGD-D sensors. As stated before, the entire sense module uses state-of-art algorithms that are implemented inside the PCL library.

2.2 Software - The OpenCV library

OpenCV (*Open Source Computer Vision Library*) is a popular open source library (using the *BSD* license) that contains a lot of implementations of computer vision algorithms. It's cross-platform and although being developed mainly on *C++* there are bindings for popular languages like *java*, *python* and *C#*.

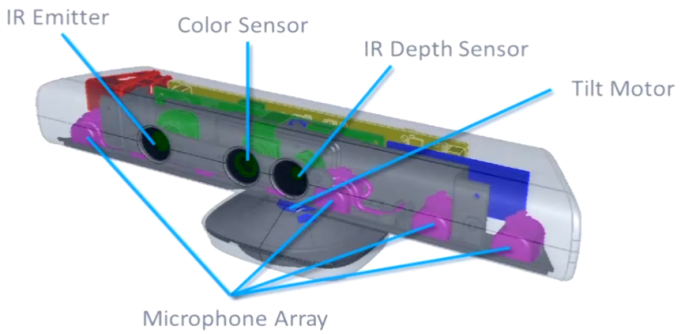


Figure 6: The main parts that conforms the MS Kinect.

OpenCV is used as the primary vision library in ROS, where there are conversion functions between image formats and ROS messages.

In this Thesis this library is mainly used because it's integrated inside ROS and because it contains the algorithm's implementations that extract ***image descriptors*** from images (see the dedicated section on this chapter).

2.3 Hardware - The RGB-D sensor

The *sense module* makes use of the Microsoft's RGB-D sensor: the ***Kinect***. the Microsoft Kinect is a console Xbox accessory, and became the fastest selling consumer electronics device in the world after its launch in November 2010.

Kinect and other RGB-D or 3D sensors consist of a color (**RGB**) camera and depth (**D**) sensor, figure 6 shows the kinect's main parts. The color sensor takes the RGB values, the IR Depth sensor combined with the IR emitter takes the depth values and the microphone array is used for speech recognition. In order to *see* objects, they must be placed at a minimum distance of 1.2 meters to a maximum distance of 3.5 meters. The output camera's resolution is 640x380 pixels at running at 30Hz.

Typically the RGB-D sensor's output is a point cloud image, an RGB-D image. Since the kinect's release lots of researches around the world are producing different datasets and using this device on a wide range of applications.

Generally, RGB-D sensors are low accuracy and low precision devices. Diverse studies (i.e. [16]), show that their accuracy is on the order of 2-3%. At a distance of 4m from the sensor, this corresponds to *RMS error*

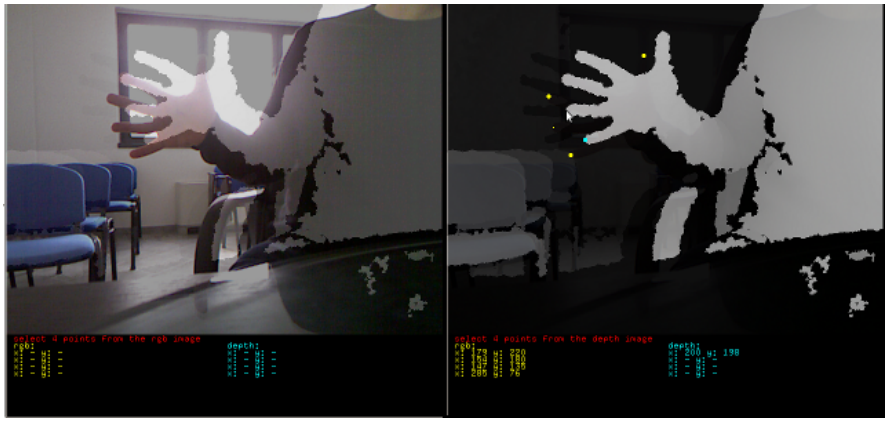


Figure 7: Calibration procedure in order to associate the correct pixel with the correct depxel.

on the order of 10cm. This level of accuracy is quite satisfactory in e.g. human interaction applications. However, it may appear unsuitable in some specific robotic uses (e.g. grasping). Since grasping is our actual task to be performed, a calibration procedure was applied to the camera. The rgb and depth camera in the kinect need to be calibrated in order to associate the correct pixel with the correct *depxel*.

This procedure can be done by using an specific ROS's package[5]. The calibration process improves the level of accuracy, for example the figure (7) shows the image's color quality before and after the calibration procedure.

2.4 Tabletop object detection problem

The main task of the sense module is detecting the objects that the robot has to interact with.

The tabletop object detection phase is composed of 3 different steps, first the plane's table must be separated from objects:

- **Table plane estimation** (by RANSAC): the first step is find the table's plane. This is done using the ***RANSAC (RANdom SAmple Consensus")*** algorithm [11], witch is an iterative method to estimate the parameters of a mathematical model from a dataset containing *outliers*, so the method return witch points of the input dataset fits

[5]http://wiki.ros.org/openni_launch/Tutorials/IntrinsicCalibration

a certain model (with some level of accuracy the algorithm is non-deterministic). In our case the input model to fit will be a plane, the table's plane. The points of the table are detected estimating first a plane in the point cloud, all the points which belong to such a plane are the points of the table.

- **Calculating the 2D Convex Hull of the table**: The hull could be defined as the points that conform the outermost boundary of the point's set, like a shell around the volume. So once we have the points of the table's plane model, a 2D convex hull is computed in order to get a 2D shape (the plane itself) containing those points. Figure 8 shows a table's plane and the calculated hull.

- **3D Polygonal prism creation and projection**: the hull calculated on the previous step is extruded at a given height, creating a 3D prism. All the points that lie inside this prism are extracted. So all the points are projected on the table plane previously estimated and all the points which projections belong to the 2D convex hull are considered to be points of tabletop objects.

The main vantage of this algorithm is that it's able to detect the tabletop objects by detecting the table's plane, so other possible planes of the scene like the floor's plane will be discarded since the projections of that planes on the table's plane don't belong to the table's *convex hull*.

This part uses the PCL's implementations. A tutorial about using a planar model with RANSAC is found here[6], for convex hull there's another tutorial[7] and the 3D prism calculation is showed in this tutorial[8].

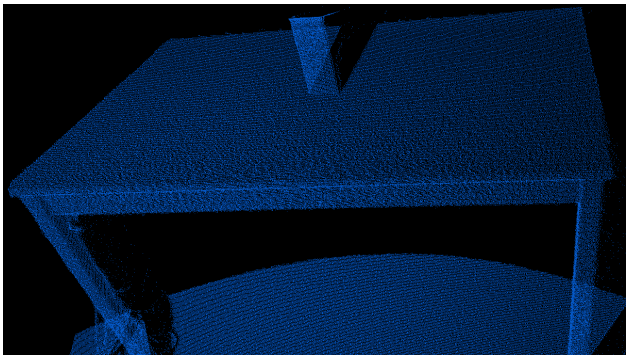
2.5 Point cloud Segmentation

Segmentation is the process where the scene's point cloud is divided (*segmented*) onto different point clouds one by every detected object, so the final result is a set of separated point clouds. This process gives the vantage of working with smaller point clouds, so the performance of many algorithms that use these point clouds can be improved a lot.

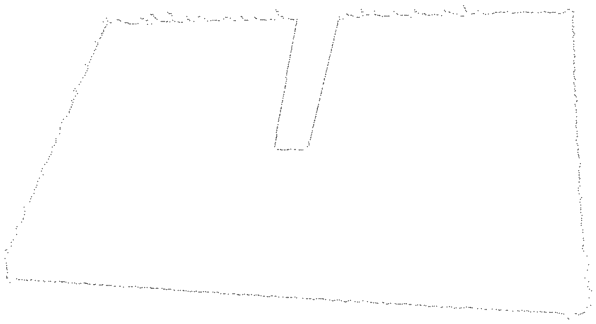
[6] http://www.pointclouds.org/documentation/tutorials/project_inliers.php

[7] http://www.pointclouds.org/documentation/tutorials/hull_2d.php

[8] http://robotica.unileon.es/mediawiki/index.php/PCL/OpenNI_tutorial_3:_Cloud_processing_%28advanced%29



(a) table's point cloud.



(b) Calculated hull.

Figure 8: These figures show the tabletop's point cloud and the calculated hull.

So once the objects have been detected (using the previous section's methods), another phase must be performed: the point cloud segmentation phase.

2.5.1 The supervoxel concept

The principal goal is to deconstruct a scene into separate object parts without the need for *training or classification*. As psycho-physical studies suggest, in humans the lowest level decomposition of objects into parts is closely intertwined with 3D concave/convex relationships. The **VCCS (Voxel Cloud Connectivy Segmentation)** algorithm [15] used in this phase (an overview of which is given in 9, the image was taken from this[9] tutorial) tries to reliably identify regions of ***local convexity*** in point cloud data.

The *VCCS algorithm* over-segments 3D point cloud data into patches. These patches are called *supervoxels*. A supervoxel is a group of voxels that share similar properties. Supervoxels are the 3D analog of *superpixels*[10]. Using voxels is possible since modern RGB-D sensors provide the object's 3D geometry as output.

The **VCCS** algorithm is divided in the following phases:

1. Generation the labeling of points. This is done using a variant of *k-means clustering*. The result of this phase is a *voxelized* point cloud.

2. Adjacency Graph construction. Using the voxelized point cloud the graph is constructed and is a key element for the segmentation.

3. Spatial Seeding. In this phase the algorithm selects some point cloud's points that will be used to initialize the supervoxels.

4. Features and Distance Measure. Supervoxels are clusters in a 39 dimensional space. This vector is created using the spatial coordinates, the point's color information and a 33-dimensional feature vector that is obtained using the ***Fast Point Feature Histograms (FPFH)*** algorithm [22]. This a local geometry extractor method.

[9] http://pointclouds.org/documentation/tutorials/supervoxel_clustering.php

[10] ***Superpixel:*** A polygonal part of a digital image, larger than a normal pixel, that is rendered in the same colour and brightness.

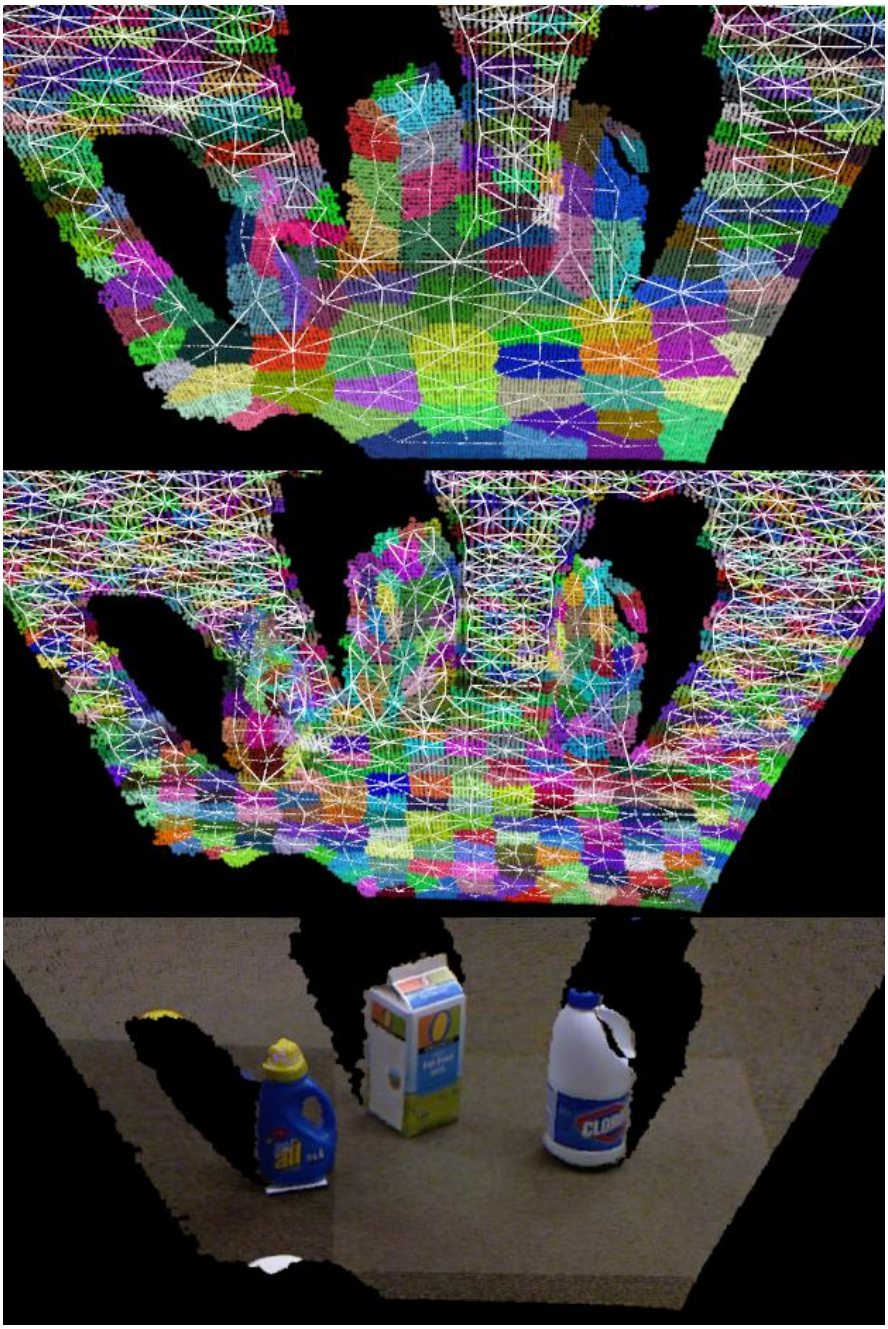


Figure 9: the **VCCS** algorithm's phases.

5. Flow Constrained Clustering. The last phase assigns at every point cloud's voxel a supervoxel. This is done iteratively, using a local k-means clustering method.

The supervoxels method works directly on point clouds, which has advantages over other methods which operate with projected images. An important characteristic of this algorithm is the ability to segment clouds coming from many sensor observations i.e. using multiple cameras or accumulated clouds from one. Computationally, this is advantageous, as the speed of this method is dependent on the number of occupied voxels in the scene, and not the number of observed pixels. As observations will have significant overlap, this means that it is cheaper to segment the overall voxel cloud than the individual 2D observations.

2.5.2 Segmentation algorithm

Using the supervoxels of the previous section there are a lot of interesting algorithms to perform object's segmentation. The ***Object Partitioning using Local Convexity*** [7] is a good algorithm that can be found in the last PCL's release, the version 1.7.

There's another state-of-art algorithm that can be found in the the not stable release of the PCL'S library, the 1.8. This is the ***Local Convex Connected Patches Segmentation (LCCP)*** [7] that is a state-of-art algorithm for segmentation using supervoxels as input.

2.5.3 LCCP algorithm

The **Local Convex Connected Patches Segmentation** (as other segmentation algorithms based on voxels) takes the assumption that objects are ***convex***. This true for many objects but not for all and is good to have present this condition on the experiment's phase.

The principal goal is to deconstruct a scene into separate object parts without the need for *training or classification*. As psycho-physical studies suggest, in humans the lowest level decomposition of objects into parts is closely intertwined with 3D concave/convex relationships. The **VCCS (Voxel Cloud Connectivy Segmentation)** algorithm [15] used in this phase (an overview of which is given in 10) tries to reliably identify regions of ***local convexity*** in point cloud data.

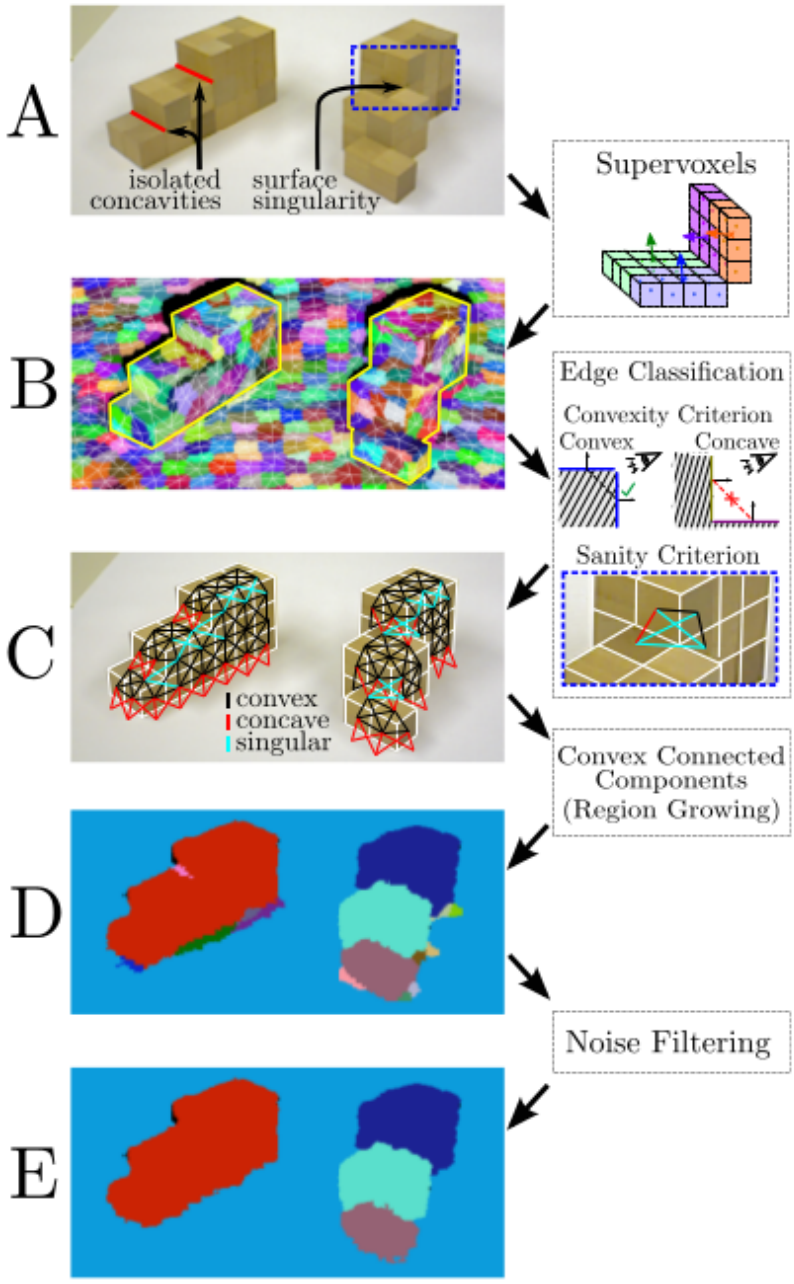


Figure 10: Segmentation phases using the **VCCS** algorithm. The image was taken from [15]

The *VCCS algorithm* over-segments 3D point cloud data into patches. These patches are called *supervoxels*. A supervoxel is a group of voxels that share similar properties. Supervoxels are the 3D analog of *superpixels*. Using voxels is possible since modern RGB-D sensors provide the object's 3D geometry as output.

The main algorithm's phases are presented in the Figure 10.

The ***VCCS*** algorithm gives a surface patch adjacency graph as output, this supervoxel adjacency graph is segmented by classifying the connection between two supervoxels is convex (valid) or concave (invalid). Two criteria are used to classify this connection:

- ***Extended Convexity Criterion (CC)*** - uses the centroids of two adjacent supervoxels, the connection between them (covex or concave) can be calculated by seeing the relation of the surface normals to the vector joining their centroids.
- ***Sanity criterion (SC)*** - takes the adjacent supervoxels that are potentially convex and search for surface's geometric discontinuities. If discontinuities are found, connections between these two supervoxels are invalidated.

The *PCL* algorithm's implementation is very good, so the segmentation phase is very fast (using a lowend 2 core duo Thinkpad with 4GB of RAM).

The algorithm also works very well when must segment cluttered scenes. The only problem is that there's a minimum dimension threshold that the input point cloud must have. If the point cloud is too small the algorithm can't see anything. So the algorithm presents problems for ***segmenting small objects***.

For a detailed description of the algorithm's parameters and installation dependencies see [19].

2.6 The sense module package implementation

The sense module was implemented as a ROS package (Figure 11). The package has the following structure:

- **sense metapackage**: container of the sense module's packages.
- **sense_segmentation package**: contains the service that calls segmentation algorithm's implementation.

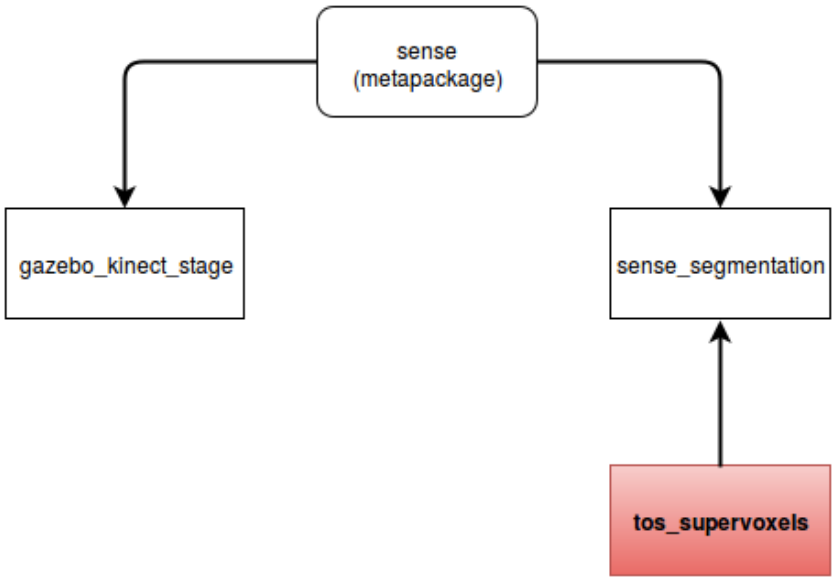


Figure 11: The sense module package's structure.

- **gazebo_kinect_stage package**: contains a simulated scene using the gazebo simulator and a virtual kinect model that produces point clouds.

The main module's node is the **sense_segmentation package**, it provides two launch files that can be used to call the segmentation algorithm:

- **sense_segmentation.launch**: starts a ROS node that receives the point clouds as inputs and publish another *topics* that contains the single object's point cloud.

- **sense_segmentation_server.launch**: is a ROS service that calls the segmentation algorithm and return a list containing the segmented object's point cloud.

The *sense_segmentation* package internally calls another segmentation algorithm that uses the state-of-art ***LCCP*** algorithm of previous section. This is the ***tos_supervoxels*** showed in Figure 11. This implementation was made by [19] as a standalone *C++* program (it must be placed outside the ROS's workspace).

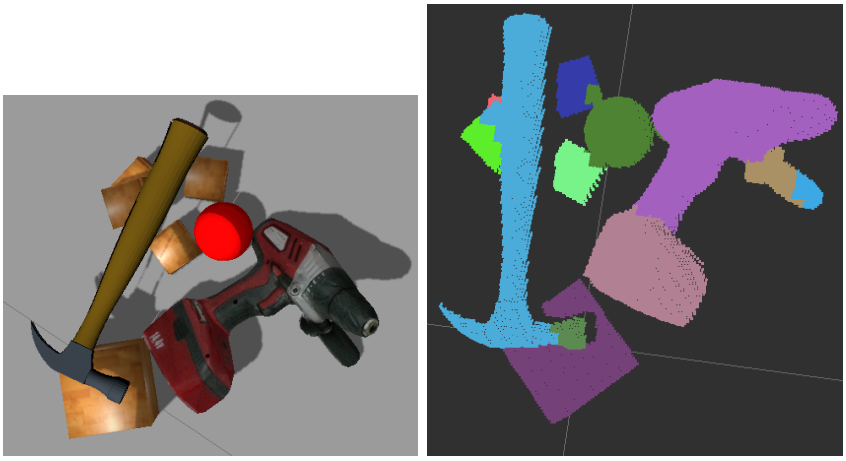
Figure 12 shows a segmentation done using as input the virtual point clouds of the *gazebo_kinect_stage* package.

In order to use the segmentation module inside this Thesis's pipeline, the following ROS nodes must be started. First a real RGB-D sensor must connected, the module listens by default the topic named */camera/depth_registered/points* but can be changed in the *sense_segmentation_server.launch* file. If not RGB-D sensor is available, the simulated Tabletop object stage can started by launching the following console's command:

```
$ roslaunch gazebo_kinect_stage kinect_gazebo.launch
```

Once an RGB-D sensor starts publishing point clouds the ROS segmentation service can be started by using this console's command

```
$ roslaunch sense_segmentation sense_segmentation_server.launch
```



(a) table's virtual point cloud from the **gazebo_kinect_stage package**.

(b) Calculated hull.

Figure 12: These figures show the segmentation phase (images taken using Rviz).

2.7 The image's descriptors

In this section an important argument is covered and is a set of techniques that will help to identify images, compare with others and eventually understand if these images are similar or not. These descriptors are not used in the segmentation phase but will be critical in the matching object's phase that will be presented in the next chapter.

Image content characteristics are often described by ***visual features***. These features are a compact description of the image itself. They are called *visual* because these descriptors visually shows some specials characteristic. Visual descriptors are divided into two big families:

- *Global features* - these features encode the visual content into a single description global these features compromise the entire image. On this family we have features such overall appearance, color, intensity, etc. The main vantage of this family of features is that they are fast to compute due to the single description.

- *Local features* - this family of features describes an image region (named ***patch***) around a point of the image. This point is called ***interest point*** and is a point that helps to *distinguish* the image. Local features algorithms usually contain two steps. The first is the interest point detection, where a set of point of interest are identified on the image. The second step is the extraction of a local feature descriptor around every point.

2.7.1 HOG

Histogram of Oriented Gradients (***HOG***) [8] is a local feature descriptor used with success for human detection and object recognition. HOG descriptors are similar to SIFT because they describe image gradients by a *histogram of gradients orientations*. The key concept behind HOG is that local object appearance and shape within an image can be described by the distribution of intensity gradients or edge directions. The image is divided into small connected regions called cells, and for the pixels within each cell, a histogram of gradient directions is compiled. The descriptor is the concatenation of these histograms. The HOG descriptor has a few key advantages over other descriptors. Since it operates on local cells, it is invariant to geometric and photo-metric transformations, except for object orientation.

2.7.2 SIFT

SIFT is an algorithm proposed by [17] in 2004 since it's publication has become the most widely used local feature algorithm in the domain of feature retrieval. The proposed SIFT features contain the two methods to find local features: an scale and rotation-invariant detection of interest point and a

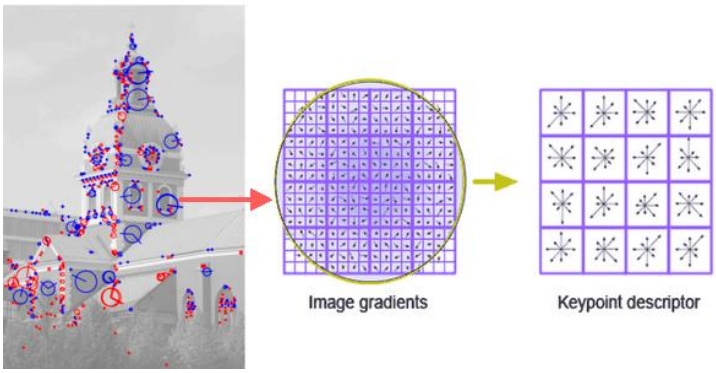


Figure 13: shows how a SIFT point is described using a histogram of gradient magnitude and direction around the feature point.

robust description of local neighborhoods around these points. Figure 13 shows a SIFT's descriptor example.

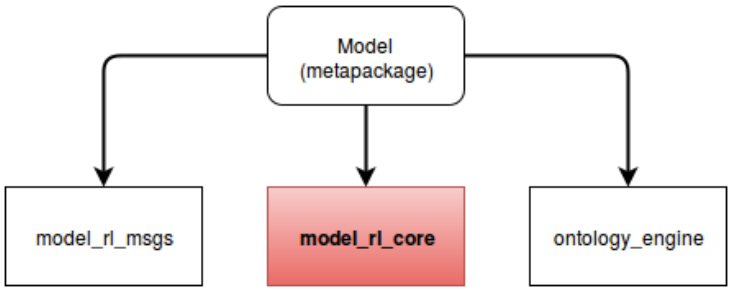


Figure 14: Model package's structure.

3 The model module

This is the main Thesis's chapter. First a general description of the *model* ROS package is given, in order to understand how the different module's parts are connected. Follows an introduction about the state of the art in Cloud Robotics and a presentation of a past work that gives inspiration to the cloud part of the Thesis. Then we have a description of the server side's ROS nodes. At this point a general introduction to ontologies is given and to the *CORA* standard [24]. Follows a description of the ontology engine package that retrieves the information of an ontology RDF database.

Finally the last introduction of this chapter is dedicated to Reinforcement learning, after a generic introduction, some state-of-art grasping algorithms are presented. The last section that closes this chapter describes the implemented Reinforcement learning node.

3.1 The model package

The entire model package's structure is showed in figure 14. There's a ROS metapackage containing the other packages, the main package (representing the core of the model part of the framework) is showed in red color. This is the ***model_rl_core*** package. This node controls the others, *senses* the ambient using the sense module, reasons using the cloud engine and the reinforcement learning engine, and finally *acts* using the act module.

In order to launch the ***model_rl_core*** package we must pass one parameter: the cloud's server IP (witch must have the form *ip:port*), so from the console we can use the following command

```
# roslaunch model_rl_core model_rl_core_client.launch
```

```
server_ip_addr:=CLOUD_SERVER_IP:9090
```

At the initialization phase, the node waits for a point cloud of the scene that arrives from the RGB-D sensors, the topic's name where the node waits for the point cloud is ***/camera/depth_registered/points*** but can be changed inside the file *model_rl_core model_rl_core_client.launch*. When the new point cloud arrives the node calls the *sense_segmentation_service* that returns a list with the segmented object's point clouds. At this point the object can be grasped by the robot but first this reasoning engine must *understand* how to grasp the object itself. The node finds the grasping by calling:

1. The cloud engine - searches the best object's grasping.
2. The reinforcement learning engine - if not grasping if found by the cloud part.

The following sections will explain these two engines. Before the engine's description some arguments are introduced in order to facilitate the understanding of the engines.

3.2 Cloud Robotics

The relationship between network and robots begins more than 30 years ago. Industrial robots began using primitive networks to be controlled from remote locations. Since then and after the incredible development of Internet, robots began using the same protocols, like the *HTTP* protocol, to communicate between them. Until a few years ago, communication between robots was not standardized and/or engineered. There was not an standard that all robots could use but only *adHoc* communications.

In 2010 James Kuffner coined the term ***Cloud Robotics*** to define the branch of robotics related to robot's communication using the modern *cloud technologies*. This new branch is also related to another popular branch of computing, the *Internet of things* that describes how devices communicate and share information.

The *Cloud* can help robotics development in many ways, not only with communication between robots but also by extending hardware, software and cognitive possibilities of every single robot. Efforts are now focused on developing the standard protocols and tools in order give an easy and fast access to remote resources to robots.

This survey [9] gives a complete introduction (also as a historic point of view) to *Cloud Robotics*.

3.2.1 Related work and state-of-art developments

In the last years many works focused on standardizing robot's cloud access. An popular work is *RoboEarth* [20]. A big project that provides standards to share information between robots using the Internet infrastructure as channel. The way that robots share information is very important because knowledge must be managed in a certain way in order to optimize sharing and using with robots. On the *RoboEarth* platform *knowledge* is represented by using an *Ontology* (see the next section for further details about ontologies) but the problem is that it's not an standard ontology. Anyway the main problem with *RoboEarth* is that the system has not an ***scalable*** recognition pipeline.

Another interesting state-of-art work is ***CORE*** [27]. It proposes a complete object's recognition pipeline but using a distributed and scalable architecture. The best part of this work is the distributed architecture that introduces modern web application standards like *JSON* and *websockets* on a Cloud Robotics Architecture. The creators payed particular attention to optimize transmission of large amounts of data. In particular the generated point clouds are compressed (using the method described in [14]) before transmission and the compressed point clouds are only send to server only if the scene's variation is over a fixed threshold. This techniche is known as ***point cloud culling***. Using this method [5] point clouds are selectively culled prior to transmission to the cloud. Culling is performed by measuring the scene ***entropy*** of sequential point cloud frames.

On the other side this work has some characteristics that are not well suited for this Thesis's objectives: the recognition engine needs a separated ***training phase*** to work correctly since learning it's based on a *Support Vector Machine (SVM)*. Finally as authors points out the work could be extended to develop a complete *service oriented architecture*.

3.2.2 Introduction to ontologies

On the machine learning domain there are many definitions of an ontology; some of these contradict one another. A common definition follows: *an ontology is a formal explicit description of concepts in a domain of discourse (classes (also called concepts)), properties of each concept describing various*

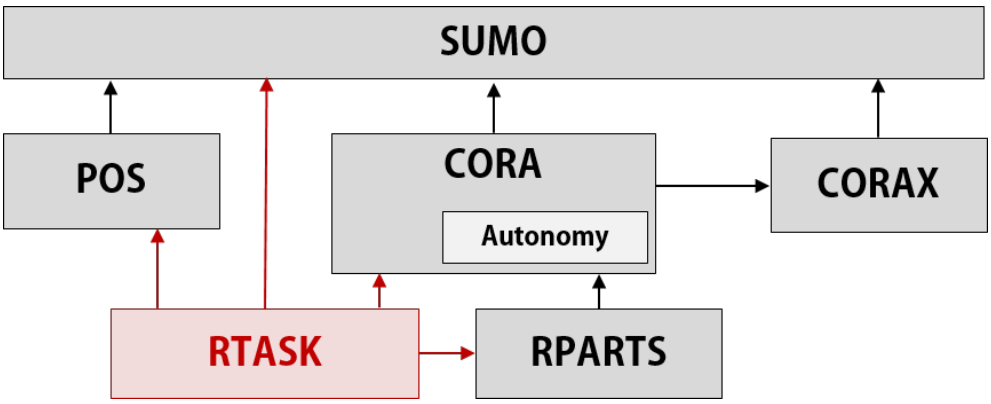


Figure 15: The RTASK extension to the IEEE Ontology for Robotics and Automation.

features and attributes of the concept (slots (also called roles or properties)), and restrictions on slots (facets (also called role restrictions)). An ontology together with a set of individual instances of classes constitutes a *knowledge base.*

With respect to the existing Knowledge Bases, the one proposed, that will be called ***RTASK*** [10], is scalable because of the adoption of an Ontology that defines data. It guarantees an intelligent data storage and access because of the type of data saved. Every object is characterized by multiple visual features (2D Images, B-Splines, and Point Clouds). No onerous manual work is required to store objects from different view points: an object is stored even if there exists only a single registration of one its views. A human teacher helps robots in recognizing objects when viewed from other orientations. The teacher exploits the connection between the new view point and other object properties, e.g., name and function. These new features will be stored in the Ontology gradually incrementing robots knowledge about the object itself. Moreover, the proposed Ontology observes the IEEE standards proposed by the IEEE Robotics and Automation Society (RAS)'s Ontology for Robotics and Automation (ORA) Working Group (WG) by extending the Knowledge Base it proposed (see Figure 15).

3.2.3 The proposed RTASK ontology

Figure 16 depicts the Ontology design: a Task is assigned to an Agent, e.g., a Robot. It should be executed within a certain time interval and requires

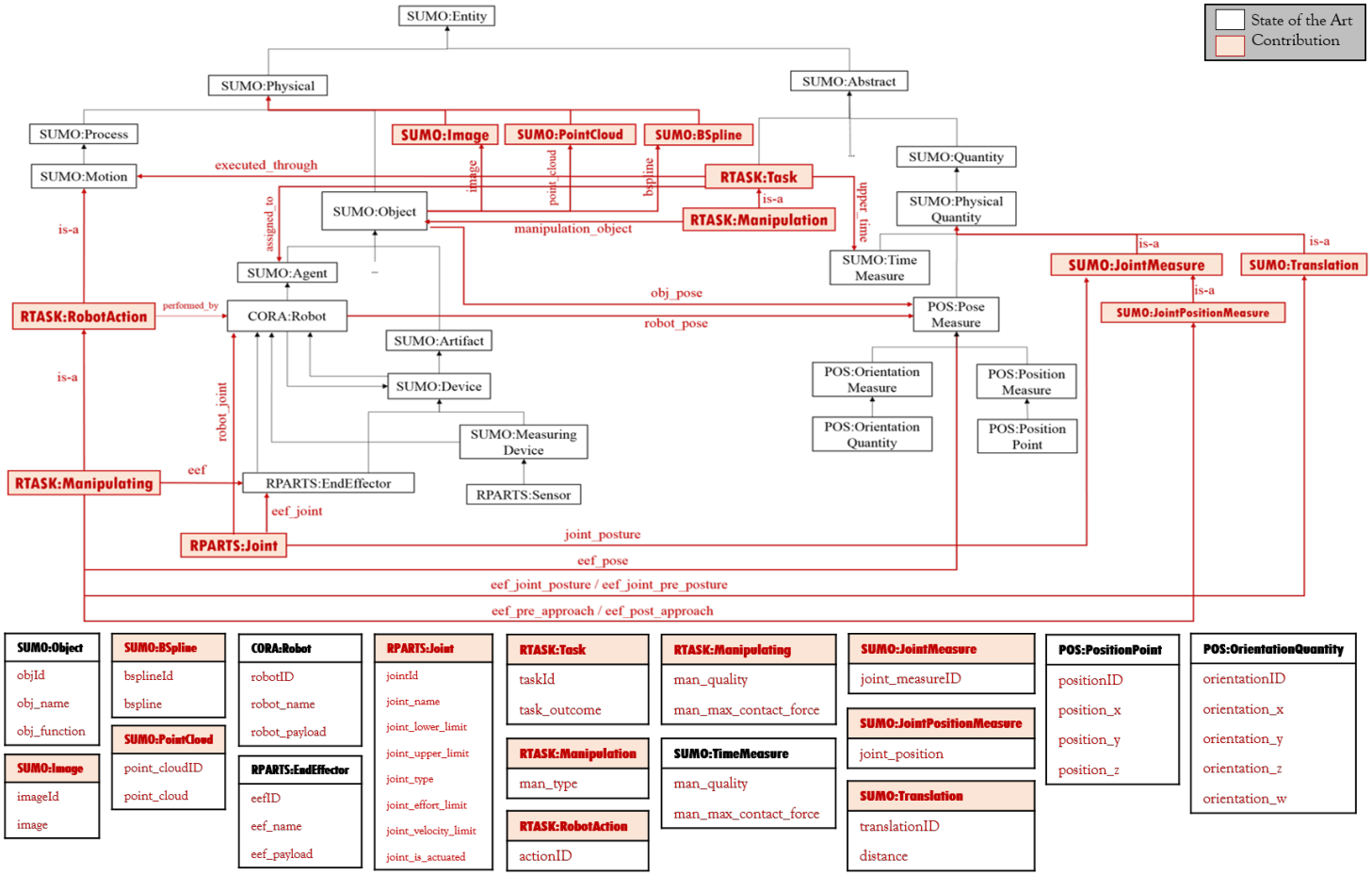


Figure 16: The implemented ontology.

the fulfillment of a certain Motion in order to be performed. Manipulation is a sub-class of Task. Several types of manipulations exist, e.g., grasps and pushes.

They involve the handling of an Object located at a certain Pose (Position and Orientation) through the execution of a Manipulating action. If the Task is assigned to a Robot, then the Motion will be represented by a Robot Action. In detail, the Robot Manipulation Action involves the activation of the robot End Effector.

In order to retrieve the manipulation data of an object in the scene, the object should be recognized as an instance previously stored in the Ontology. For this purpose, every Object is characterized by an id, name, function, and the visual features obtained by the Sensors. For every Object, RTASK stores multiple types of visual features: 2D Images, B-Splines, and (compressed) Point Clouds.

3.2.4 The implemented Cloud-based Engine

The local robot that's performing the grasping task has an internal *ROS node* that receives the segmented point clouds of the scene's objects and sends them to a cloud server that is running another *ROS node* capable of reading the message's content. The communication between nodes uses the rosbridge interface, providing a websocket channel between nodes, the actual implementation takes as starting point [27] but extends it by implementing a complete ROS service architecture. Figure 17 shows a working example of this service oriented architecture.

There are 3 types of messages: ***Request recognition object grasping*** This message asks the cloud server to retrieve a grasping for an object. It sends the robot's gripper type and the compressed point cloud of the single object. The compressed point cloud representation saves space and connection time. The compressed point cloud is also encoded in order to be transmitted over the websocket's channel. After the encoding the whole message is converted in JSON format. The ROS message encoded on the websocket's request is

```
# Task's name (pick, place for example)
string task

# Gripper's name
string gripper_id
```

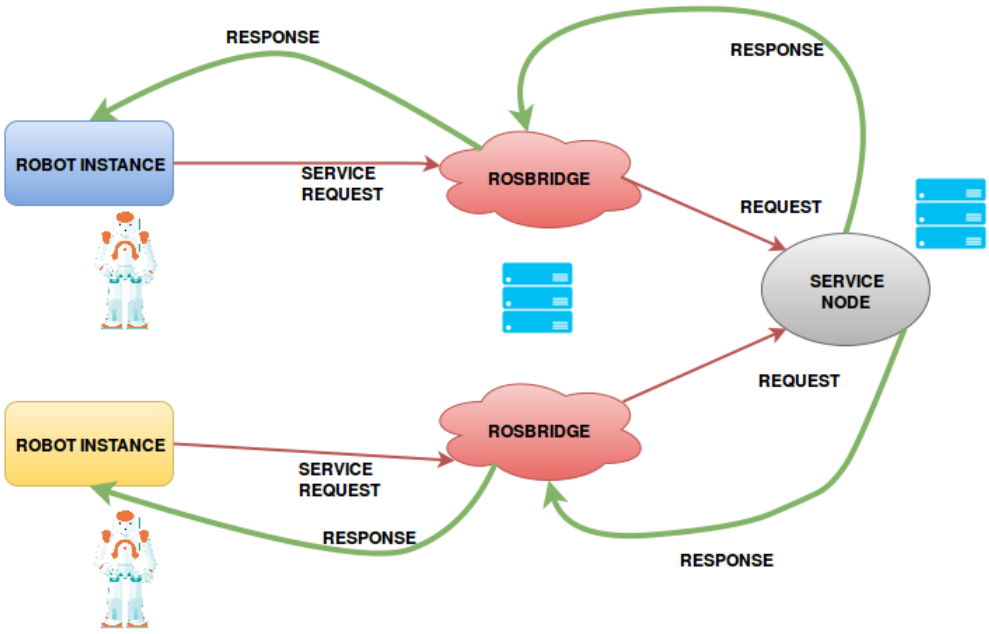


Figure 17: The cloud engine architecture.

```
# Compressed point cloud (object's representation)
string data
```

The server receives the client's request and performs a superfast search that compromises the following steps (Figure 18 shows the pipeline's phases)

- Decode the message and decompress the point cloud.
- Convert the point cloud to color image using the OPENCV's functions.
- Extract the new image's SIFT features (using [25]), and store them on a binary file of a server's folder.
- Match the image's features against the stored features that reside on the server. The matching uses a novel and superfast algorithm, ***Cascade hashing*** [13]. This algorithm is not only very fast but allows constructing a dataset without a learning phase, there's not need to train the hashing functions as in other Approximate Nearest Neighbor (ANN) methods. The function return the names of the features of the most similar images (according to the SIFT parameters) to the input image. These file names are also the object's classes we defined on the

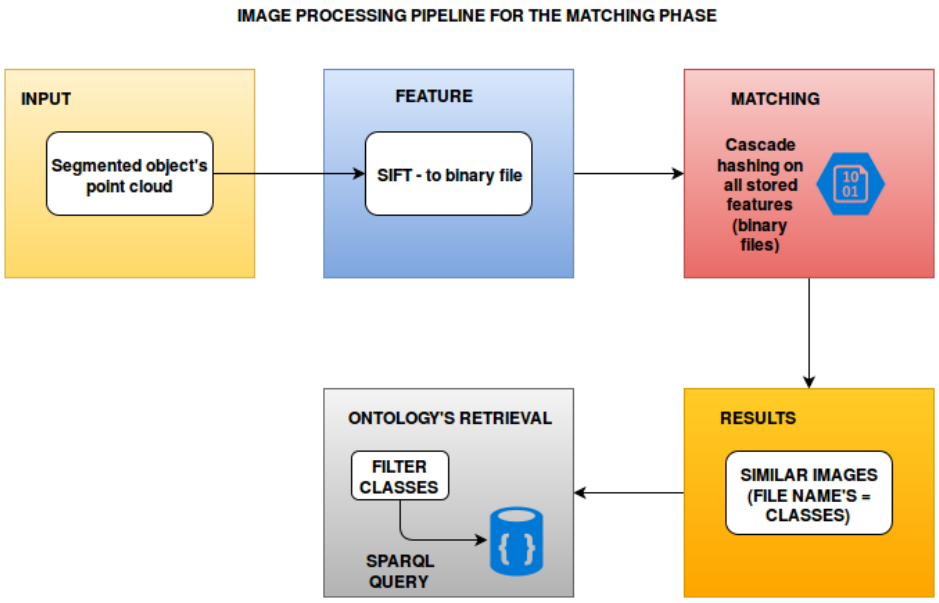


Figure 18: The pipeline for fast image retrieval.

ontology database. So using only one *sparql* query with the classes as filters we reduce the seek time inside the ontology, filtering only the classes of the most similar objects.

- finally the *sparql* query returns the grasping information.

The similar object's searching is very fast not only because we use the cascade hashing algorithm but also because the features of the stored objects are already precalculated and stored on a server folder (they aren't download every time from the database). Along with this, using the feature's name (an integer, like 1,2,3) as object class we avoid to perform many queries on the ontology, performing only one query at the end of the matching process.

The server (using the websocket interface) encodes the following ROS message that contains the matched grasping information:

```
string task_response

string task_id

# The internal posture of the hand for the pre-grasp
# only positions are used
trajectory_msgs/JointTrajectory pre_grasp_posture
```

```
# The internal posture of the hand for the grasp
# positions and efforts are used
trajectory_msgs/JointTrajectory grasp_posture


# The position of the end-effector for the grasp.
# This is the pose of
# the "parent_link" of the end-effector, not actually the
# pose of any link *in* the end-effector.
# Typically this would be the pose of the
# most distal wrist link before
# the hand (end-effector) links began.

geometry_msgs/PoseStamped grasp_pose

# The approach direction to take
# before picking an object
moveit_msgs/GripperTranslation pre_grasp_approach

# The retreat direction to take after
# a grasp has been completed (object is attached)
moveit_msgs/GripperTranslation post_grasp_retreat
```

On the client's side the grasping information is used by the act module to perform the grasping action, if the robot success grasping the object, finish the action. In other case, it comes to play the Reinforcement learning algorithm. First given the point cloud some grasping positions are generated using the geometry's gripper. These positions are then validated by the trial/error process that resides insides the Reinforcement learning algorithm. If a grasping action is successfully executed, an human operator validates the result. With this information the client creates another message:

```
{ "op": "send_new_object",
"service": "insert",
"new_object":
 [
 {"new_point_cloud": "zc5p81H1cO8P+Ksmfdf...",
 "X": "0.5", "Y": "2.5", "Z": "1.5", "rad": "0.314"},
 ]
}
```

Finally the server receives the message, calculates the object's SIFT features and inserts them on the feature's folder and on the ontology.

The last type of message is used for human robot interaction.

```
# Operation type
string operation

# Object class's name
string class_description

# Compressed point cloud (object's representation)
string data
```

In this case the human operator shows an object to the robot and gives an object's description (like coke can, pen), then the server non only seeks for a similar object but filters the ontology using the object's description.

The local node sends the following information to the server using the ***json standard*** as in [27] :

```
{ "op": "search_object",
"service": "send_point_cloud"
"args": "zc5p81H1cO8P+Ksmfdf..." }
```

The *args* part in the above message is the compressed point cloud. The server takes the point cloud, gets the *SIFT* features and search a similar object using the *cascade hashing algorithm* that uses a server's directory containing all the object's features in text format. If a match is found, the algorithm takes the unique ID's of the matched objects and performs an RDF query in order to retrieve the object's grasping information inside the ontology. Then the server returns the following message to the client:

```
{"similar_objects" :
[
 { "class":"1", "X":"0.5","Y":"2.5","Z":"1.5","rad":"0.314" },
 { "class":"2", "X":"0.1","Y":"24.5","Z":"1.3","rad":"0.24" },
 { "class":"3", "X":"0.2","Y":"3.5","Z":"0.5","rad":"2.14" }
]
}
```

If no similar objects are found, the response will be

```
{ "op": "search_object_response",
"service": "query",
"similar_objects":
 [

 ]
}
```

And on the client's side comes to play the Reinforcement learning algorithm. First given the point cloud some grasping positions are generated using the

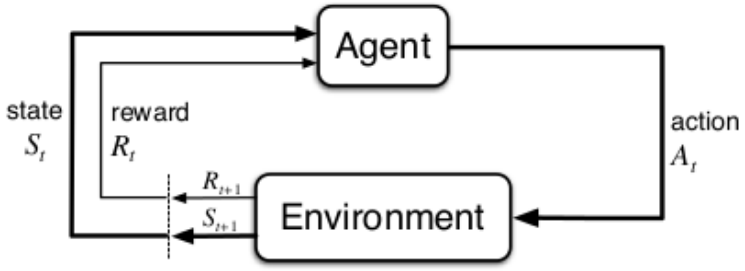


Figure 19: This scheme shows the main elements of a Reinforcement Learning algorithm.

geometry's gripper. These positions are then validated by the trial/error process that resides insides the Rl algorithm. If a grasping action is successfully executed, an human operator validates the result. With this information the client creates another message:

```
{ "op": "send_new_object",
"service": "insert",
"new_object":
 [
 {"new_point_cloud": "zc5p81H1cO8P+Ksmfdf..."},
 "X": "0.5", "Y": "2.5", "Z": "1.5", "rad": "0.314"},
 ]
}
```

Finally the server receives the message, calculates the object's SIFT features and inserts them on the feature's folder and on the ontology.

3.3 Reinforcement learning

3.3.1 Introduction

Learning from the interaction with the environment this is the main concept. Reinforcement Learning (RL) is a branch of Machine Learning where algorithms learns from the interaction with the external ambient, without having (a priori) an specific theoretical model. In it's simplest form, Reinforcement Learning uses a ***Trial and error*** approach, meaning that the algorithm *tries* an action e receives the result, if this result is not enough, it tries again and again, until it reaches the objective.

Being a so simple approach Reinforcement learning was among the first

branch of Machine learning to be studied in the sixties and in the seventies, but since the results were not so brilliant, scientists decided to move to another Machine learning's branch.

A must read introduction to Reinforcement Learning is the Sutton and Barto's book ([23]), the online version of the last edition can be downloaded for free.

The main elements that are always present on a RL algorithm are

- Agent: This element learns from the interaction with the environment. Uses *actions* to interact with the environment and has at least one objective to reach.
- Environment: this element is sometimes seen as the external world to the robot. Has internal states that changes in function to the *agent's actions*.

There are another important elements conforming an RL algorithm:

- Action: the activity that performs the *agent* in order to interact with the *environment*. It can change the environment's state.
- State: The set of internal states that the environment can adopt.
- Sensation: the ability that has the agent to *sense* in witch state the environment is and taking account of this to select the best action.
- Goal: the objective that the agent has to reach when it interacts with the environment.

The last 3 elements that are common in a RL algorithms are:

- Policy: the way in witch the agent must act in an specific time. It's a map that connects the environment's states with the agent's actions.
- Reward signal: the objective that agent has to reach is measured by this signal, typically a real value. This signal is produced by the interaction with the environment. The agent must maximize or minimize (depending on the problem) this signal over the time.
- Value function: The previous *Reward signal* offers an evaluation on the short term, on the long term the *Value function* tries to optimize the reward at a global level.

Figure 19 shows the main elements that conforms a Reinforcement learning algorithm.

There's another element that compares only in some families of RL algorithms, the ***model***. This element *emulates* in some way the environment's behavior, it allows to infer the environment's future behavior and to understand, for example, the next environment's state or the next reward if a certain action is taken. For this reason the model is used to *insert* some kind of *planning* inside the Reinforcement Learning algorithm. Using this predictions the algorithm can *predict* which actions to take.

The Reinforcement Learning algorithms can be classified using this criteria. So we have some RL algorithms that uses a *model* and can perform some kind of planning are called ***model-based***. Instead we have another family of Reinforcement learning algorithms that are pure ***trial and error*** and don't use models or planning, this algorithms are called ***model-free***.

Another classification that can be done is if the Reinforcement learning algorithm is ***Evolutive*** or not. A Reinforcement learning algorithm is evolutive if it doesn't use a *Value function* but *tries* some agents and selects only the best according to a reward function, this is the *Evolution's concept*. This algorithms are normally use when is difficult, according to the problem's nature, identify (for the agent) the Environment's state or alternately there's only one state.

Some Reinforcement learning algorithms are not classified like *evolutives* although they use a Value Function. The policy on this algorithms has the form of numerical parameters so the algorithm search on the policy's space the gradient direction in order to maximize or minimize the policy. They differ from the evolution algorithms because they produce an estimation of the policy using the single actions results. In this way every single action counts instead of evaluating only the final result.

This type of Reinforcement learning algorithm is know as ***policy gradient***.

Figure 20 shows how the different families of reinforcement learning algorithms are correlated. The simplest algorithm (policy gradient) needs more supervision for example.

It's important to remark that all Reinforcement Learning algorithms must perform ***exploration*** of the action's space in order to discover the *best* actions to take. The best action are chosen repeatedly, a concept that is know as ***exploitation***. This is because these actions warrant a good reward. Reinforcement Learning algorithms differ of other learning algorithms because

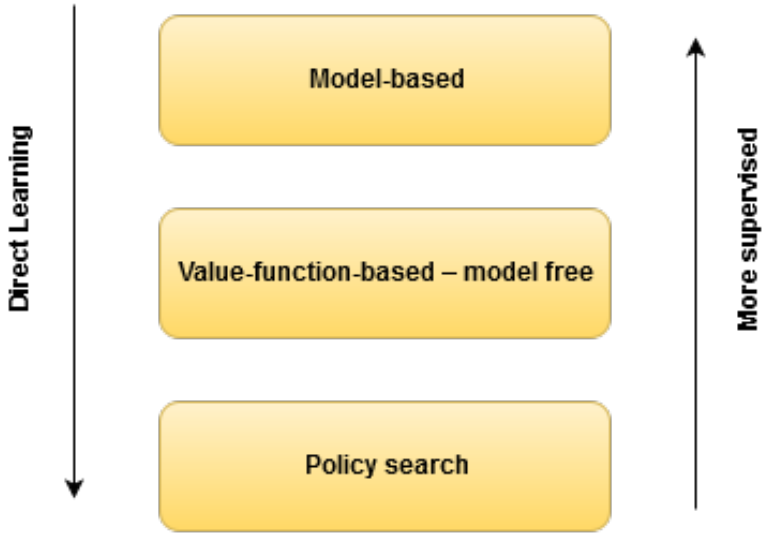


Figure 20: Different families of Reinforcement Learning algorithms.

they have the so called ***exploration/exploitation trade/off***: if a very little *exploration* is performed the algorithm can't optimize the task at a global level, but doing a lot of exploration can leave some important actions that generates a good reward.

Finally, to complete this introduction, we can say that the agent and environment definitions depends on the problem's nature that have to be solved. It's not true, for example, that the agent is physically the robot and that the environment is the outside world. Some times the environment is inside the robot or the agent is an external entity that evaluates the actions of other robots.

3.3.2 Main Reinforcement Learning algorithms

Reinforcement Learning algorithms can be defined using the **M**arkov **D**ecision **P**rocess (MDP) framework. An MDP is defined as a tuple $< S, A, T, R >$ where:

- **S** Is the set of all possible states.
- **A** Is the set of all possible actions.
- **T** Is the state's transition function.

Algorithm	Citation	Sample Efficient	Real Time	Continuous	Delay
R-MAX	Brafman and Tennenholtz, 2001	Yes	No	No	No
Q-LEARNING	Watkins, 1989	No	Yes	No	No
with F.A.	Sutton and Barto, 1998	No	Yes	Yes	No
SARSA	Rummery and Niranjan, 1994	No	Yes	No	No
PILCO	Deisenroth and Rasmussen, 2011	Yes	No	Yes	No
NAC	Peters and Schaal 2008	Yes	No	Yes	No
BOSS	Asmuth et al., 2009	Yes	No	No	No
Bayesian DP	Strens, 2000	Yes	No	No	No
MBBE	Dearden et al., 2009	Yes	No	No	No
SPITI	Degris et al., 2006	Yes	No	No	No
MBS	Walsh et al., 2009	Yes	No	No	Yes
U-TREE	McCallum, 1996	Yes	No	No	Yes
DYNA	Sutton, 1990	Yes	Yes	No	No
DYNA-2	Silver et al., 2008	Yes	Yes	Yes	No
KWIK-LR	Strehl and Littman, 2007	Yes	No	Partial	No
FITTED R-MAX	Jong and Stone, 2007	Yes	No	Yes	No
DRE	Nouri and Littman 2010	Yes	No	Yes	No
TEXPLORE	Questa tesina	Yes	Yes	Yes	Yes

Figure 21: Comparition table of different Reinforcement Learning algorithms.

- **R** Is the *reward* function for every *action-state* couple.

An *MDP* satisfies the *Markov's property*: the previous state and the last action are the only values needed in order to describe the present state and the cumulative reward. This means that the next action to be executed depends only of the last state. All the main Reinforcement learning algorithms are based on this property. The most popular algorithms (and also implemented in the library used in the next section) are:

- **Q-Learning** this algorithm tries to maximize the reward (the mean value of it, for all iterations). Uses a Q table where the rows represent the states and the columns represent the actions. Every value $Q(s, a)$ of this table has a measure of how good is the action **a** for the state **s**. This *measure* is calculated using the following equation:

  $Q(s, a) \Leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \Upsilon.max_{a'}Q(a', s')]$

  Where $Q(a', s')$ is the state/action's value of the next state.

- **SARSA** is a variant of the Q-Learning algorithm. The law that uses to calculate the state-action table is

  $Q(s, a) \Leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \Upsilon.Q(a', s')]$

the difference with *Q-learning* is that the right member of the equation doesn't uses the *max* function and is used only the next state-action. This changes gives the algorithm a fast conversion to the optimal policy on certain conditions.

- **RMAX** this a *model-based* algorithm. It's very simple as implementation and allows to find an optimal reward's value (at least as mean) in polynomial time. The agent builds an environment's complete model (not always accurate) and works on the calculated model's policy in order to handle (with an internal mechanism) the *exploration vs exploitation*'s dilemma.

- **Dyna** as other algorithms it builds a similar table to that of the *Q-learning* algorithm for the state-action pairs. The difference is that this algorithm introduces *planing*. Planing means that Reinforcement learning is applied to *simulated* model's experiences and this experiences are combined with that obtained from the real environment.

- **TEXPLORE** ([26]) this is another *model-based* algorithm that *learns* an specific environment's model called *Random Forest*. The agent explores all the states that seems promising in order to build the final policy, ignoring the non promising states. Internally, this algorithm uses a parallel architecture that allows real time action selection. For this the algorithm is particularly useful to solve *real-time* problems like autonomous vehicle driving.

Figure 21 shows a comparison between the different types of RL algorithms. As we can see the most complete algorithm in that table is the *TEXPLORE* algorithm and this is the Reinforcement learning algorithm that will be used in the next section.

For a proper Reinforcement learning introduction the best source is the Sutton's book [23].

3.3.3 Related works - learning and grasping

The grasping pose generation problem of unknown objects is still an open problem on Robotics. In the last two years were published interesting works that deal with the grasping problem using learning algorithms.

The first work is *Supersizing Self-supervision: Learning to Grasp from 50K Tries and 700 Robot Hours* [21], where deep learning and neural networks are used to solve the grasping problem of household objects. The interesting part of this work is how the grasping action itself is modeled. It models a grasping action as a *planar grasp*. A planar grasp is one where the grasp configuration is along and perpendicular to the workspace. Hence the grasp configuration lies in 3 dimensions, (X, Y): position of grasp point on the surface of table and (θ) angle of grasp.

The second work is [3]. In this work a *Reinforcement learning framework* is proposed to solve the grasping of objects in a clutter problem. The interesting part of this work is that the grasping problem is modeled as a Reinforcement learning problem. It also uses the *planar grasp* definition.

The last work reviewed is the *Antipodal Grasping Identification and LEarning (AGILE)* [6]. This algorithm detects grasps directly from point clouds. Grasping detection is performed using geometrical features of the point cloud. The algorithm search for *antipodal points* inside the point cloud so the calculated grasping pose is referred to that antipodal point. The only assumption is that the gripper is a *simple* two fingers gripper. The algorithm outputs a lots of antipodal points, so to reduce the candidate's list a second phase with a machine learning algorithm (SVM) was implemented. Another interesting characteristic of this work is that it works with 1 or 2 RGB-D sensors as input in order to improve the object's point cloud quality.

The *Agile* algorithm has been selected to generate the grasping points of this Thesis's work. An example of the Agile's output from a virtual kinect is showed in Figure 22.

3.3.4 The Reinforcement learning package

The actual reinforcement learning implementation used on this work, starts with the problem's modeling analysis. Since the actual robot has not a simulation (virtual) environment for the grasping task, the number of trials must be ***minimized***.

Learning is a time consuming task. It's very important to model the problem ***SEEKING*** to reduce the learning time. Model the problem, in the Reinforcement Learning's domain means *define state and actions*.

To reduce the learning time, it's better to learn ***simple*** relations because the richest knowledge derives from the cloud-based ontology database where a crowed of robots load the experiences.

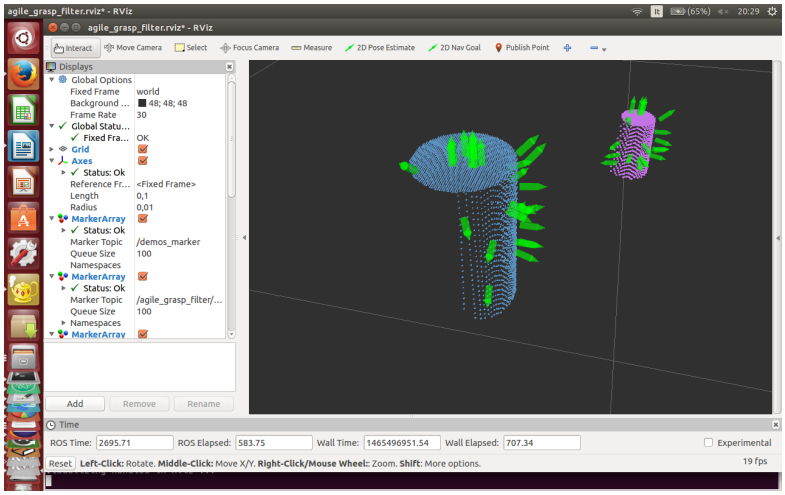


Figure 22: *AGILE*'s algorithm output on RVIZ.

Using the fact that we already have the scene divided into objects. We use a single object as input to the RL algorithm to create the state.

Figure 23 shows the phases of state's creation. The single object's point cloud arrives from the segmentation algorithm, then using the *SIFT* algorithm, features are extracted. Since *SIFT* features vector is very large a method known as ***Principal Component Analysis (PCA)*** is applied to reduce vector's dimensions. The resulting method is a 5 component real vector. This vector will define the **state** of the single object inside the Reinforcement learning algorithm.

The second element that needs the Reinforcement learning algorithm is the action's definition. Figure 24 shows how the actions are defined. The main idea is that the relationship between actions and states must be *simple*. The *Reinforcement learning engine* calls the *AGILE* grasp algorithm giving the single object's point cloud as input. The algorithm returns the list of possible *planar grasps* (X, Y, θ) for that object. Instead of assigning a single action to every (X, Y, θ) of the grasp's list, the algorithm works a follows:

- Only 4 actions are defined, every action takes a grasping parameter if only if it's (θ) value lies in the angle's range.
- The ranges are *from 0 to 90 degrees*, *from 90 to 180 degrees*, *from 180 to 270 degrees* and *from 270 to 0/360 degrees*.

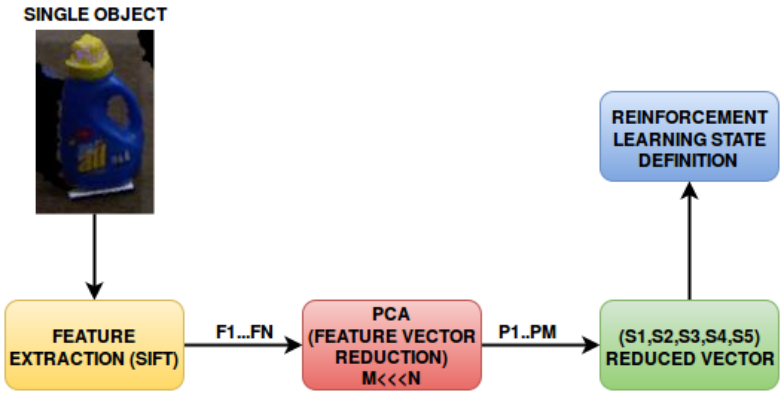


Figure 23: State definition for the grasping task.

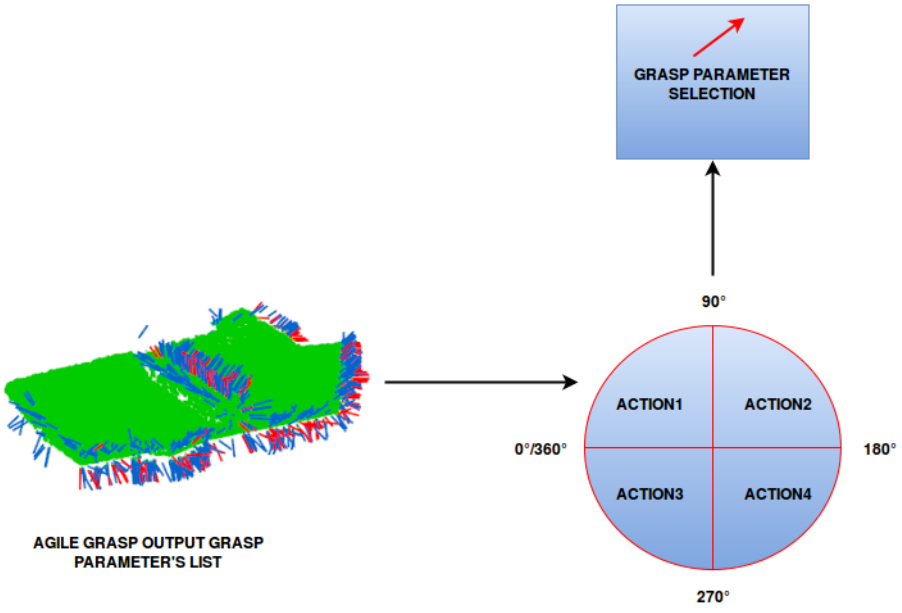


Figure 24: Action definition for the grasping task.

The expected result is that the algorithm will choose the action (the planar grasp) that has a high probability to be successful.

Once the (X, Y, θ) parameters are selected, the RL engine calls the *Act* module that executes the action on the robot.

The feedback if the grasp was done correctly is given by a human operator. The RL engine waits for the feedback in order to understand if continuing to grasp the object selecting another action or finishing if the grasping action ends successfully.

Inside the implemented ROS node, the class that manage Reinforcement learning's states and actions is the ***ModelRlGrasping.cpp*** that calls the Reinforcement learning's environment for the grasping task (another class inside the ROS's *rl-texplore-ros-pkg* package). For the agent element, the TEXPLORE algorithm is used (the implementation is inside the *rl-texplore-ros-pkg* package) .

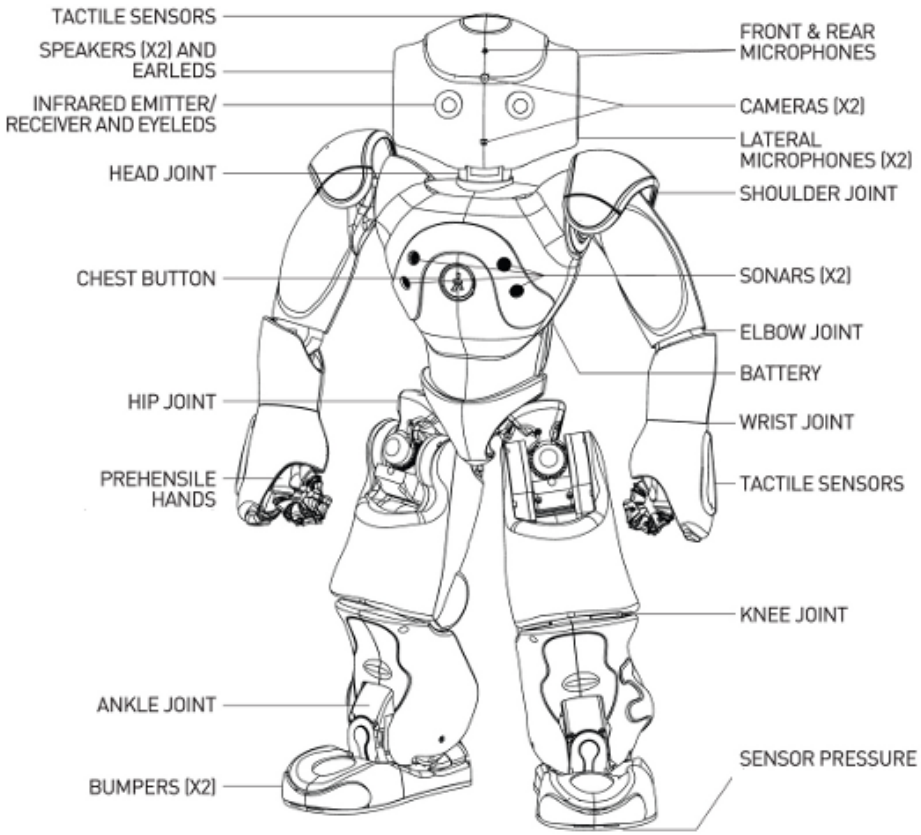


Figure 25: The humanoid Nao, detailing the different robot's parts.

4 The act module

4.1 Hardware - The Aldebaran Nao

The Aldebaran Nao, is a humanoid robot produced by Aldebaran Robotics. Figure 25 shows the robot along with the different parts that composes it (images are courtesy of Aldebaran's site[11]).

The robot has a very articulated body. Each arm has four degrees of freedom giving a workspace similar to the human's arm. Figure 26 shows the degrees of freedom of the robot's arm. The joints use modern servo motors that are controlled by dedicated microcontrollers (*DSPic* microcontrollers).

[11] http://doc.aldebaran.com

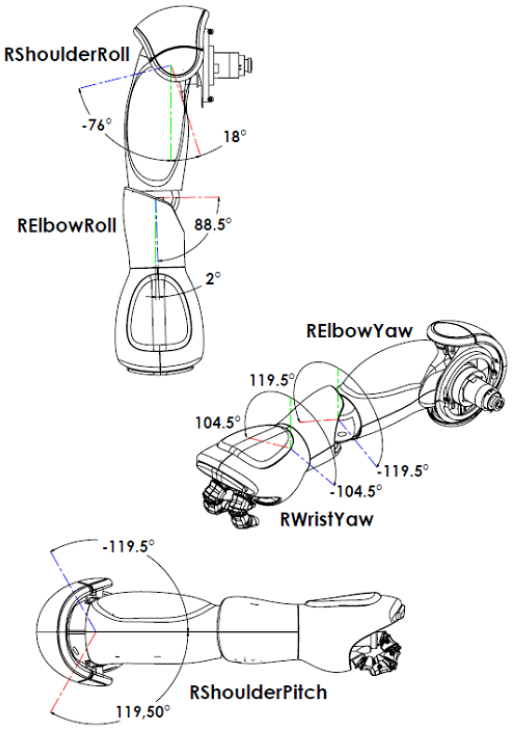


Figure 26: Zoom on the right arm joints, detailing the arm's DOF

Table 1 describes the main hardware characteristics:

Nao humanoid robot (V5 version)	
Height	58 cm
Weight	4.3 kg
Power supply	lithium battery providing 48.6 Wh
Autonomy	90 minutes (active use)
Degrees of freedom	25
CPU	Intel Atom @ 1.6 GHz
Built-in OS	NAOqi 2.0 (Linux-based)
Compatible OS	Windows, Mac OS, Linux
Programming languages	C++, Python, Java, MATLAB, Urbi, C, .Net
Sensors	Two HD cameras, four microphones, sonar rangefinder, two infrared emitters and receivers, inertial board, nine tactile sensors, eight pressure sensors
Connectivity	Ethernet, Wi-Fi

Table 1: Nao's hardware characteristics.

A good Nao hardware's description for the grasping task is given at [12].

4.2 Software - The Moveit! library

The *Act* package's implementation is based on a popular motion planing library: ***MoveIt!***.

MoveIt! [1] is an opensource (*BSD* license) library for motion planning library (fully integrated into ROS) originally developed by the *Willow Garage* team and now maintained by the *Open Source Robotics Foundation (OSRF)*. This library delivers a complete pipeline for motion planning, such pipeline

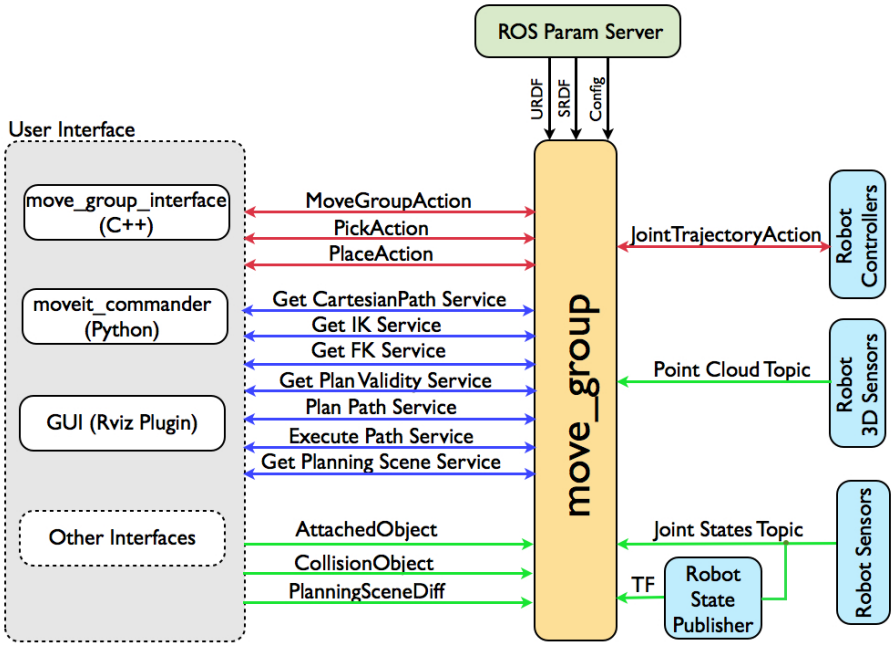


Figure 27: *MoveIt!* pipeline for motion planning.

is showed in Figure 27 (image taken from MoveIt's site). As we can see from the pipeline's image a central part of the library is the ***move_group*** node. This node serves as an integrator between the components of the library in order to provide a set of services and actions. These actions can be accessed using ***C++*** (*move_group_interface*) or ***python*** (*moveit_commander*) programs. The *move_group node* is configured using the parameter server (upper part of the image).

In this pipeline the *ROS param server* search for three elements in the system:

- **URDF** - the *robot_description* parameter on the ROS param server returns the URDF for the robot.
- **SRDF** - *the robot_description_semantic* parameter on the ROS param server returns the SRDF for the robot. The SRDF can be created using the MoveIt! Setup Assistant.
- **MoveIt! configuration** - other configuration information specific to MoveIt! including joint limits, kinematics, motion planning, perception, etc. The configuration files for these components are automatically generated by the MoveIt! setup assistant and stored in the config directory of the corresponding MoveIt! config package for the robot.

The library communicates with the robot (real or simulated) using topics and actions. The topics used by the *move_group* node are:

- *joint_states* topic - using this topic the node retrieves information about the robot's joint like position, state, velocity, etc.
- *Transform Information* topic - needed to retrieve the robot's pose in relation with the real world.
- *Controller Interface* - the node talks to the robot's controller using the *FollowJointTrajectoryAction* ROS action, so this kind of service must be implemented on robot's side in order to allow MoveIt! control.
- *Planning Scene* this monitor serves to maintain an internal world's representation and the robot's current state like objects attached to robot's grippers in order to take account of them when planning the robot's movements. The topics that uses the planing scene are *CollisionObject*,

PlanningSceneDiff and *AllowedCollisionMatrix*. Other topics used by this planning module are the ones produced by RGB-D sensors like the kinect cameras.

Another interesting MoveIt!'s characteristic is that it provides a planning plugin for the *Rviz visualization tool* which can be used to create planing requests with the help of a Graphical User Interface.

Using all this information MoveIt! generates a time parametrized trajectory consisting of a set of way points. A ***waypoint*** is a joint configuration defined by a tuple *(p, v, a, t)*. The four tuple's elements (for n joints) are:

1. **p** - the joint's positions, $p \in \Re^n$.
2. **v** - the joint's velocities, $v \in \Re^n$.
3. **a** - the joint's accelerations, $a \in \Re^n$.
4. **t** - the above vector must be considered at time t, which is a real number.

These way points define the trajectory that every joint must follow along the time.

In order to solve many trajectory generation's problems, MoveIt! uses some plugins components. The most important plugins inside the MoveIt! library are:

- The *kinematics plugin* used to solve the inverse kinematics problem, uses the Kinematics and Dynamics Library (KDL)[12].
- The *planning plugin* uses the Open Motion Planning Library[13], that contains many implementations of motion planning algorithms.

4.3 The Act ROS package

This node was implemented from scratch using the *MoveIt!* library. Figure 28 presents the package's structure. As for the other implemented modules there's a ROS metapackage that contains the other packages.

The main package is ***action_grasping*** package. It contains the interface to control a *Nao* robot. In order to use this package (along with the *Nao* robot), the following ROS packages must be installed:

[12] http://www.orocos.org/kdl
[13] http://ompl.kavrakilab.org

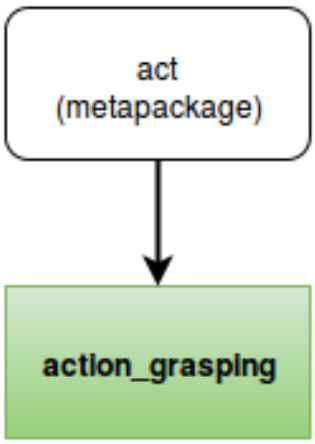


Figure 28: The ***Act*** package's structure.

- **nao_meshes** - contains part of the Nao's description.
- **nao_robot** - the Nao's *URDF* description file.
- **nao_moveit_config** - this package contains the *MoveIt!* configuration for the Nao robot.
- **nao_dcm_robot** - complete package stack containing controllers for ***Real*** Nao robots.

The package contains two main elements that can be used by running the specific launch file.

By launching *action_grasping_nao_test.launch* a test interface is started. With this interface users can execute the following actions using the real or a virtual robot:

- generating random positions of imaginary objects so the Nao tries to reach these positions.
- generating random positions for placing the imaginary object.
- test the positions that the gripper can reach, the *gripper's workspace*.

Figure 29 shows the test program in action, in this image the virtual robot tries to pick up a cylinder.

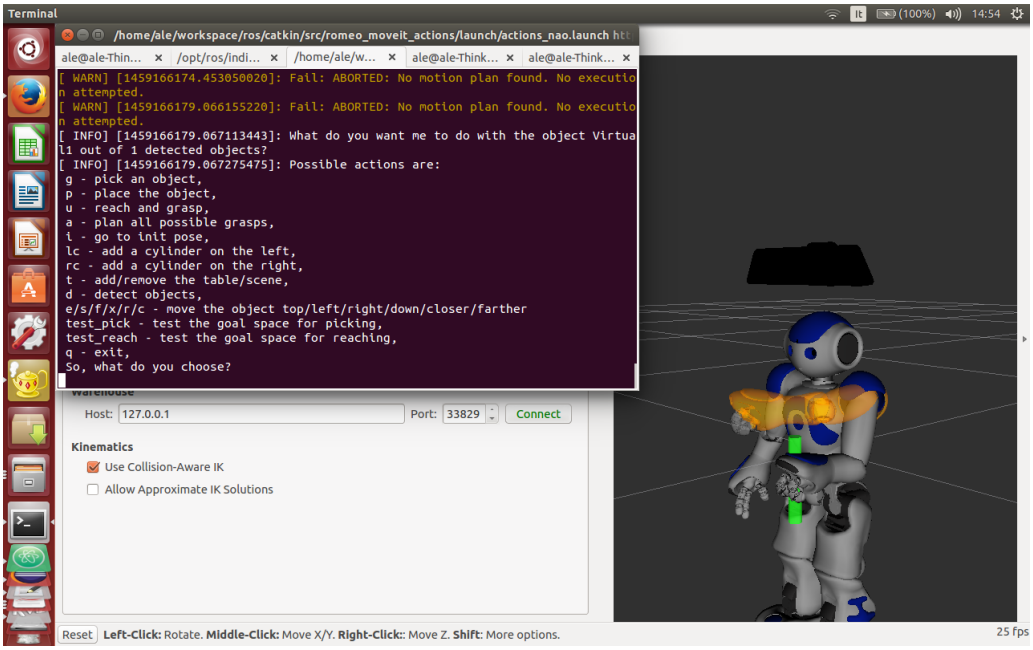


Figure 29: Grasping's simulation using the implemented ***Act*** package.

By launching *action_grasping_nao_service.launch* file a ROS service node is started. The service uses the *grasping_nao_service.srv* file witch defines the service's input and output parameters:

```
string robot_name
string action_type
string arm_name
string plan_name
string end_eff_name
int32 attemps_max_in
float32 planing_time_in
float32 tolerance_min_in
float32 angle_resolution_in
float32 grasp_depth_in
float32 distance_error_allowed
geometry_msgs/Pose goal_pose
---
uint32 success
```

The input parameters include some common information to other services like the robot's name, the action's type, and the gripper's information. Other specific parameters include the maximum attempts, the maximum planing

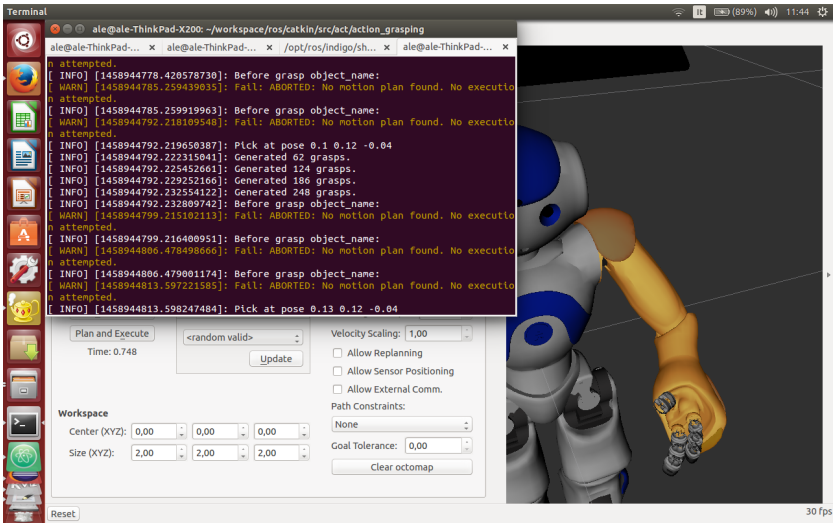


Figure 30: Detail on the interface of the node *Act*.

time in order to solve the trajectory, the angle's resolution (tolerance) and the distance's tolerance. Another important parameter is the goal pose that defines the position of the object's center of mass (CoM) to be grasped. This position is defined as the plane's surface coordinates and the angle of grasp, (X, Y, θ).

The service gives as output a single Boolean value. This value is true if the robot (with the problem's constrains) reach the goal position. So it doesn't give a value about the grasping quality or if the object was actually grasped.

Figure 30 shows a detail of the hand (from the Rviz viewer) when the service is running.

Finally, in order to launch the ***test*** program the following console's commands must be used (one line is a different console's tab):

```
$ roslaunch nao_moveit_config demo.launch
$ roslaunch action_grasping action_grasping_nao_test.launch
```

The ***service*** program can be launched using the following console's commands (one line is a different console's tab):

```
$ roslaunch nao_dcm_bringup nao_dcm_H25_bringup_remote.launch
$ roslaunch action_grasping action_grasping_service.launch
```

If the real Nao is used, then we must follow some steps otherwise the robot could perform some actions that can damage the robot's servo motors:

1. Connect the Nao to the local *LAN*, by pressing the chest's button, the Nao will say the assigned IP.

2. Open a browser's window and type the Nao's IP, the robot's configuration page will be loaded. Inside set the option ***Alive by default*** to **OFF**. This will disable the robot's internal node and is better if the robot receives commands for an external program.

3. Launch the following command on a console (set the Nao's IP inside the *nao_dcm_H25_bringup_remote.launch* file)

```
$ nao_dcm_bringup nao_dcm_H25_bringup_remote.launch
```

4. Turn on the real Nao's node inside ROS by using this console command

```
$ roslaunch nao_bringup nao_full.launch
  nao_ip:=192.168.202.102
  roscore_ip:=127.0.0.1 network_interface:=wlan0
```

5. Turn *ON* the robot's servo motors by using the following ROS service

```
$ rosservice call /nao_robot/pose/body_stiffness/enable
```

6. Start the robot's MoveIT! configuration node by running this console command

```
$ roslaunch nao_moveit_config moveit_planner.launch
```

7. Finally launch the action_grasping service or the action_grasping interface test program using the commands described before.

5 Experiments

Experiments aim to provide timing results on Cloud access and ontological data retrieval and on grasping execution. The Cloud Engine have been integrated inside the Cloud environment of the CORE project [27], which is available on the CloudLab platform [14] cluster under the project ***core-robotics***. It consists of one x86 node running Ubuntu 14.04 with ROS Indigo installed. To test the recognition pipeline a dataset was used to populate the ontology and the related folders, the Object Segmentation Database (OSD) [4]: a data set of 726 MB currently containing 111 different objects, all characterized by a 2D color image and a point cloud. This chapter differentiates the following experiment's types:

- *Recognition engine timing retrieval* - here we are measuring the time that the object takes to arrive to the server and different times of the recognition pipeline as described in Figure 17.
- *Grasping generation timing* - in this experiments the grasp generation time is measured when using the proposed act module.

The client-side ROS nodes run on a lowend *Thinkpad* laptop with an Intel's *Core2Duo* and 4GB of RAM.

5.1 Recognition engine

As explained in the Cloud Engine section the recognition pipeline includes different phases to retrieve similar objects in the *RTASK* Ontology. A important step inside the pipeline is the time that the Engine *employs* to extract the features of the object. The features extracted are the *SIFT* features and the OpenCV's implementation of the SIFT algorithm is used.

Table 2 shows the time that the Cloud engine employs to extract the feature's objects. Using the entire dataset ([4]) the server employs only 45 seconds to extract and write on a binary file the features of 111 objects. But in this case the most interesting time is the second, for one object it employs only .037 seconds. This is the case we have when a new retrieve query arrives from a robot.

Table 3 shows the time that the Cloud engine employs to match the entire dataset image's pairs setting the number of *inliers*. Using *300 inliers*

[14] http://cloudlab.us

Single object and multiple object's features extraction				
	1 Thread	2 Thread	3 Thread	4 Thread
111 images	45.1554 s	21.8381 s	15.331 s	12.7545 s
1 image	0.379014 s	0.408295 s	0.396394 s	0.387422 s

Table 2: Feature extraction time for a single and a group of objects.

Features Matching		
	Time	Matches on 6105 possible image pairs
15 inliers	4.82 s	1592
300 inliers	4.48500 s	18

Table 3: Feature matching time for a group of objects.

Round response and request time	
Object	Request and response time
object 1	5.531 s
object 2	4.995 s
object 3	4.350 s
object 4	6.014 s

Table 4: Round trip time for object retrieval.

the match is very accurate and is only 0.003 seconds lower than using a much less accurate matching of 15 inliers.

The last table 4 shows the *round trip time* employed by the entire recognition pipeline to query (for an object) and response to the robot. It includes the transmissions times, the queries times and the object matching times.

Grasp generation and execution time	
Object	Generation and execution time
object 1 - pen	10.501 s
object 2 - ball	15.285 s
object 3 - bottle	18.235 s
object 4 - cylinder	12.101 s

Table 5: Grasp action, execution time for a set of objects.

5.2 Grasping experiments

Table 5 shows the time that the *Act* module employs to generate the solve and execute the grasp position having as input a planar grasp position (X, Y, θ). This trials were executed on the virtual and on the real Nao robot. Figure 31 shows the real Nao preparing trying to grasp a pen.

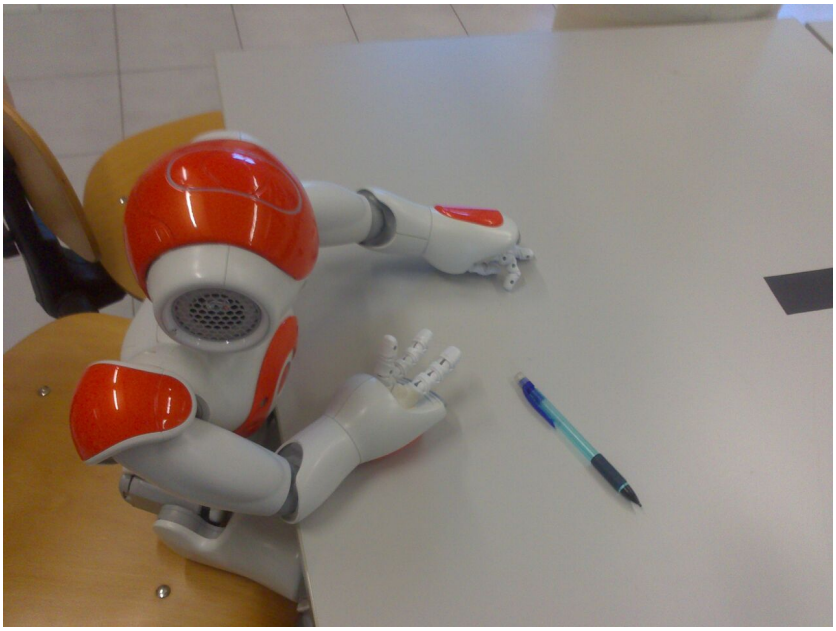


Figure 31: Real Nao robot before performing the grasping task.

6 Conclusions and further work

This Thesis presented a Framework able to increase robots knowledge and capabilities on objects manipulation. The Framework is composed of a Cloud-based Engine, an OWL Ontology and a Reinforcement Learning engine. From the study of human actions when handling objects, the Ontology formulates a common vocabulary that encodes the robotics manipulation domain. The Engine, instead, was developed in order to transfer the computation on the Cloud: it off-loads robot CPUs and speeds up the robots learning phase. Given an object in the scene, the Engine retrieves its visual features and accesses the Ontology in order to extract the corresponding manipulation action. If no information is stored, a Reinforcement Learning technique is used to generate the gripper manipulation poses that will be stored in the Ontology.

The Reinforcement learning approach offers an alternative to popular methods like ***deep learning***. The method has two main vantages:

- there's no need to construct a huge training set, we are working on the concept of *growing dataset*, the robot starts acting from the beginning

without the need of a training phase.

- learning is always present, every input changes the robot's internal world representation. The long-term model of the world is stored on the ontology database and shared with others robots.

An extension of the Thesis's work could involve:

- Using a bigger robot for the grasping task like the Universal Robotics's ***UR-10***.
- Using a different segmentation method, capable of working with smaller point clouds. So the manipulation task can be tested on small objects.
- Using the entire pipeline on a *group* of robots in order to demonstrate the scalability of the Cloud-based engine.
- Performing large-scale experiments using the implemented Reinforcement learning algorithm, to understand how fast the robot learns to grasp unknown objects.

Finally the complete pipeline source's code will be released inside the *IAS-LAB* repository[15] under the project's name ***cloud_based_rl***.

[15] https://github.com/iaslab-unipd

7 Ringraziamenti

Ringrazio di cuore alla mia famiglia per l'appoggio che mi ha dato in tutti questi anni, Maximiliano, Alberto e Isabel. Un rigraziamento speciale a Hortensia e Sita.

Ringrazio al Professore Enrico Pagello e alla Dott.ssa Elisa Tosello per l'opportunità, che mi hanno dato, di sviluppare un lavoro di ricerca interessante e innovativo, e per tutti i consigli che mi hanno dato nel corso dello sviluppo della Tesi.

Ringrazio infine tutti gli amici per tutto quello che abbiamo condiviso negli anni, Marco, Giuliano, Marco, Giuseppe, Duilio, Massimo.

Ringrazio a Sabrina e Fabiana, per tutto quello che abbiamo condiviso negli anni e per la mano con le correzioni della Tesi stessa.

Infine ringrazio tutti i colleghi di lavoro per l'aiuto e la pazienza che hanno portato in questo periodo di studio.

References

[1] I. sucan and s. chitta, moveit! [Online] Available: http://moveit.ros.org.

[2] Official wiki of ros middleware. http://wiki.ros.org/.

[3] J. Bagnell A. Boularias and A. Stent. Learning to manipulate unknown objects in clutter by reinforcement. *AAAI*, 2015.

[4] J. Prankl M. Zillich A. Richtsfeld, T. Morwald and M. Vincze. Segmentation of unknown objects in indoor environments. *IROS, pages 4791–4796*, 2012.

[5] J. Prankl M. Zillich A. Richtsfeld, T. Morwald and M. Vincze. Point cloud culling for robot vision tasks under communication constraints. *Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 4791–4796*, 2014.

[6] R. Platt. A. ten Pas. Using geometry to detect grasp poses in 3d point clouds. *International Symposium on Robotics Research (ISRR), Italy*, 2015.

[7] J. Papon C. Stein, M. Schoeler and F. Woergoetter. Object partitioning using local convexity. *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, 2014.

[8] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. *IIEEE Computer Society Conference on Computer Vision and Pattern Recognition, Vol. 1, pp. 886–893*, 2005.

[9] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. *IIEEE Computer Society Conference on Computer Vision and Pattern Recognition, Vol. 1, pp. 886–893*, 2005.

[10] A. Gatto Castro E. Tosello, Z. Fan and E. Pagello. Cloud-based task planning for smart robots. *14th IAS — International Conference on Intelligent Autonomous Systems*, 2016.

[11] M. Fischler and R. Bolle. Random sample consensus. a paradimg for model fitting with applications to image analysis and automated cartography. *Communications of the ACM Volume 24, Number 6*, 1981.

[12] G. Cotugno H. Mellmann. Dynamic motion control: Adaptive bimanual grasping for a humanoid robot. *Fundamenta Informaticae 112, 89-101*, 2011.

[13] J. Wu H. Cui J. Cheng, C. Leng and H. Lu. Fast and accurate image matching with cascade hashing for 3d reconstruction. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[14] R. B. Rusu M. Gedikli S. Beetz J. Kammerl, N. Blodow and E. Steinbach. Real-time compression of point cloud streams. *Proceedings of the 2012 IEEE International Conference on Robotics and Automation (ICRA), pp. 778–785.*, 2012.

[15] M. Schoeler J. Papon, A. Abramov and F. Woergoetter. Voxel cloud connectivity segmentation - supervoxels for point clouds. *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, 2013.

[16] B. Karan. Calibration of kinect-type rgb-d sensors for robotic applications. *FME Transactions*, 2015.

[17] D. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision, 60, 2, pp. 91-110*, 2004.

[18] Paul Malmsten. Object discovery with a Microsoft Kinect, 2012.

[19] G. Alenyà N. Covallero. Grasping novel objects. Technical Report IRI-TR-16-01, Universitat Politècnica de Catalunya -Institut de Robòtica i Informàtica Industrial (IRI), May 2016.

[20] R. d'Andrea O. Zweigle, R. van de Molengraft and K. Haussermann. Roboearth: connecting robots worldwide. *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human. ACM, pp. 184–191*, 2009.

[21] L. Pinto and A. Gupta. Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours. *CoRR*, abs/1509.06825, 2015.

[22] N. Blodow R. B. Rusu and M. Beetz. Fast point feature histograms (fpfh) for 3d registration. *Robotics and Automation, (ICRA). IEEE International Conference on,pages 3212 –3217*, 2009.

[23] A. Barto R. Sutton. Reinforcement learning: An introduction. *Free online version from https://webdocs.cs.ualberta.ca/ sutton/book/the-book.html*, 2015.

[24] Craig I. Schlenoff. An Overview of the IEEE Ontology for Robotics and Automation (ORA) Standardization Effort. In *Standardized Knowledge Representation and Ontologies for Robotics and Automation, Workshop on the 18th*, pages 1–2, Chicago, Illinois, USA, 2014.

[25] Chris Sweeney. Theia multiview geometry library: Tutorial & reference. http://theia-sfm.org.

[26] P. Stone T. Hester. The open-source texplore code release for reinforcement learning on robots. *RoboCup-2013 Robot Soccer World Cup XVI Springer Verlag*, 2013.

[27] J. Spruth W.J. Beksi and N. Papanikolopoulos. Core: A cloud-based object recognition engine for robotics. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015.