Master's Degree in
Automation, Control and Robotics

Action Intention Recognition for Proactive Human Assistance in Domestic Environments

Master's Thesis

Author: Javier Gallostra Acín
Director: Joan Aranda López

Call: June 2019

Escola Tècnica Superior

d'Enginyeria Industrial de Barcelona

Action Intention Recognition for Proactive Human Assistance in Domestic Environments

Javier Gallostra Acín

Master's Thesis in Automation, Control and Robotics

Abstract

The current Master's Thesis in Automatics, Control and Robotics covers the development and implementation of an Action Intention Recognition algorithm for proactive human assistance in domestic environments. The proposed solution is based on the use of data provided by a real time RGBD Object Recognition process which captures object state changes inside a defined region of interest of the domestic environment setup.

A background analysis is performed to analyze state of the art approaches to both real time RGBD object recognition and action intention recognition methods. The preliminary analysis serves as the base for the proposal of a new volume descriptor for object categorization and an improved formalism for Activation Spreading Networks in the context of action intention recognition.

Several tests are performed to study the performance of the proposed solution and its results are analyzed to define the conclusions of the project and propose future work. Finally, the project budget and environmental impact as well as the project schedule are presented and briefly discussed.

Contents

Abstract i

Acronyms ix

List of Figures xi

List of Tables xv

I Introduction 1

1 Objectives 2

2 Scope 3

II State of the Art Analysis 5

3 RGBD Object Recognition 6

3.1 Category and Instance Object Recognition 6

3.2 Object Descriptors 7

3.3 Recognition Methods 7

4 Action Intention Recognition **8**

4.1 Recognition Approaches 8

4.2 Project Approach 9

III System Setup 11

5 Overview **11**

6 Robots **13**

6.1 Mico 13

6.2 Baxter 14

6.3 CAPDI 14

7 Other Hardware **14**

7.1 Kinect One 15

7.2 Server PC 15

8 Software **15**

8.1 ROS 16

8.2 RViz 17

8.3 MoveIt! 17

8.4 OpenCL & OpenCV 18

8.5 IAI-Kinect & libfreenect2 18

8.6 Point cloud library 18

8.7 Docker 19

IV Object Recognition **21**

9 Camera Input **21**

9.1 Camera Setup 22

9.1.1 Placement 22

9.1.2 Calibration 23

10 Algorithm Overview **25**

10.1 Nodelet Structure 27

10.2 Calibration nodelet 30

10.3 Plane Segmentation Nodelet 32

10.4 Surface Segmentation Nodelet 37

10.5 Object Segmentation Nodelet 39

10.6 Object Recognition Nodelet 40

10.6.1 Recognition methods 44

10.6.2 SVM Classification 44

10.6.3 Histogram Comparison Classification 45

10.6.4 Descriptors 45

10.6.5 Two stage classification 48

10.7 Dispatcher Node 50
10.7.1 Message Synchronization 51
10.8 ROS Graph 52

V Action Intention Recognition 55

11 Activation Spreading Networks 56

11.1 Structure 56
11.2 Activation and recognition procedures 57
11.3 Performance indicators 58
11.4 Limitations and improvements 59

12 Autonomous Activation Spreading State Networks 59

12.1 A2SN structure 59
12.2 Autonomous creation of A2SN 60
12.3 A2SN decision function 61
12.3.1 Node Depth 61
12.3.2 Relative Value between Graphs 63
12.3.3 Minimum Value Threshold 63
12.3.4 Final Decision Function 63
12.4 Training A2SN networks 64
12.4.1 Performance Metrics 65

13 Algorithm Implementation **66**

13.1 Auxiliary classes 66

13.1.1 State 66

13.1.2 KitchenObject/s 67

13.1.3 SequenceGenerator 69

13.2 A2SN implementation 71

13.2.1 A2SN_BASE 72

13.2.2 A2SN_BUILD 74

13.2.3 A2SN_RUN 74

Input handling 75

Activation message flow 77

Activation values evolution 77

Consecutive Action Recognition 78

A2SN_RUN Structure 79

13.2.4 Running the Action Intention Recognition 80

VI Results **83**

14 Object Recognition Results **83**

14.1 Recognition by Volume and Color 84

14.2 Algorithm Performance 87

14.3 Test Parameters 89

15 Action Intention Recognition Results **89**

15.1 Testing database 90

15.2 Test process 90

15.3 A2SN recognition results 91

15.4 A2SN performance results 91

VII Project Schedule and Budget **95**

16 Schedule **95**

17 Budget **97**

VIII Environmental Impact **101**

IX Conclusions **103**

18 Future Work **104**

References **105**

Acronyms

- 2D - *Two Dimensional*
- 3D - *Three Dimensional*
- A2SN - *Autonomous Activation Spreading State Network*
- AASSN - *Autonomous Activation Spreading State Network*
- ACOD - *Average Confidence of Detection*
- ASN - *Activation Spreading Network*
- BOE - *Boletín Oficial del Estado*
- COD - *Confidence Of Detection*
- CPU - *Central Processing Unit*
- CV - *Computer Vision*
- DOF - *Degree(s) Of Freedom*
- ECTS - *European Credit Transfer and Accumulation System*
- EDR - *Early Detection Rate*
- ETSEIB - *Escola Tècnica Superior d'Enginyeria Industrial de Barcelona*
- FOV - *Field Of View*
- FSM - *Finite State Machine*
- GPU - *Graphics Processing Unit*
- GRINS - *Grup de Robòtica Intel·ligent i Sistemes*
- HMM - *Hidden Markov Model*
- HSV - *Hue, Saturation, Value*
- HTN - *Hierarchical Task Network*
- ICP - *Iterative Closest Point*

- IoT - *Internet of Things*
- IR - *Infrared*
- ISS3D - *Three Dimensional Intrinsic Shape Signatures*
- LCD - *Liquid Crystal Display*
- ML - *Machine Learning*
- NN - *Neural Network*
- OS - *Operating System*
- PC - *Personal Computer*
- PCL - *Point Cloud Library*
- RANSAC - *Random Sample Consensus*
- RBF - *Radial Basis Function*
- RGB - *Red, Green, Blue*
- RGBD - *Red, Green, Blue, Depth*
- ROI - *Region Of Interest*
- ROS - *Robot Operating System*
- SAC - *Sample Consensus*
- SIFT3D - *Three Dimensional Scale Invariant Feature Transform*
- SVGA - *Super Video Graphics Array*
- SVM - *Support Vector Machine*
- ToF - *Time of Flight*
- UPC - *Universitat Politècnica de Catalunya*
- USB - *Universal Serial Bus*
- VFH - *Viewpoint Feature Histogram*

List of Figures

5.1 General laboratory view 12
6.1 Commercial robots 13
8.1 ROS related software 16
8.2 CV & ML software 17
8.3 Point cloud processing software 18
8.4 Operating system virtualization software 19
9.1 Kinect One cameras' FOV 22
9.2 Raw Kinect One point cloud 23
9.3 Raw Kinect One point cloud (viewpoint 2) 24
9.4 Raw Kinect One point cloud (viewpoint 3) 24
9.5 Ideal Pinhole Camera model 25
10.1 Scheme of the object recognition pipeline 28
10.2 Workflow of the node implementation as a nodelet 29
10.3 Registrator and Preprocessor class structures 31
10.4 Calibration node sample output 33
10.5 Volume of interest 33
10.6 PlaneSemgentation and Plane class structures 35
10.7 Plane Segmentation node sample output 36
10.8 SurfaceSegmentation and Surface class structures 38

10.9 Surface Segmentation bounding boxes sample output 39

10.10ObjectSegmentation and Object class structures 41

10.11Object Segmentation bounding boxes sample output 42

10.12Recogniser class structure 45

10.13SVM and Histogram comparison class structures 46

10.14FeatureExtraction class structures 47

10.15Object Recognition sample output 50

10.16Dispatcher Node sample occupancy grid output 52

10.17Object Recognition pipeline node graph 54

12.1 Sample A2SN graph building sequence 62

12.2 A2SN Maximum Node Value evolution with different parameters 64

13.1 State class structure 68

13.2 KitchenObject class structure 68

13.3 SequenceGenerator class structure 71

13.4 Sample A2SN values evolution over time 76

13.5 Action Intention Recognition process structure 81

14.1 Normalized Confusion Matrix of the Object Recognition results 85

14.2 HSV color space cone 86

14.3 Recognition of objects in a cluttered scene 87

14.4 Object Recognition processing time 88

14.5 Sample object images 89

15.1 A2SN Maximum Node Value evolution during an Action Recognition Process - cereals 92

15.2 A2SN Maximum Node Value evolution during an Action Recognition Process - coffee 92

15.3 A2SN execution time versus total number of nodes 93

16.1 Project Gantt chart 96

16.2 Difference between programmed and real hours for project tasks 96

16.3 Difference between programmed and real hours for each month 97

List of Tables

5.1 Laboratory elements description 11

10.1 Calibration node characteristics 30

10.2 Calibration node paramenters 32

10.3 Plane Segmentation node characteristics 34

10.4 Plane Segmentation node parameters 35

10.5 Surface Segmentation node characteristics 37

10.6 Surface Segmentation node parameters 39

10.7 Object Segmentation node characteristics 40

10.8 Object Segmentation node parameters 42

10.9 Object Recognition node characteristics 43

10.10Dispatcher node characteristics 50

14.1 Object list categorized by volume 84

14.2 Object Recognition average processing time and rate 88

14.3 Object Recognition test parameters 89

15.1 Statistical Action Recognition Results 91

17.1 Amortization cost of project hardware 98

17.2 Energy cost of the project 99

17.3 Project Budget 99

Part I

Introduction

Robotics is one of the scientific fields which has grown faster [19] and raised a huge public interest in the past few years. Together with the IoT, it is envisioned as the next industrial revolution step and investors around the world are supporting all kinds of robotic companies which are expected to increase productivity and efficiency in every company. As close as we are to building completely robotized factories with an impressive production capacity, there is another side to the robotics story which is still in its earliest stages.

This second field of robotic applications is considerably more ambitious and has far more life-changing potential than the industrial robot systems. Known by the name of Service Robotics or Assistive Robotics, it matches the lifelong science fiction concept of robots: complex humanoid machines whose purpose is to assist humans in their daily tasks for the progress of society and the pursuit of a better world. Reality is however far from these great expectations.

The AURORA Project is part of the scientific effort to bring robots closer to people. Its goal is to explore the capabilities of state of the art technologies and apply them in robotized environments to create assistive solutions in order solve daily problems for the people in need of them. It began at the UPC Control Department in the GRINS research group and has offered a research environment for students, teachers and doctors who have contributed with their work and effort to the robotics community. This Master's Thesis has been developed as one more step of the ambitious AURORA project.

The vision that drives AURORA is to explore and implement robotic solutions in a real world scenario to assist disabled people in their daily kitchen tasks. It has several research lines such as human-machine interaction, robotic task planning or responsive robot movement. This Thesis is focused on human-machine interaction and it aims at developing and testing an action recognition algorithm that can provide knowledge about the user's intentions without active user input.

The report is organized as follows. In the first chapter, an introduction to the project as well as its objectives and scope are explained. Secondly, two chapters are devoted to the state of the art analysis and to describe the system setup as well as each of its individual components. The fourth chapter deals with the computer vision object detection algorithm. Then the action recognition algorithm is explained in the fifth chapter. The

sixth chapter describes the tests performed and analyzes the results. Finally, the last three chapters are devoted to the economic and environmental impact of the project and to draw conclusions and propose future work for the project.

1. Objectives

The main objective of the project is to develop and test a vision based action recognition algorithm to detect human intentions in real time under specific conditions in a kitchen environment. This general objective can be subdivided into:

- Carry out a state of the art analysis in order to choose the most suitable approach to solving the computer vision and action recognition tasks
- Develop a computer vision object recognition algorithm under the ROS framework that
  - robustly detects a defined set of kitchen objects
  - precisely positions the detected objects in a given world reference frame
  - works in real-time
- Develop an action recognition algorithm based on the computer vision object detection that
  - differentiates a set of human kitchen tasks
  - detects the correct task while it is being performed (i.e. detects the human intention: which task the user is performing)
  - provides information about the current task to the robotic system in order to suggest assistive actions
- Document the developed material to enable its use in future applications
- Perform real world tests to analyze the results and propose future work and improvements

2. Scope

The project is considered successful if the final system is able to recognize from a defined set the task that is being performed using only the computer vision information in real time. The success rate of the system does not need to achieve a perfect recognition score for the project to be satisfactory.

Part II

State of the Art Analysis

The analysis of the state of the art is divided in two parts, following the project development structure: object recognition and action intention recognition. Of these two topics, object recognition (sometimes referred as identification) has been widely studied and considerable progress has been made by the scientific community since the first time the problem was faced, although there does not exist a lightweight, flexible and accurate enough solution. On the other hand, action intention recognition is a relatively new topic which still needs years of research in order to provide robust results.

Among the many challenges posed by these two recognition objectives, the main one is that the number of classes to recognize greatly increases when trying to escalate a particular solution. This makes the development of a universal recognition algorithm an almost utopian task, leading many researchers to limit the scope of their studies.

Both problems have similar characteristics: their object of study tries to classify an endless amount of individual entities which often have a high degree of inter class and even intra class variability. Moreover, the most common approaches to find a reliable recognition algorithm for these problems are based on the same type of data: RGB images, 3D data or a combination of both. Although the processing of 2D and 3D image data is a computationally heavy task, the processing power of computers has increased exponentially over the years enabling the use of larger databases for recognition purposes. Taigman et al. [30], for example, used a database of four million faces belonging to roughly 4000 classes in order to train a neural network with more than 120 million parameters; and achieved a recognition success rate higher than 97,3%.

As promising as these results may be, they prove that nowadays the amount of data, the hours of work and the computer power needed to achieve such success rates is only within the reach of very few research centers in the world. Regarding the scope of this project, several assumptions and restrictions have been made in order to provide the minimum setup required to validate the theoretical proof of concept of the project. Despite the scope and field of application restrictions imposed by these assumptions, if the solution proves to be scalable it could be enhanced for its use in fields other than the one of the project.

3. RGBD Object Recognition

The launch of low cost RGBD cameras for commercial purposes in the recent years has also opened many possibilities for researchers to develop, implement and test computer vision algorithms using RGBD data. In the field of object recognition, where the recognition task was traditionally performed using RGB images, the addition of a 3D component to the input data has also enabled a new wave of algorithms. Using a combination of depth and color information the recognition success rates have improved over time.

Lai et al. [21], for example, showed that the addition of depth data to traditional color based object recognition improved the recognition results by 10 to 15%. When recognizing object categories they reached a peak of 84% recognition accuracy, whereas when recognizing objects individually the accuracy obtained was around 74%. Additional works such as those by Bo et al. [4] and Blum et al. [3] also present results that range from 70 to 90% accuracy. In the current project, the approach taken also combines color and depth for recognition.

3.1 Category and Instance Object Recognition

On the aforementioned works the same results are presented twice: once for category and then for instance recognition. The distinction between category and instance (individual) recognition is relevant as it affects the results on objects from the same database and influences the recognition algorithm approach.

From a recognition point of view, it is neither better nor worse to pursue one goal or the other. For example, an instance object recognition of a database where objects have large intra category variability, like chairs, will perform better than recognizing categories on the same data. On the other hand, a database with low intra category variability like soda cans will provide better results when recognizing categories than when trying to recognize instances. The purpose of the recognition outcome is what decides whether instance or category recognition should be pursued.

In this project the object recognition focuses on instance recognition, due to the fact that objects with similar characteristics have a completely different meaning for action recognition. This is the case, for example, of a milk and a juice brick. Therefore the object

database has been built creating a single class for each distinct object in the considered set.

3.2 Object Descriptors

In any recognition procedure the classification algorithm used is as crucial as the choice of adequate feature descriptors. A suitable classification function with non discriminant features does not have any chance of properly classifying the input data. This has led to a continuous research to find descriptors which are as small as possible while still maintaining the highest discriminant capabilities (see the aforementioned Bo et al. [4] and Blum et al. [3]).

With RGBD data the most common approach for classification is to separately compute a color and a depth descriptor and merge them afterwards, as in the work of Gupta et al. [7], or to just consider the information from color images (Saffar et al. [26]). One common feature of almost all the color based object recognition algorithms is the usage of the HSV color space for feature descriptor extraction, as mentioned by Soleimanizadeh et al. [29], Chapelle et al. [6] and Mazzeo et al. [22]. The color space used in this project for object recognition is also HSV.

As for volume object descriptors, the work by Alexandre [1] in 2014 presents relevant results based on an in depth review of 3D object descriptors for a real RGBD object dataset. The results show that ISS3D and SIFT3D outperform the rest of descriptors when used for object recognition. However, the computation time required to compute them makes them unusable for a real time application. As an alternative Rusu [25] proposed the VFH descriptor, a local geometry based descriptor which gathers information about the relative normal directions of the points in the cloud. VFH is proposed as a valid object recognition descriptor capable of running in real time applications. This volume descriptor was tested for the project and based on it a new descriptor was created.

3.3 Recognition Methods

When limiting the scope of recognition methods to the ones suitable for real time applications, two of them stand out from the others. These are the use of previously trained SVMs or classification using Histogram distance comparison (refer to the work by Saffar et al. [26]). In the current project, both methods have been tested in order the choose

the one most suitable to the gathered training data. The SVM used has a RBF kernel and the Histogram distance metrics tested are: Correlation, Chi-Square, Intersection and Bhattacharyya distances.

4. Action Intention Recognition

The field of Action and Intention Recognition is younger than Object Recognition but also has encouraged the proposal of a wide variety of algorithms and methods to detect and recognize human actions. The efforts of these works have a common goal: to provide an intelligent system with knowledge about human actions and intentions in order to provide a tailored assistance to users. The fields of application of this techniques ranges from daily household tasks to robbery detection, which gives an idea of the relevance of this research topic for the future of robotics and intelligent systems.

4.1 Recognition Approaches

When analyzing the state of the art for action recognition the proposed methods can be categorized using three different characteristics:

1. **Action/Intention Recognition**: this difference might seem trivial but it affects the core objective and development of the recognition algorithm. As mentioned in the article by Kelley et al. [20], the main difference between both objectives is the temporal factor introduced in Intention Recognition. Unlike normal Action Recognition algorithms, its goal is not quite to classify observed sequences into actions but rather to identify the intentions of a human while performing an action. This implies an online real time processing pipeline which tries to detect intentions in order to predict future actions and eventually offer assistance.

2. **Statistic/Heuristic approach**: many action recognition algorithms are based on hierarchical statistic processes such as Bayesian Networks or HMMs (see Saponaro et al. [27] and Kelley et al. [20]), but the adaptation of these approaches to online recognition is still an unsolved problem. Moreover, they are mainly focused on gesture recognition which is not the case of the project. Heuristic approaches,

however, are simpler and mostly based on learning sequences, but perform well for concealed applications and most importantly can be easily adapted to work for online recognition. Some example works are those by Aranda and Vinagre [2] or Saffar et al. [26].

3. **Movement/Interaction based actions**: as mentioned before, most of the revised works focus on human movement and gesture recognition. The field of human actions which involve very small movements with a high object interaction has been scarcely explored by the research community. In the case of the project, the fact that the system is built around the goal of helping disabled people and that the volume of interest for actions is a static kitchen, the way to classify and recognize human action necessarily involves analyzing the human object interactions.

4.2 Project Approach

This project focuses on developing a new approach to an Action Intention Recognition algorithm, working on a heuristic sequential approach based on human object interaction by object recognition. Among the works reviewed, two of them proposed suitable ideas and apporaches that could help achieve the project goals. The first one is the one by Aranda and Vinagre [2], where they used object queues in different object states as the base for intention recognition. The second article is written by Saffar et al. [26], where they start from an HTN interpretation of actions and propose their translation into ASNs for action intention recognition.

Both of these articles present characteristics that suit the objectives and scope of the project: the learning database is relatively small, the actions are represented as human object interactions and the approach taken is sequential and heuristic.

The present project presents a modified version of ASN for action recognition which includes some concepts proposed by Aranda and Vinagre such as object state sequences. The main objectives behind proposal of the new approach are

1. Build an autonomous system that can learn new action sequences by a supervised learning method. This improves current ASNs as there is no need for a manual generation of the graphs in order to run the recognition process.

2. Remove the task hierarchy present in previous recognition methods, as it introduces a higher degree of abstraction which is not easily learned by self taught recognition algorithms.

3. Simplify the action intention recognition process to enable the use of machine learning methods for training the recognition algorithm parameters and discard the use of hard coded or experimental ones.

Part III

System Setup

5. Overview

The robotic system used in this project has a certain setup which must be taken into account as it conditions the development of solutions and algorithms. It is located in Lab 204 of the K2M building. The system is built around a kitchen table that has a ceramic hob and a sink, placed by a large piece of furniture with drawers and shelves. The area of interest for the project is the table surface and the shelves located both left and right to the user, located in front of the table. This space is where the user will perform the tasks and where the robotic assistive actions will take place.

An overview of the whole system can be seen in Figure 5.1. In the picture there are some highlighted elements, described in Table 5.1.

Color	Description
Red	CAPDI robot
Yellow	Kinova Mico robot
Blue	Baxter robot
Green	Kinect One camera
Orange	Projector
Purple	TV server monitor

Table 5.1: Description of the highlighted elements in Figure 5.1

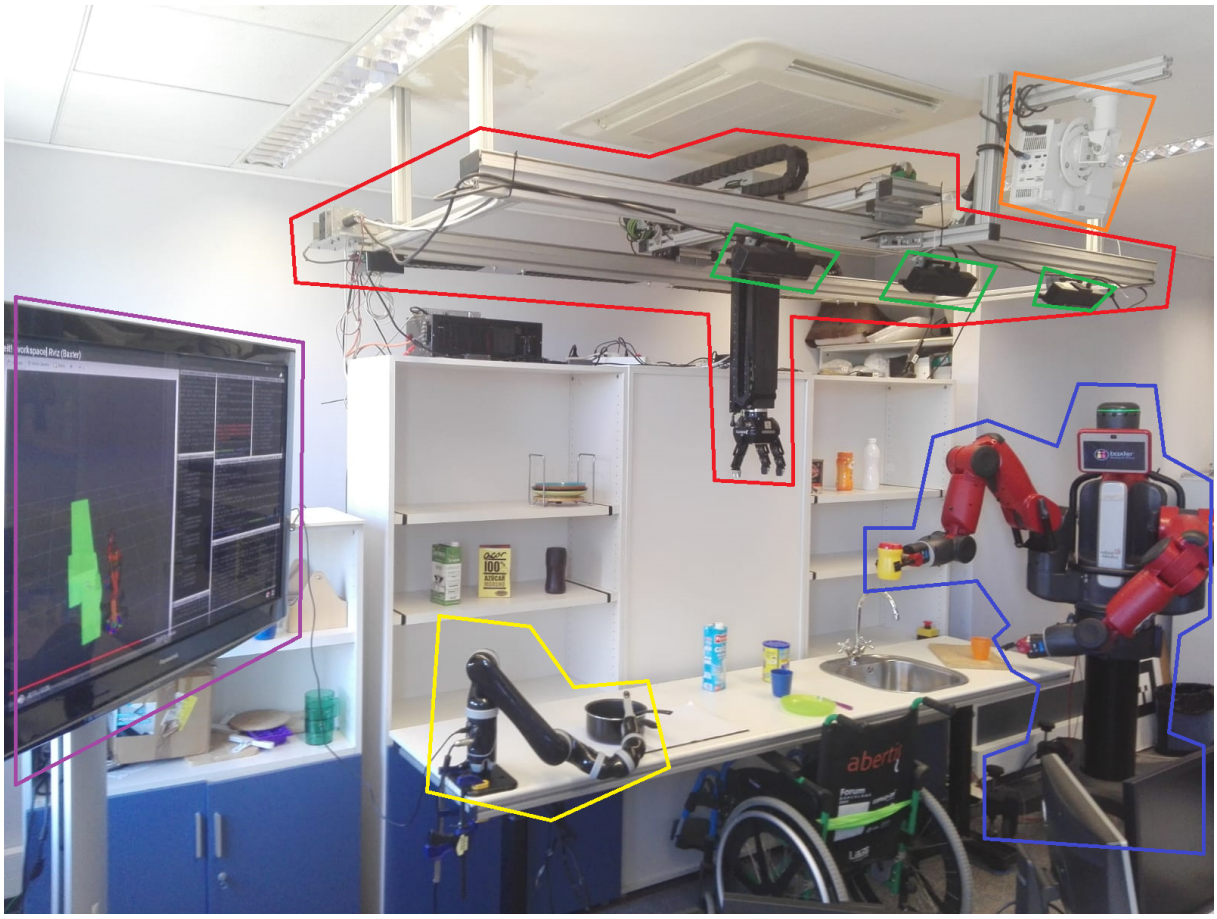


Figure 5.1: General view of the laboratory with the kitchen furniture and the robotic system.

6. Robots

Three robots are used to interact with the user: Mico (Kinova Robotics), Baxter (Rethink Robotics) and CAPDI. Both Mico and Baxter are collaborative robots and all three can be controlled using ROS commands and programs. The next subsections provide a more in depth description of each robot.

(a) Picture of a Mico Robot.

(b) Picture of a Baxter robot.

Figure 6.1: Commercial robots used in the project.

6.1 Mico

Mico is a 6/4 DOF robotic arm produced by the Canadian company Kinova. It comes with a default gripper which can have two or three fingers. There are several available versions with minor differences between them, and in this project the robot used is the *m1n6s200*: 6 DOF and two fingers that can be controlled individually (see Figure 6.1). The Mico robot is lightweight (4.6 kg) and has a reduced working area of 700 mm reach with a maximum payload of 2.1 kg. It is covered by a reinforced plastic frame and its average power consumption is of 25W. Software packages for operating it using ROS are available online.

6.2 Baxter

Baxter is a collaborative robot with 14 DOF (7 per arm), wheels and a 1024 x 600 SVGA LCD screen. It is considerably heavier than the Mico (almost 140 kg) and has a larger working area with a reach of 1210 mm per arm, without taking into account that the robot can be manually moved around thanks to its pedestal with wheels. Its maximum payload per arm including the end effector is 2.2 kg. The end effector that comes by default with the Baxter is a hand with two parallel fingers, but unlike Mico both fingers are dependent on one another as the controlled variable is the distance between them. Baxter is designed to work under the ROS framework so it has been easily integrated in the system. See Figure 6.1 for a complete view of the robot.

6.3 CAPDI

CAPDI has been designed and built entirely in the UPC as part of the Inhands project and is suited to work in this specific environment. It is a 3 DOF cartesian robot mounted on the ceiling, with a 3 finger adaptive Robotiq gripper as its end effector. The gripper has two possible movements: open/close the fingers to grip, and widen/narrow the gap between the two fingers located at the same side of the hand. The main advantage of CAPDI is its working area, as it can reach the whole table surface at different height levels. However, it only has 3 DOF so it is not able to rotate its gripper in any direction. CAPDI has also been designed to work with ROS, and due to the fact that Robotiq is integrated in the ROS Industrial project, it can also be controlled using ROS.

7. Other Hardware

Apart from the robots the system has three RGBD Kinect One cameras for computer vision and one PC Server connected to a plasma TV. All the robots are connected to the server: Mico via USB, Baxter via Ethernet and CAPDI via a specific control unit. The cameras are also connected to the server with 3 USB cables.

7.1 Kinect One

Three Kinect One cameras are attached to the CAPDI frame, looking downwards. As one of the restrictions for the computer vision subsystem is to cover the whole area of interest, they are placed in a zenithal position from where they can see the whole table and shelves. These cameras have three main elements: a IR projector, a RGB camera and an IR camera. The projector emits a matrix of IR rays that are captured by the IR camera, which computes the depth of each pixel using the Time of Flight principle. Then, an algorithm registers the depth image and the RGB image to obtain an RGBD image (2.5 D). Finally, in order to obtain the real world X,Y,Z coordinates of each pixel, the camera pinhole model is applied.

7.2 Server PC

The core of the whole robotic system is the Server PC, located behind the kitchen shelves. It receives information from the cameras and the robots, performs all the computations and algorithms required to process the information and then sends instructions to each robot in order to perform the desired tasks. This server runs under Ubuntu 16.04.

8. Software

In order to connect all the hardware elements together and share and process their information, a strong and robust software basis is needed. As it has been pointed out in the previous points, the main software working framework of the project is ROS, given its modularity and distributed capabilities. RViz is used for simulation and visualization, and to control the robots, a robot planning package designed for ROS is used: MoveIt!. As for the computer vision, libfreenect2 and IAI-Kinect are used for communicating with the cameras and for preprocessing the images. These preprocessing algorithms make use of the OpenCL and OpenCV libraries, and OpenCV is also used for machine learning as it has powerful and varied standard algorithms. PCL is used to process the point cloud data obtained from the Kinect One cameras. Finally, as there are many independent hardware systems that require different software and operating system configurations, this project uses Docker containers to provide the needed packages for each part of the system.

The Object Recognition algorithm is written in C++ and compiled using Catkin, whereas the Action Recognition is developed in Python 3.7, using NetworkX as a graph library. The packages are documented using Doxygen.

(a) Programming framework. [17]

(b) Motion Planning. [11]

(c) Visualization. [18]

Figure 8.1: ROS related software used in the project.

8.1 ROS

As stated in its website:

> ROS is a *flexible framework* for writing robot software [...] a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a *wide variety of robotic platforms* [..] ROS was built from the ground up to encourage *collaborative robotics software development*.

These three highlighted design specifications make ROS the perfect choice for developing software for this system setup. The variety of robots and hardware requires a framework that supports all of them, while being flexible enough to enable parallel development for each module. Moreover, as there is more than one subject working on the project, there is also a need for collaborative and flexible development, which is also one core feature of ROS.

ROS is based on a simple structure made of *nodes* that communicate with each other through a *master node*. Each one of these nodes is run in parallel on its own process, enabling concurrent programming with a minimal implementation cost. Nodes can communicate with themselves through *topics*, *services* or *actions*, passing data as messages. One drawback of this architecture is that the data of each message is deep-copied before being passed as a message, which can result in slow communication rates if the data to send is large. However, ROS provides a workaround for this issue: *nodelets*, node-like modules which run under the same process and have a different callback structure than normal nodes. Nodelets are loaded to a *nodelet manager* in a widget-fashion way.

In the current project, the whole computer vision pipeline is implemented in nodelets, as the RGBD data is too heavy to meet the real-time specification if working with nodes. This enables the aforementioned intraprocess communication, thus exchanging the deep-copied messages for pointer messages. The server runs ROS kinetic, which controls the cameras and the CAPDI and Mico robots. Baxter, however, runs its own ROS master server using the ROS indigo version.

8.2 RViz

RViz is a lightweight but powerful visualization software package for ROS. It is the standard package the comes with the ROS software, and has a wide range of functionalities: 3D real-time simulation and visualization, logging, handling parameter servers, visualization of node structures and connections, supervision of message rates, etc.

RViz is useful in this project as it enables a visual representation of the RGBD data as well as the object detection algorithm results through all its steps.

8.3 MoveIt!

MoveIt! provides a range of software solutions for robot planning, collision detection and multi-robot systems for almost any robot that can be controlled using ROS. It is used to plan robot movements and simulate them before the actions are performed, to check their viability and avoid collisions between robots or with the environment.

After simulating the movements, it also handles the dispatching of planned movements as instructions for each robot.

(a) GPU programming. [12]

(b) CV & ML. [13]

(c) Kinect ROS wrappers. [10]

(d) Kinect listeners. [14]

Figure 8.2: Computer Vision & Machine Learning related software used in the project.

8.4 OpenCL & OpenCV

The current project relies heavily on complex vision algorithms and machine learning procedures. Coding each single software piece from zero would be a task out of the scope of the project, so the open source libraries OpenCL and OpenCV are used as they provide a trusted and efficient basis for developing new algorithms.

OpenCL is used by IAI-Kinect to compute the registration of RGB to Depth images given a set of precomputed camera parameters, while OpenCV is used for computing 3D and 2D image descriptors for training and then recognising objects using machine learning.

8.5 IAI-Kinect & libfreenect2

Connecting and getting data from the Kinect One cameras is not a trivial task. IAI-Kinect provides a ROS frame to launching the cameras, take pictures, preprocess the data and finally retrieve the result as a ROS message. The low-level camera communication is handled using another open source library: libfreenect2.

IAI-Kinect2 also has the option to work as ROS nodelets, lowering the message load in the ROS network. The calibration parameters for each camera are obtained by a calibration procedure also handled by the IAI-Kinect package.

Figure 8.3: Point cloud data processing software used in the project. [16]

8.6 Point cloud library

The data obtained from the Kinect One cameras is often noisy and covers unnecessary areas. Moreover, in order to extract defined regions of the point clouds to detect objects, a pipeline of data processing is needed. PCL is an open source point cloud processing library

that can be easily integrated in ROS, and which also provides many useful processing algorithms such as ICP, RANSAC, filtering methods, segmentation methods and much more.

In the current project, it is one of the core components of the object recognition pipeline because it takes care of receiving the raw point cloud Kinect data and extracting the point clusters that are then passed to the recognition step. It is designed to be used in C++ code, which suits the development frame of this project.

Figure 8.4: Operating system virtualization software used in the project. [8]

8.7 Docker

As flexible and modular as ROS is, it cannot cope with situations where some hardware elements require different library versions for the same software packages. It may be the case that one robot can only work with ROS Kinetic whereas a newer model is designed to work with Melodic. As this is the case of the current project, another layer of flexibility is needed to connect al the system elements together.

Docker provides the solution to this problem: it encapsulates all the software requirements of a particular system into a Docker *image*, including the OS specifications. This image can be then used to create a *container* which runs as a separate operating system with the encapsulated packages. In programming terms, an image can be regarded as a class and a container as an instance of the class.

This architecture enables the launching of different systems with varied package versions (or even different OS versions) in the same machine, while retaining the communication capabilities between them . This is the reason behind the use of Docker in this project.

Part IV

Object Recognition

In order to perform a successful action recognition algorithm, the system requires input from the real world about how the agent is interacting with its environment. In the case of the current work, this interaction involves using and moving daily kitchen objects and food such as cups or a milk brick. Therefore the system must be able to recognize and identify this agent-object interaction in order to deduce the intention of the actions, which is done using computer vision.

The process begins by getting the input data from the RGBD cameras. This point cloud data is then passed onto the segmentation processing block: the data is subsampled and cut, then the planes and surfaces are extracted and finally the object clusters are extracted for recognition. The object recognition process ends with a descriptor computation and its posterior classification using a trained SVM model. This whole algorithm is coded in C++ and each single step works as a ROS nodelet. All the nodelets run under the same manager and so the processing and data transfer between steps is faster.

The libraries and packages used in the object recognition algorithm are libfreenect2, IAI-Kinect, PCL, OpenCV and OpenCL.

9. Camera Input

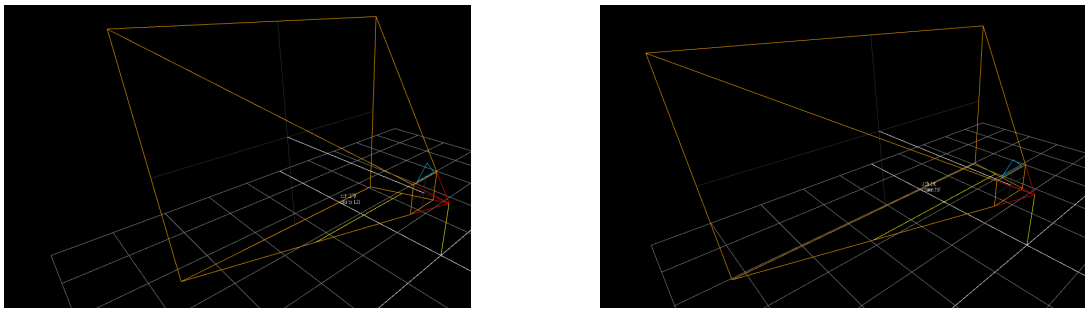
In the current project our vision system consists of three Kinect One RGBD cameras that provide 512x424 RGBD images at a top frame rate of 15 Hz. Due to the fact that our goal is to cover the whole kitchen table to get vision input, the cameras are placed in a zenithal position so as to see the table from above but still manage to capture the objects present in the shelves.

9.1 Camera Setup

The placement of the cameras relative to the world scene has a great influence on the performance of the recognition algorithms, as it determines the point cloud density of the volume of interest and the visual angle of the scene, which can result in better or worse scanning of planar surfaces. The following descriptions explain the placement used in the project and the calibration procedure used to tune the intrinsic and extrinsic camera parameters.

9.1.1 Placement

The FOV of the Kinect cameras is 71 degrees wide and 60 degrees high for the Depth camera and 84 degrees wide and 54 degrees high for the RGBD camera (see Figure 9.1, with a depth reach between 0.5 and 4.5 meters. Despite having a long reach, it is advised to place the objects of interest at approximately 1 meter from the camera to obtain the highest quality data (a resolution between 1.5-2 mm per pixel). In this project, however, this is not possible.



(a) Depth camera FOV. (b) RGB camera FOV.

Figure 9.1: Kinect One cameras' FOV. [24]

The cameras are attached to the CAPDI frame hanging from the ceiling, looking upside down. To achieve the minimum invasive impact of the robotic hardware in the subject space, the camera placement was fixed at this position. One drawback of the camera placement is the fact that neither the table nor the shelves surface are perpendicular to the camera's focal axis, resulting in worse quality point clouds. Moreover, the average distance between the camera lenses and the objects of interest is about 2 meters.

Figures 9.2 shows the average point cloud provided by the cameras from its position. It can be seen in Figures 9.3 and 9.4 how the image borders are heavily distorted, and the

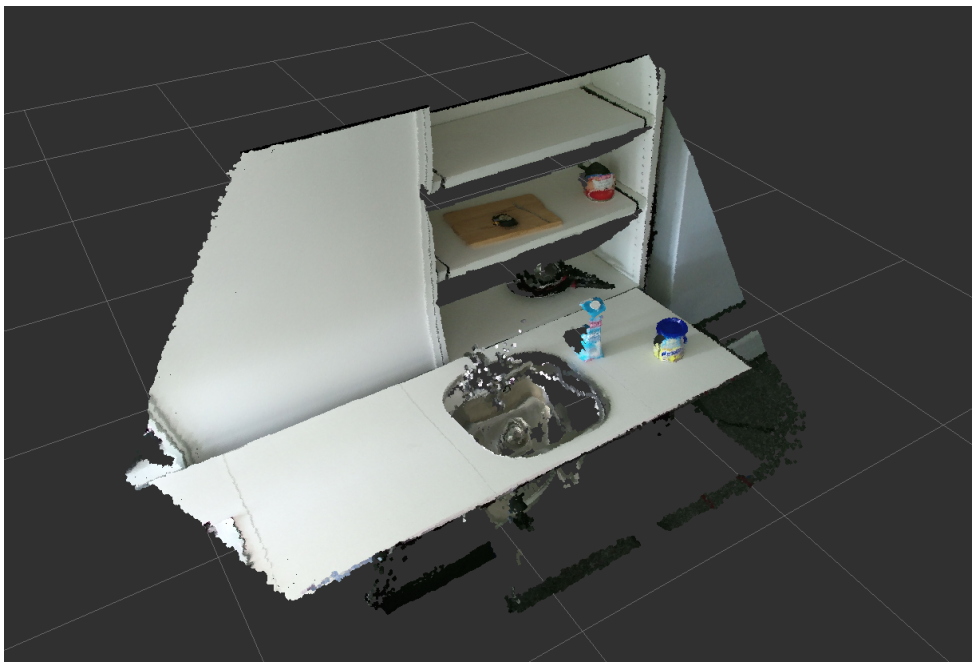


Figure 9.2: Raw point cloud received from one camera after correcting the distortion.

high levels of noise throughout the whole image can also be seen. The algorithm developed must be able to overcome this imprecision to provide a robust object segmentation and recognition.

9.1.2 Calibration

Before capturing any data, the cameras must be calibrated to obtain their extrinsic and intrinsic parameters. The extrinsic parameters provide the geometrical relationship between the projector, the IR camera and the RGB camera in order to enable the registration of both RGB and Depth images into a single RGBD image. The more precise these parameters are, the better the correspondence between Depth and RGB pixels of the two images.

On the other hand, intrinsic parameters are those that define the physical characterization of the camera lenses. They allow to counteract and correct some of the negative effects introduced to the depth measurements by the distortion of the lenses. First the depth measurement is corrected with the intrinsic distortion parameters k_1, k_2, k_3 (Taylor coefficients of the distortion factor) and p_1, p_2 (distortion centre). Then a back projection to the 3D space is performed using the the camera focal parameters f_x, f_y (focal lengths - ideally equal, they represent the "pixel distance" to the camera plane) and c_x, c_y (camera

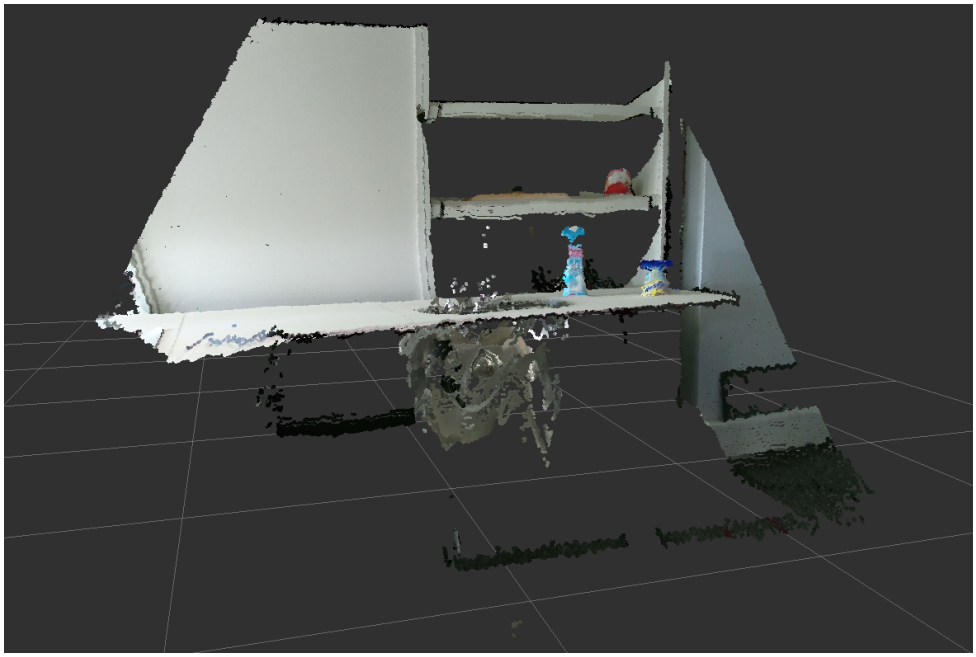


Figure 9.3: The same point cloud from Figure 9.2 from another viewpoint.

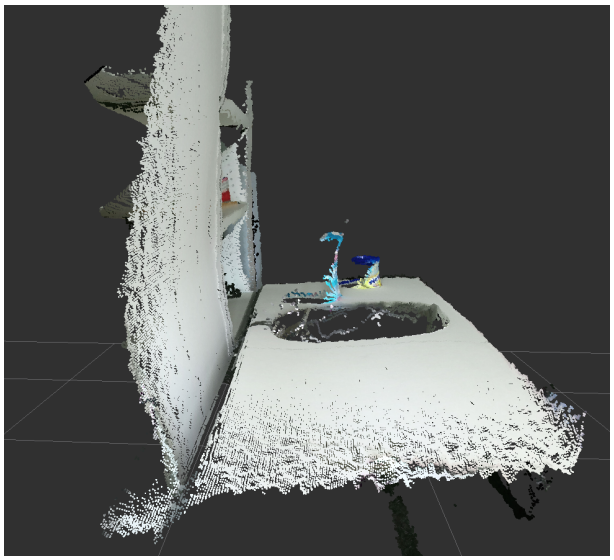


Figure 9.4: The same point cloud from Figure 9.2 from yet another viewpoint.

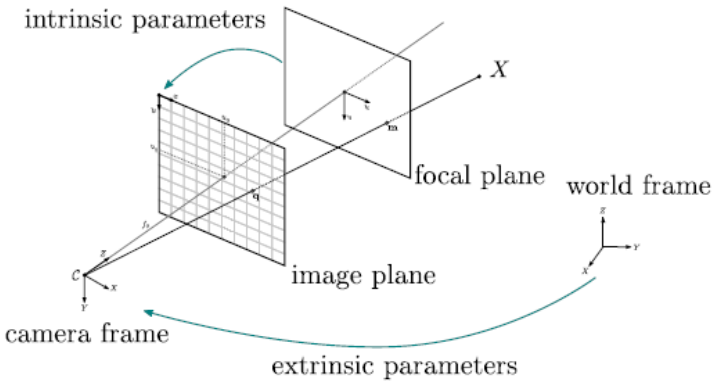


Figure 9.5: Ideal Pinhole Camera model. [15]

principal point - ideally the center pixel of the image) by applying the pinhole camera model [23] [5] (see Figure 9.5).

Given a pixel in image coordinates $p = (x, y)$, a depth measurement Z and the intrinsic parameters f_x, f_y, c_x, c_y, the back projection to the 3D space (X, Y, Z) of the corresponding pixel is computed with the equation

$$
\begin{aligned}
Z \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} &= \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \\
&\downarrow \\
X &= \frac{Z}{f_x} \cdot (x - c_x) \\
Y &= \frac{Z}{f_y} \cdot (y - c_y) \\
Z &= Z
\end{aligned}
\tag{1}
$$

10. Algorithm Overview

The main requirements for the object recognition algorithm is that it has to perform robustly and in real time. Given the characteristics of the input point clouds the developed

processing should be able to overcome any significant noise and geometrical incorrectness of the measurements. Furthermore, as it will be explained in the Action Recognition Chapter, our project considers tasks whose duration ranges from 30 to 180 seconds. The real time requirement has been thus set to a maximum of 1 second per input image.

Having stated the requirements of the algorithm, in the next sections all of its steps will be explained. It has five steps:

1. Obtain camera input - performed by the *kinect_bridge* nodelet, which communicates with its assigned Kinect camera. It retrieves the raw data, applies the correction parameters, computes the RGBD image and the 3D point cloud and passes them onto the next step.

2. Cloud Preprocessing and Registration - In order to speed up the camera to camera cloud registration and the cloud segmentation steps, it has been decided to compute a subsampled cloud version to use in the aforementioned steps. This task is carried out by the *calibration* nodelet, which subsamples the cloud, cuts out the non relevant points to keep only the areas of interest, and then applies the necessary transformations to register the point cloud to the ones of the other two cameras.

3. Plane Segmentation - As the final goal of the object recognition algorithm is to classify single objects, a segmentation has to be performed to isolate the object point clusters from the rest of the cloud. The first step of this isolation is the detection of large planes in the images. These mathematical planes will serve as the basis for segmenting and deleting large surfaces and finally obtaining a point cloud made only of object points. This task is performed by the *plane_segmentation* nodelet.

4. Surface Segmentation - From the detected planes (mathematical) the next step is to segment the point cloud surfaces that correspond to given planes (actual points). One plane can have several surfaces in the cloud (i.e. several tables of the same height close to each other) and the task of the *surface_segmentation* nodelet is to cluster and segment them and remove the corresponding surface points.

5. Object Segmentation - After removing the unnecessary surface points, the points left in the cloud belong to objects. The last step before the recognition is to group these points in clusters that represent objects. The *object_segmentation* nodelet carries out this task and passes to the final step an array of point clusters.

6. Object Recognition - The *object_recognition* nodelet performs one of the most critical tasks of the algorithm, as it is responsible of computing the descriptors of each object and using a trained model to recognize them. In this project the recognition is a two step process, the first one using volume descriptors and the second one using colour descriptors.

7. Dispatcher - The one and only task of the final node, the *dispatcher*, is to translate the information from the object recognition phase into a coded language that can be understood by all the other elements in the system, primarily the action recognition algorithm. This node gathers the information from the three pipelines (one for each camera), and as the information at this stage is no longer a points cloud but a list of recognized objects (name, dimensions and position), this step can be implemented as a node instead of as a nodelet. This approach is also useful, as the three pipelines are implemented in independent nodelet managers, and it is convenient to have an outsider node to gather the output from these managers.

A graphical representation of the pipeline is shown in Figure 10.1.

10.1 Nodelet Structure

One of the goals of the project is to document the algorithms developed to ease their use in further work. This effort relies not only on writing adequate comments and guides but also on writing well structured code. Thus all the nodelets have been written using the same code structure, with minor differences between them. In this way, by understanding the common workflow one can easily read through the process of each individual nodelet.

There are several ways to implement nodelets in ROS. One of them, which has been used in this project, is to code the nodelets as if they were nodes and then wrap them using a nodelet class so that they can be loaded into a nodelet manager. This manager handles the message queue for all its child nodelets in the following way: at each program tick (called a ROS spin), it gathers the messages received for all nodelets and then processes each one of them single-handed or using a multithread structure. This has one drawback: if a nodelet requires two or more messages to be synchronized in order to process them, they might arrive at different times and be processed at different times. One way to solve this issue is to use message filters so that no message is processed until all input topics have received messages with similar timestamps.

However, the implementation used in this project has been somewhat different. Instead of using message filters, which was tried but delivered poorly results, a flag system has been created. Whenever a message arrives, it raises a flag to indicate that it has been received. Then, each nodelet launches a separate thread which checks at each spin if all the flags have been raised. If so, it calls the nodelet processing method and resets the flags. Hence the nodes written for nodelet implementation have the workflow seen in Figure 10.2.

In this way, the structure and code of all nodes and their nodelets implementation is the

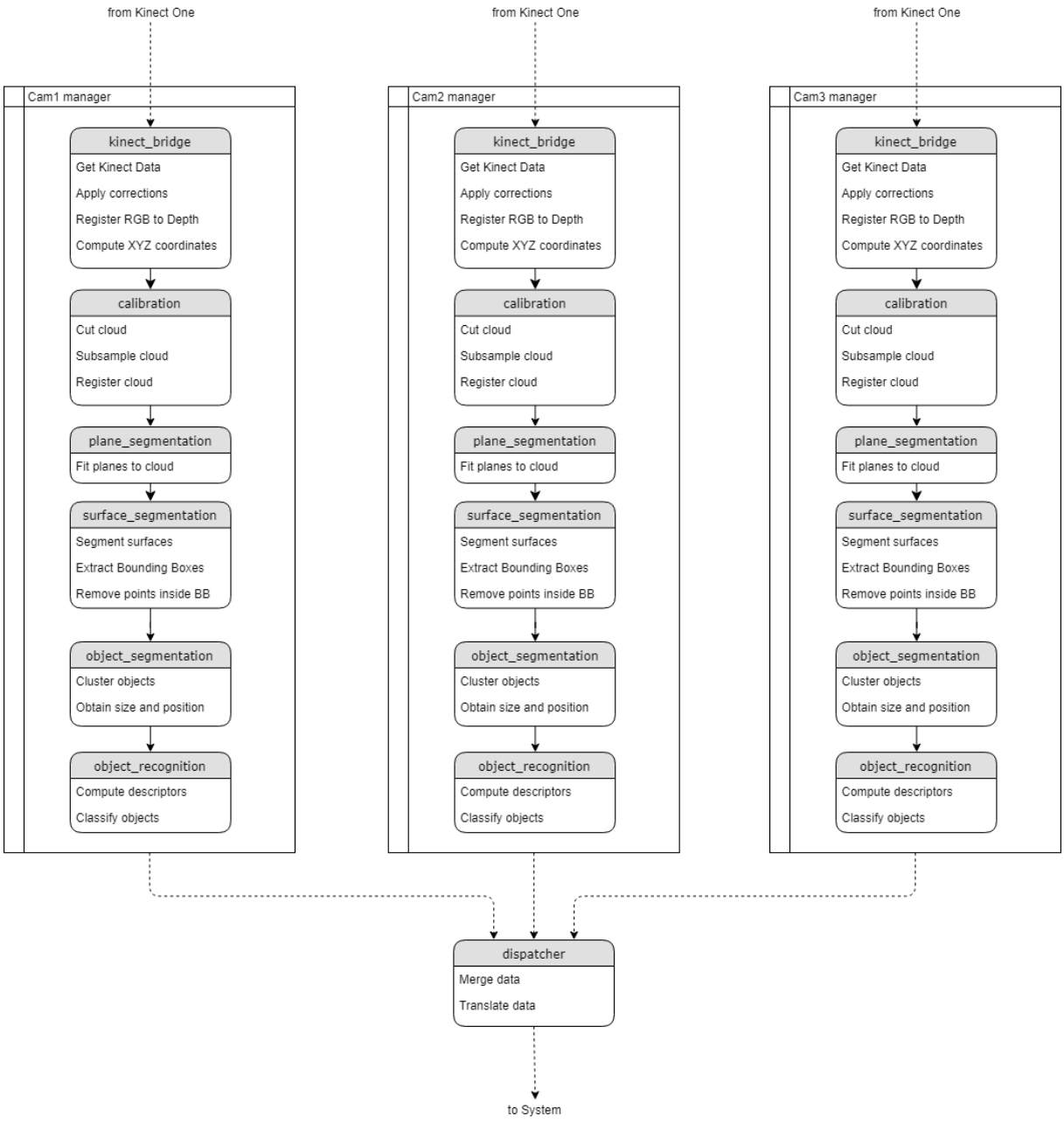


Figure 10.1: Scheme of the object recognition pipeline.

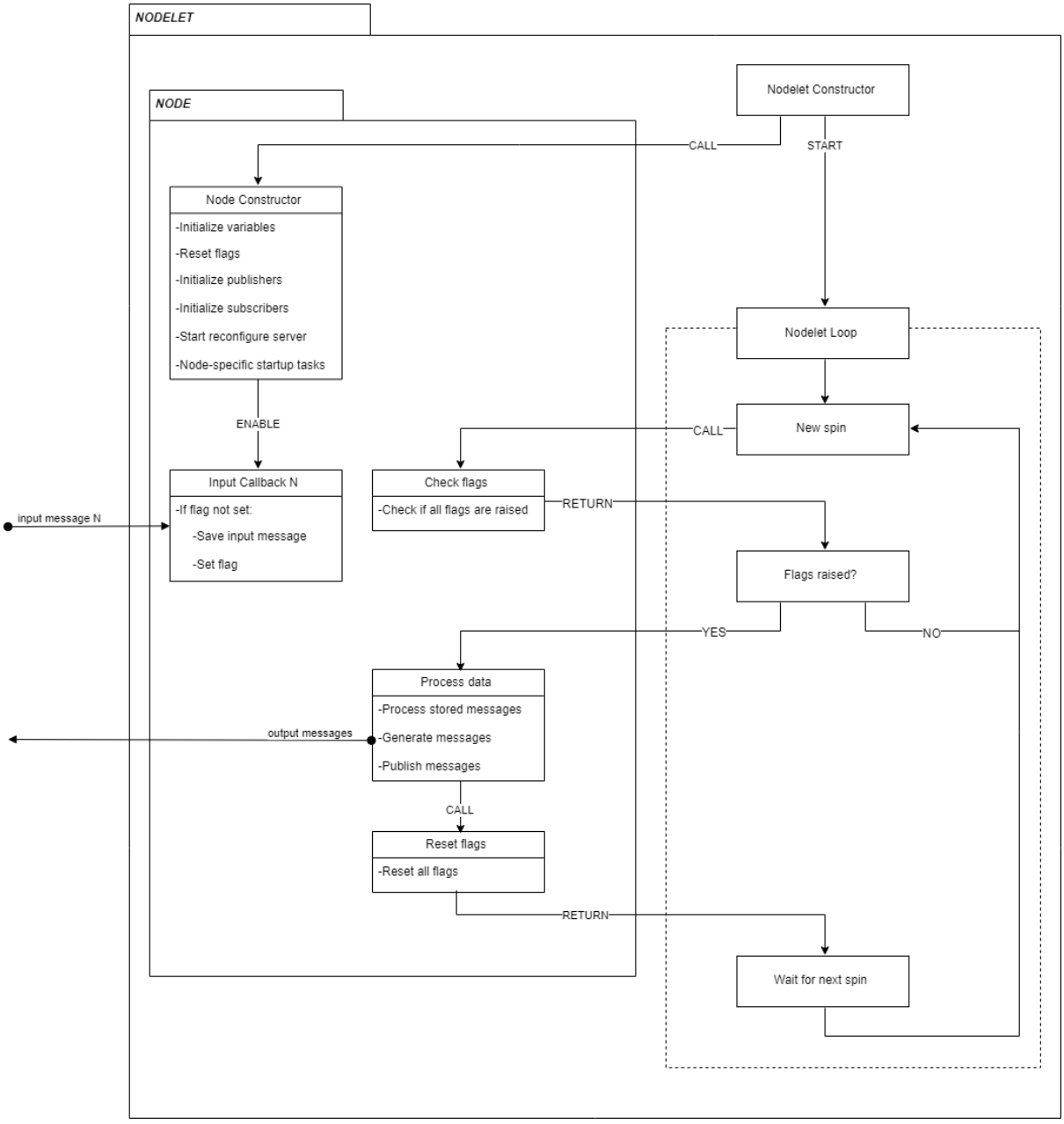


Figure 10.2: Workflow of the node implementation as a nodelet.

same. The only difference between the nodes are the number of publishers/subscribers they have, the message types, and most importantly the objects they use to process the data. Each node represents a step in the object recognition process and thus uses a unique object to process the data. A more in depth description of all the nodes follows.

10.2 Calibration nodelet

The calibration nodelet has the characteristics (the topics are represented as *topic name[message type]*) shown in Table 10.1.

Input topics (Subs)	cam_subscriber_[point cloud]
Output topics (Pubs)	cut_cloud_publisher[point cloud], cut_subsampled_cloud_publisher[point cloud]
Processing objects	Registrator, Preprocessor

Table 10.1: Calibration node characteristics.

The node subscribes to the output point cloud from the IAI-Kinect node (the one that fetches data from the camera and performs the RGBD registration, 3D computation and distortion corrections). After its own processing, it publishes two point clouds: a cut version of the one received, and a subsampled version of the cut cloud. Moreover, at startup the node loads the ground truth cloud from a predefined path to the Registrator object so that all the input clouds can be registered to the ground truth during the program execution. When a new camera point cloud is received, the processing method takes the following actions:

1. **Register** the input cloud to the ground truth cloud. This step uses ICP for registration for the first three clouds of each session, and then stores the final transformation for the following inputs. This is done to avoid unnecessary time spent on ICP computations as the camera is still, and the object that the node uses for this purpose is the Registrator.
2. **Cut** the cloud using X-Y-Z limits to retain only the points that are inside a predefined box that represents the volume of interest. Uses the Preprocessor object.
3. **Subsample** the cut cloud in order to use the subsampled version in some of the algorithms to speed up the computations, using also the Preprocessor object.

As it has been said, the node then sends out both the cut and the cut&subsampled cloud to the next node. The objects used by this node have the structure shown in Figure 10.3

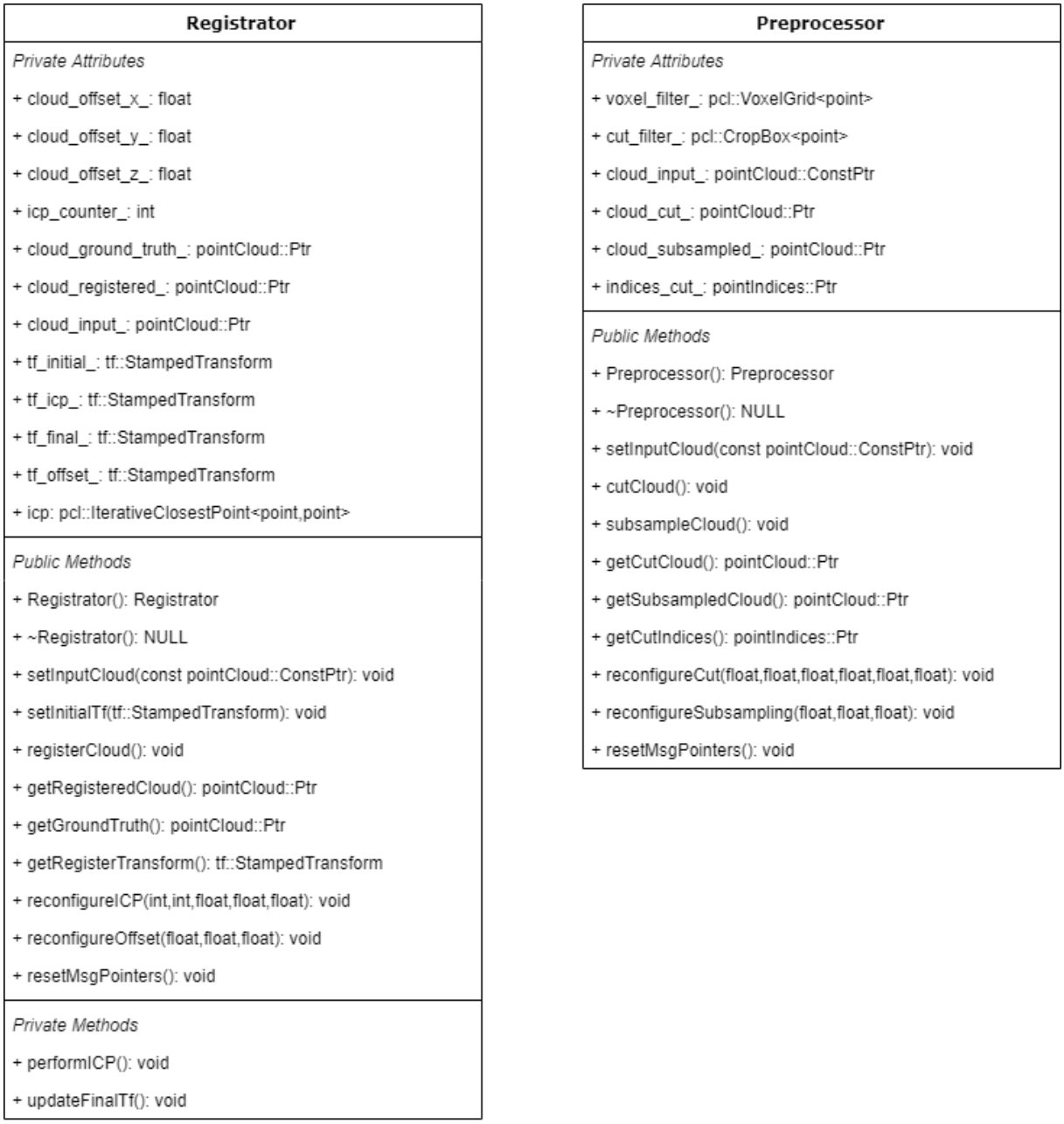


Figure 10.3: Registrator and Preprocessor class structures.

The ICP used is a standard point-to-point registration with two thresholds as ending criteria: a transformation and a fitness epsilon. For cutting the cloud, a CropBox filter is used, and the subsampling is performed using a VoxelGrid filter with voxels of a size that can be set by the user. The parameters that can be changed using the reconfigure server of this node are the following:

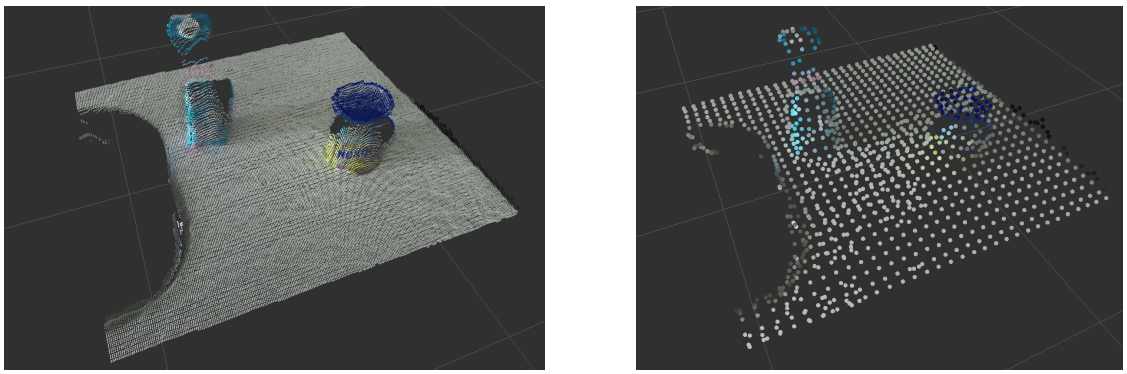
ICP	reciprocal_corr	use reciprocal correspondences
	max_iterations	maximum allowed ICP iterations
	max_corr_distance	maximum distance for finding point pairs
	tf_epsilon	transformation epsilon stopping criterion
	fitness_epsilon	fitness epsilon ending criterion
Cutting	min_x	minimum x coordinate
	min_y	minimum y coordinate
	min_z	minimum z coordinate
	max_x	maximum x coordinate
	max_y	maximum y coordinate
	max_z	maximum z coordinate
Subsampling	voxel_dimx	voxel size in x axis
	voxel_dimy	voxel size in y axis
	voxel_dimz	voxel size in z axis

Table 10.2: Calibration node paramenters.

This parameter structure allows to tune the parameters while the algorithm is running, and is used in all the nodes of the algorithm. In this way the algorithm is both flexible and adaptable to different scenarios and can be easily reused. An example result of the calibration output is seen in Figure 10.4.

10.3 Plane Segmentation Nodelet

The second step of the Object Recognition algorithm is the Plane Segmentation. The reasoning behind performing a plane segmentation is that by analyzing the kitchen setup, an early conclusion was drawn: all the objects inside the area of interest lie on a surface. This surface could be either the table or one of the drawers. Furthermore, as far as the scope of the action recognition is concerned, the actions will only occur on top of the table. The user sitting in the wheelchair cannot reach the shelves, so picking and placing objects in them is a task corresponding to the robots.



(a) Cut point cloud showing the ROI points. (b) Subsampled version of the cut cloud.

Figure 10.4: Sample output of the Calibration node for camera 1.

This leads to the conclusion that the final area of interest, where the user will perform the task, is the rectangular volume defined by the surface of the table spanning half a meter up. This is where the non-system controlled actions will occur, which are the ones analyzed by the object recognition and action recognition algorithms. The rest of the movements will be performed by the robotic system, and thus will be known to the algorithms. This interest volume is roughly sketched in Figure 10.5.

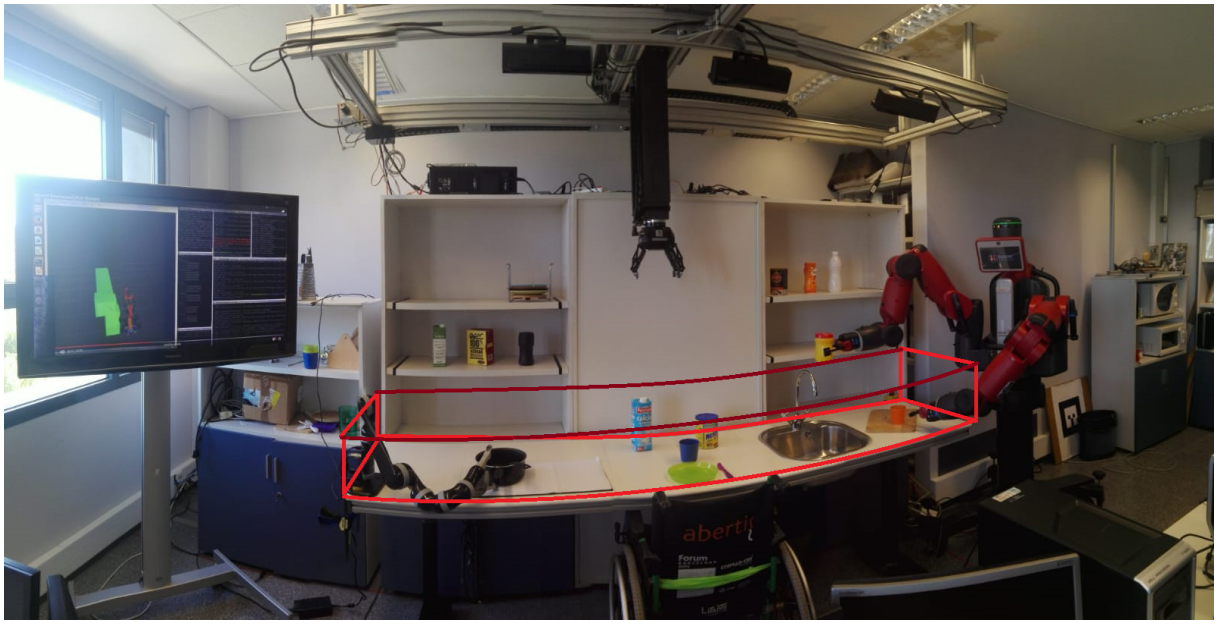


Figure 10.5: Sketch of the volume of interest for the algorithms.

As for the node inputs and outputs, they can be seen in Table 10.3. It subscribes to the output topics of the Calibration Node (cut and subsampled cloud), and outputs four different topics: the cut and subsampled cloud are passed onto the next node, a point

cloud containing the plane points is outputted for visualization, and an array of Plane objects containing the segmented planes' data is outputted too.

Input topics (Subs)	full_cloud_subscriber[point cloud], subsampled_cloud_subscriber[point cloud]
Output topics (Pubs)	planes_data_publisher[plane array], planes_pc_publisher[point cloud], full_cloud_publisher[point cloud], subsampled_cloud_publisher[point cloud]
Processing object	PlaneSegmentation

Table 10.3: Plane Segmentation node characteristics.

The object used by the node for segmenting the planes is the PlaneSegmentation object. This uses in turn a further Plane class to gather the required information and functionality for a plane. The structure of both objects is presented in Figure 10.6. It can be seen how the method used for segmentation is the Sample Consensus, using a Plane model. There are also two remarks to be made about this step: the maximum number of planes can be fixed, as well as the minimum number of points that a segmented plane must have to be accepted.

Following the reasons explained above, in the current implementation of the algorithm only one plane is detected: the table. In spite of this specific adjustment for the current setup, the algorithm has been designed and tested for working with multiple planes. The workflow of the Plane Segmentation node is quite simple

1. **Get** the input clouds from the previous node.

2. **Segment** the desired planes using the PlaneSegmentation object.

3. **Send** out the segmented planes' info and pass on the received input clouds.

Finally, the parameters that can be changed of this node are detailed in Table 10.4. Also an example of the output from a Plane Segmentation node can be seen in Figure 10.7.

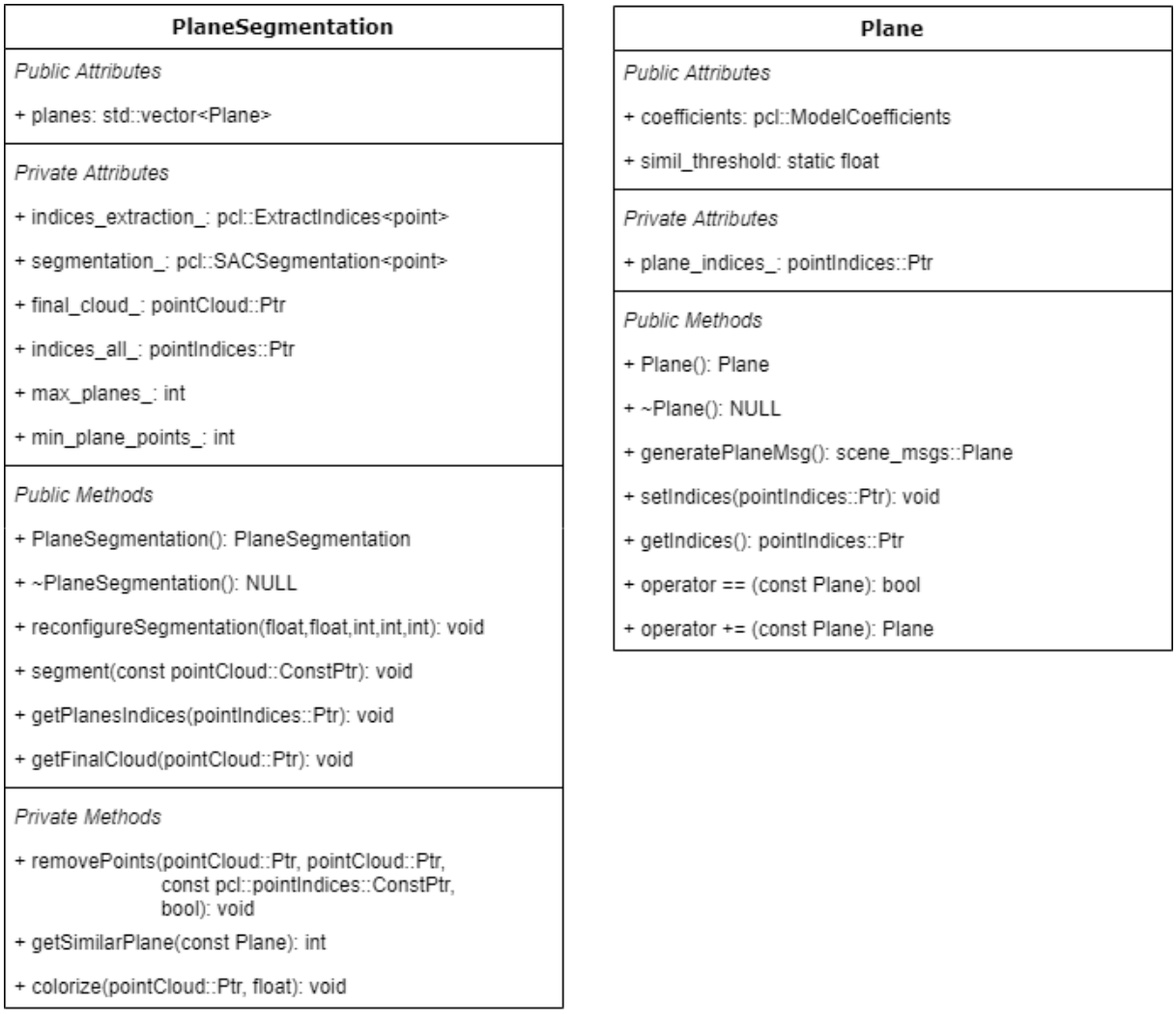
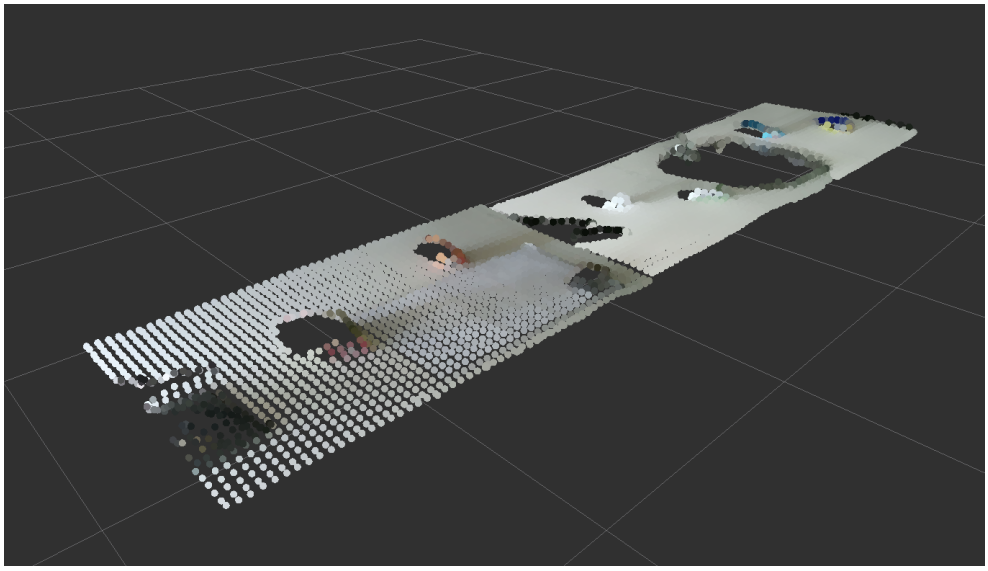


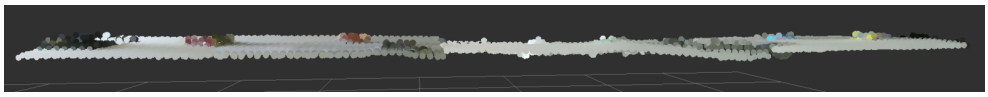
Figure 10.6: PlaneSegmentation and Plane class structures.

General	max_planes	maximum planes to detect
	min_plane_points	minimum points a plane must have
SAC Segmentation	distance_tolerance	distance threshold for SAC
	orientation_tolerance	orientation threshold for SAC
	max_iterations	maximum number of iterations

Table 10.4: Plane Segmentation node parameters.



(a) Point cloud of the points which belong to the detected plane.



(b) Frontal view of the plane points.

Figure 10.7: Sample output of the Plane Segmentation node for all cameras.

10.4 Surface Segmentation Nodelet

Next in the processing pipeline comes the Surface Segmentation. This node has the task of extracting bounding boxes of all the surfaces which correspond to the planes detected in the previous step. The bounding boxes should contain only plane points and leave aside the rest which are supposed to belong to objects. The input and output topics of this node are the following, shown in Table 10.5. It receives the full cut cloud and its subsampled version from the Plane Segmentation node, as well as the segmented planes' data, and outputs three topics: the surfaces' data, a marker array to visualize the surfaces' bounding boxes, and the full cut cloud is passed on. As the following steps will be the Object Segmentation and Recognition, only the full cut cloud is passed on. The idea is to use as many points as possible for the object recognition, but in the Plane and Surface segmentation only the subsampled cloud is used, as there is a lower point density requirement for this steps to be successful.

Input topics (Subs)	full_cloud_subscriber[point cloud], subsampled_cloud_subscriber[point cloud], planes_subscriber[plane array]
Output topics (Pubs)	surface_publisher[surface array], marker_publisher[markers], full_cloud_publisher[point cloud]
Processing object	SurfaceSegmentation

Table 10.5: Surface Segmentation node characteristics.

In order to segment the surfaces, the node uses the SurfaceSegmentation object and the surface data and functionality is encapsulated in the Surface Object. The structure of this two objects can be seen in Figure 10.8. As for the processing workflow of this node, it is the following

1. **Get** the input cloud and the plane array from the previous node.
2. **For each** plane in the plane array:
   (a) **Get** the indices of the plane points
   (b) **Get** the coefficients of the plane
   (c) **Extract** the surfaces using the plane points and coefficients and an Euclidean Cluster Extraction with Kd-Tree search.

The resulting surfaces are stored in an array as Surface objects. The Surfaces store information about their bounding box as well as functionality to generate markers and

messages for the topics. Both the structure of the SurfaceSegmentation and Surface objects can be seen in Figure 10.8.

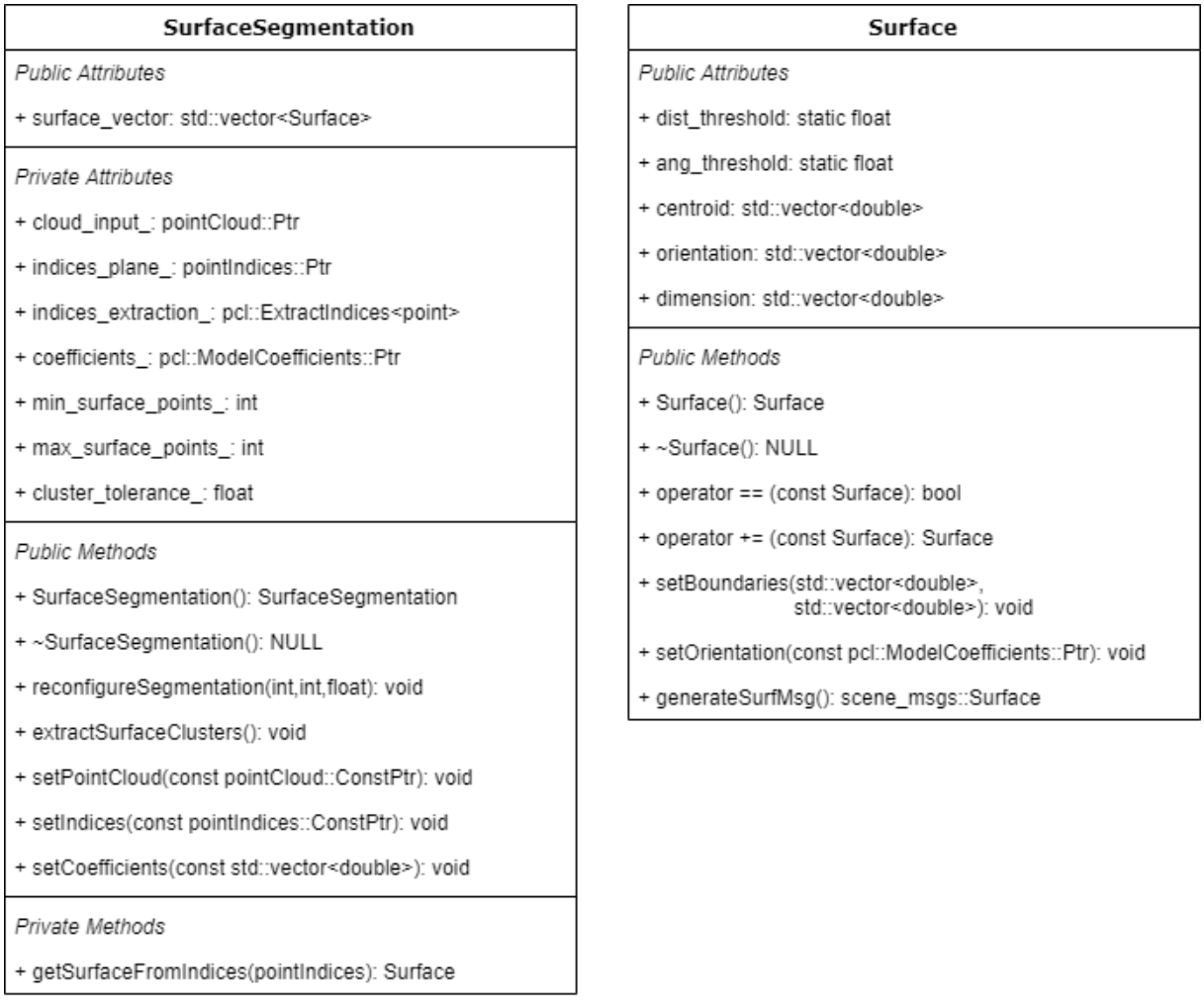


Figure 10.8: SurfaceSegmentation and Surface class structures.

Finally, the parameters of the segmentation can also be tuned using the reconfigure server. Only three parameters can be changed for this node: the minimum and maximum number of points a surface cluster must and can have, and the distance tolerance used by the Euclidean Clustering algorithm to group points together. The name of the parameters alongside their explanation is found at Table 10.6, and an example of the bounding boxes detected by the node for all three cameras is shown in Figure 10.9.

General	max_surf_points min_surf_points	maximum points a surface can have minimum points a surface must have
SAC Segmentation	distance_tol	distance tolerance for clustering

Table 10.6: Surface Segmentation node parameters.

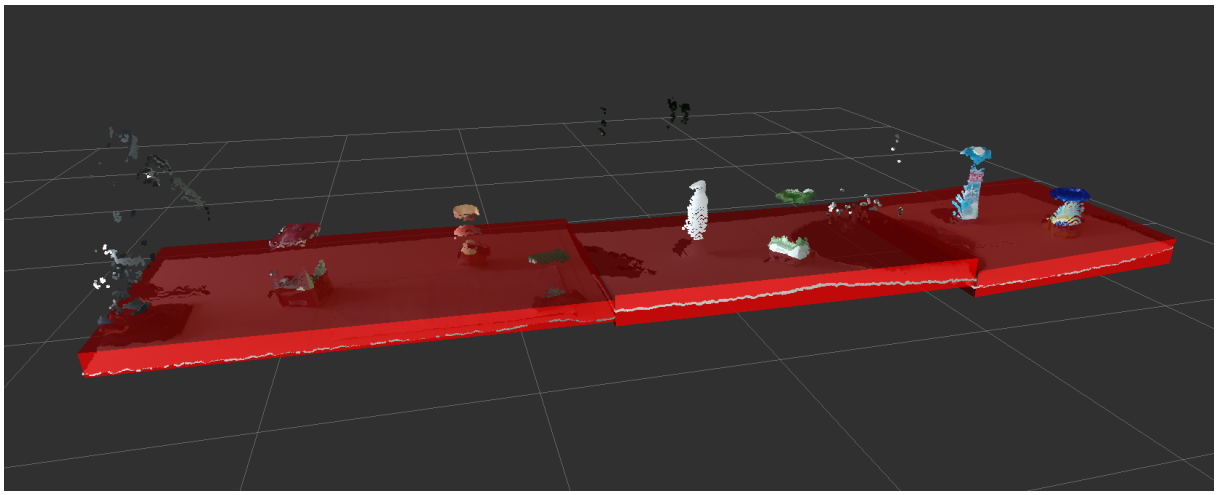


Figure 10.9: Bounding boxes of the surfaces detected by the Surface Segmentation node of the three cameras.

10.5 Object Segmentation Nodelet

After segmenting surfaces, the last step before object recognition is clustering the remaining points into object clusters, which is done by the Object Segmentation Nodelet. The node has two input topics: one to get the surfaces' data, and another to get the full cut cloud (without subsampling) from the previous node. At this step the algorithm must compute object clusters that contain a high density of points to increase the recognition rate. The less density of points a cluster has, the less specific its descriptors will be; making classification considerably harder[1]. That is the reason behind using the full cloud at this point of the algorithm.

The steps taken by the Object Segmentation node are

1. **Get** the full cloud and array of surfaces from the previous node and pass them to the ObjectSegmentation object.

[1] This is not true for all recognition algorithms. Sometimes it is advisable to reduce the input data in order to obtain recognisable descriptors. In this project, however, the point density of the subsampled cloud is too low so the use of the full cloud is recommended.

2. Call the **segmentObjects** method from the ObjectSegmentation object, which does the following
   (a) **Remove** the points from the cloud that are inside all the Surface bounding boxes using a CropBox filter
   (b) From the remaining points, **remove** outliers using a Radius Outlier Removal filter
   (c) **Cluster** the final set of points into single object point clouds.
   (d) **Generate** an array of objects for recognition and a point cloud of all the object points for visualization

The array of objects contains a list of object data. For each object, the data stored is again its bounding box: the centroid, its dimensions and the orientation. Moreover, the Object class also offers utility methods for generating markers and ROS messages. The main output topics of the node are the object data array and the full cut cloud, with the possibility to output also the cloud of all the object points for visualization. The main characteristics of the node are shown in Table 10.7.

Input topics (Subs)	full_cloud_subscriber[point cloud], surfaces_subscriber[surface array]
Output topics (Pubs)	object_publisher[object array], full_cloud_publisher[point cloud]
Processing object	ObjectSegmentation

Table 10.7: Object Segmentation node characteristics.

For a better insight on the Object Segmentation methods and the Object class for holding the data, their structure is shown in Figure 10.10. Moreover, additional parameters can be tuned to adjust the segmentation. These parameters are detailed in Table 10.8. Figure 10.11 shows also the output of the Object Segmentation node, with the bounding boxes of the segmented objects (the bounding boxes have been shaped and coloured according to the recognized object).

10.6 Object Recognition Nodelet

Probably the most crucial node in this algorithm, the Object Recognition node takes care of processing the object point clouds and perform their classification. Needless to say,

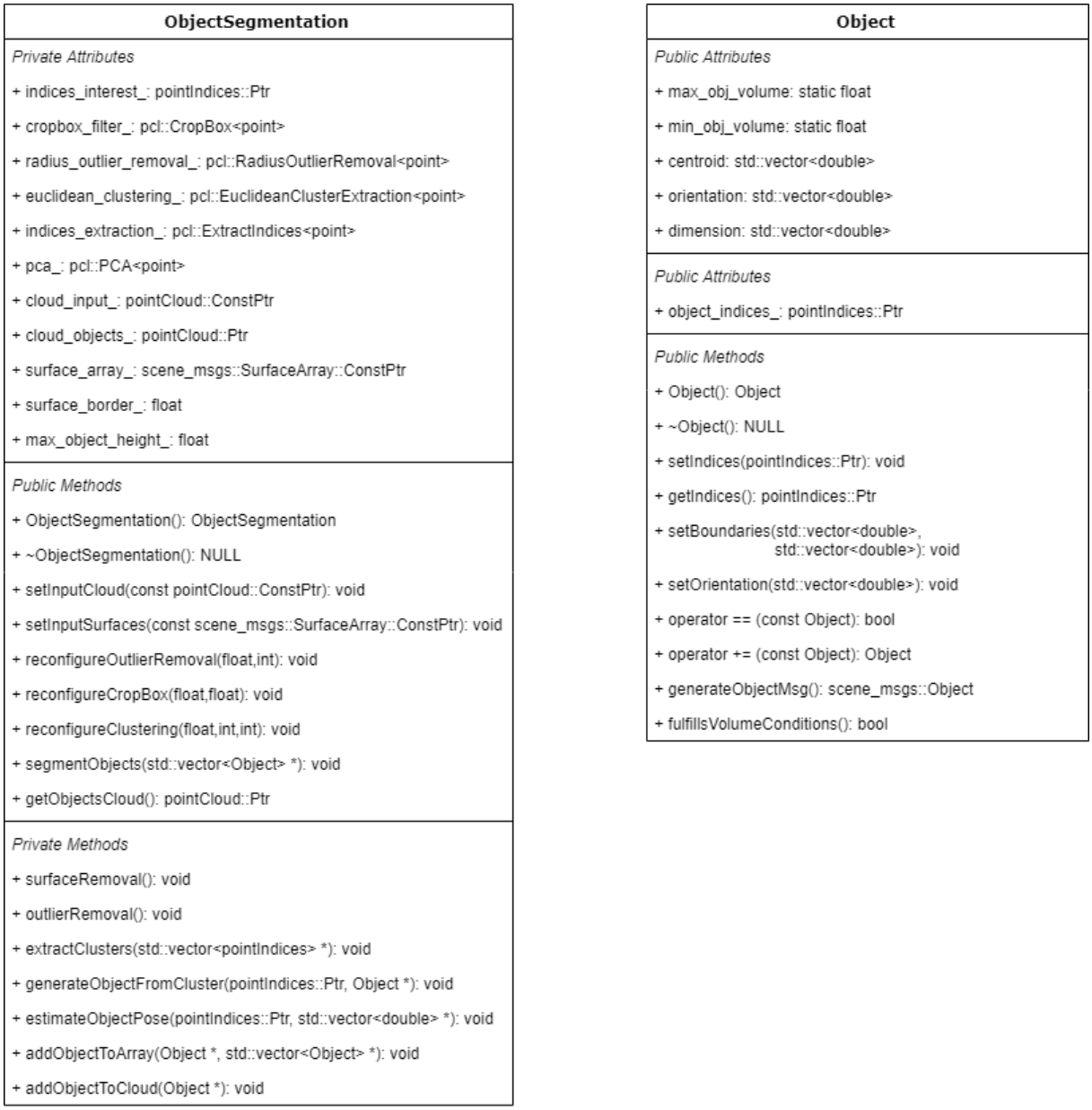


Figure 10.10: ObjectSegmentation and Object class structures.

Outl. Removal	radius_search	radius used for searching neighbours
	min_neighbors	minimum neighbors needed to be an inlier
CropBox	surface_border	border added to the BB for removing surfaces
	max_obj_height	max height considered for objects
Clustering	distance_tol	distance tolerance for clustering
	min_points	minimum points an object cluster must have
	max_points	maximum points an object cluster can have

Table 10.8: Object Segmentation node parameters.

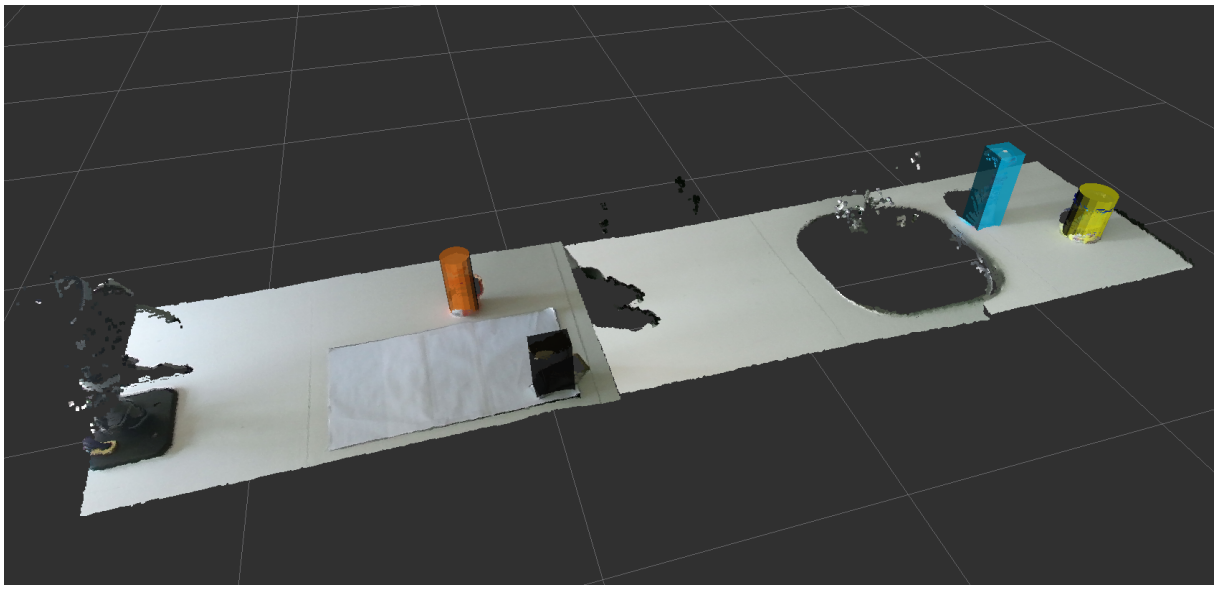


Figure 10.11: Bounding boxes of the detected by the Object Segmentation node of the three cameras.

the performance of this step has a great impact on the overall performance of the whole project. First let us go through the inputs and outputs of the node.

It receives two inputs from the Object Segmentation node: the full cut cloud that has been passed through all the steps, and the object data array containing the position, dimensions and orientation of each object cluster. The node processes the data and outputs also an object array, but this time the objects have an additional information field: the name of the object, identified by the recognition algorithm. to perform the recognition, an auxiliary object called Recogniser is used, as it is pointed out in Table 10.9.

Input topics (Subs)	full_cloud_subscriber[point cloud], objects_subscriber[object array]
Output topics (Pubs)	objects_publisher[object array]
Processing object	Recogniser

Table 10.9: Object Recognition node characteristics.

The workflow of the node is the following

1. **Get** the full cut point cloud and the object array from the Object Segmentation node.

2. **For each** object in the array

   (a) **Extract** a *rectangular* point cloud containing the object points. The relevance of a rectangular cloud (i.e. an *ordered* point cloud) is that it can be converted back to a 2D RGB image which can be used for color feature extraction.

   (b) **Recognize** the object using the defined recognition method.

3. **Publish** the resulting object data array with the recognized object labels.

The recognition is carried out by an object called Recogniser. This object has two different recognition methods: SVM and Histogram Comparison, both coded in their own classes. During the development of the project, several methods have been tested to obtain a better recognition rate and performance. Here follows an explanation of the methods used and the reasons behind using them[2].

[2] Note that this node does not have reconfigurable parameters. As the recognition must be performed using the same parameters of the training stage, it was decided that the recognition parameters should always be changed in code and not while the program is running.

10.6.1 Recognition methods

One of the most widely used methods for recognition using images is the Neural Network scheme. This method has however some requirements that are not achievable in this project. The first one is the amount of training data needed for a good recognition rate. Having a database of 50 images per class, the expected results even after fine tuning a Neural Network are not successful enough for the desired recognition rate.

The second drawback of using a Neural Network has to do with implementation issues. Even tough there are pre-trained Newtorks on the Internet for 2D object recognition, these are only able to distinguish *general* object categories, i.e. they can classify 'boxes', 'cylinders', 'cars', 'people', 'animals', but in this project the desired classification result has to be more specific. The algorithm must know whether a box is a box of cereals or a milk brick, and needs to distinguish between a Colacao and a Coffe can, which are both cylinders. Adapting a pre-trained Neural Network to achieve such degree of specification is not a trivial task.

Finally, the timing of the project requires that the recognition training should be fast enough that it can be corrected or even changed if the results are not satisfactory. This flexibility in the recognition method used is remarkably difficult to achieve in a short amount of time when using Neural Networks. That is why this option has been discarded in favour of other two commonly used methods: SVM classification and Histogram Comparison methods.

The Recognizer Class structure detailed in Figure 10.12 shows that its implementation is very lightweight and relies mostly on the code of the actual classes that perform the classification: the SVM and the HComp classes.

10.6.2 SVM Classification

Support Vector Machines offer the capabilities needed by the project. They can be trained with smaller databases, they can be easily adjusted to work with completely different descriptors, and they are fast to program and test. In this project, the OpenCV Machine Learning package has been used to implement the SVM, using the *trainAuto* function to automatically train the SVM parameters to achieve the best training recognition score.

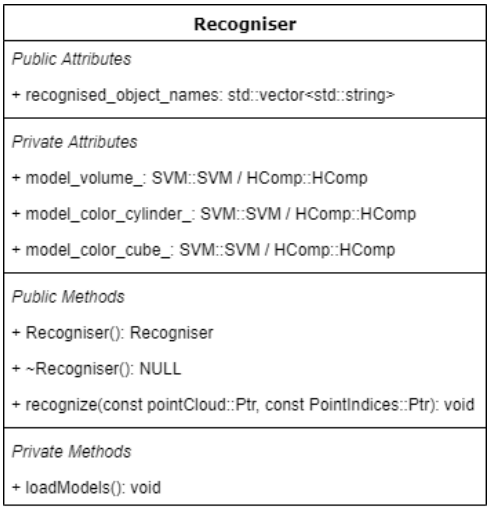


Figure 10.12: Recogniser class structure.

10.6.3 Histogram Comparison Classification

Another classification method tested in the project is to use Histogram Comparison functions. Due to the fact that the descriptors used are Histograms (even when using the SVM classification), using histogram distance functions for specific classification is also an adequate choice for the project. The HComp class is very similar to the SVM class, but its classification methods differ.

For example, the SVM needs a training phase before it can be used for recognition; whereas HComp uses directly the descriptors of the training data (or the mean of each class descriptors) without a training phase. Therefore, whereas the SVM has options to load, train and save an SVM model, the HComp class directly loads the object data set and can begin the classification right after that.

After trying different distance measures for Histogram Comparison, the correlation distance has been chosen as it provides the best classification results.

10.6.4 Descriptors

Before testing the classification methods, a decision on which descriptors to use has to be made. In this project, two different kind of descriptors have been used: a 2D HSV color histogram descriptor and a 3D local shape descriptor. A specific classification using only color descriptors is difficult to achieve, and therefore the use of 3D volume shape

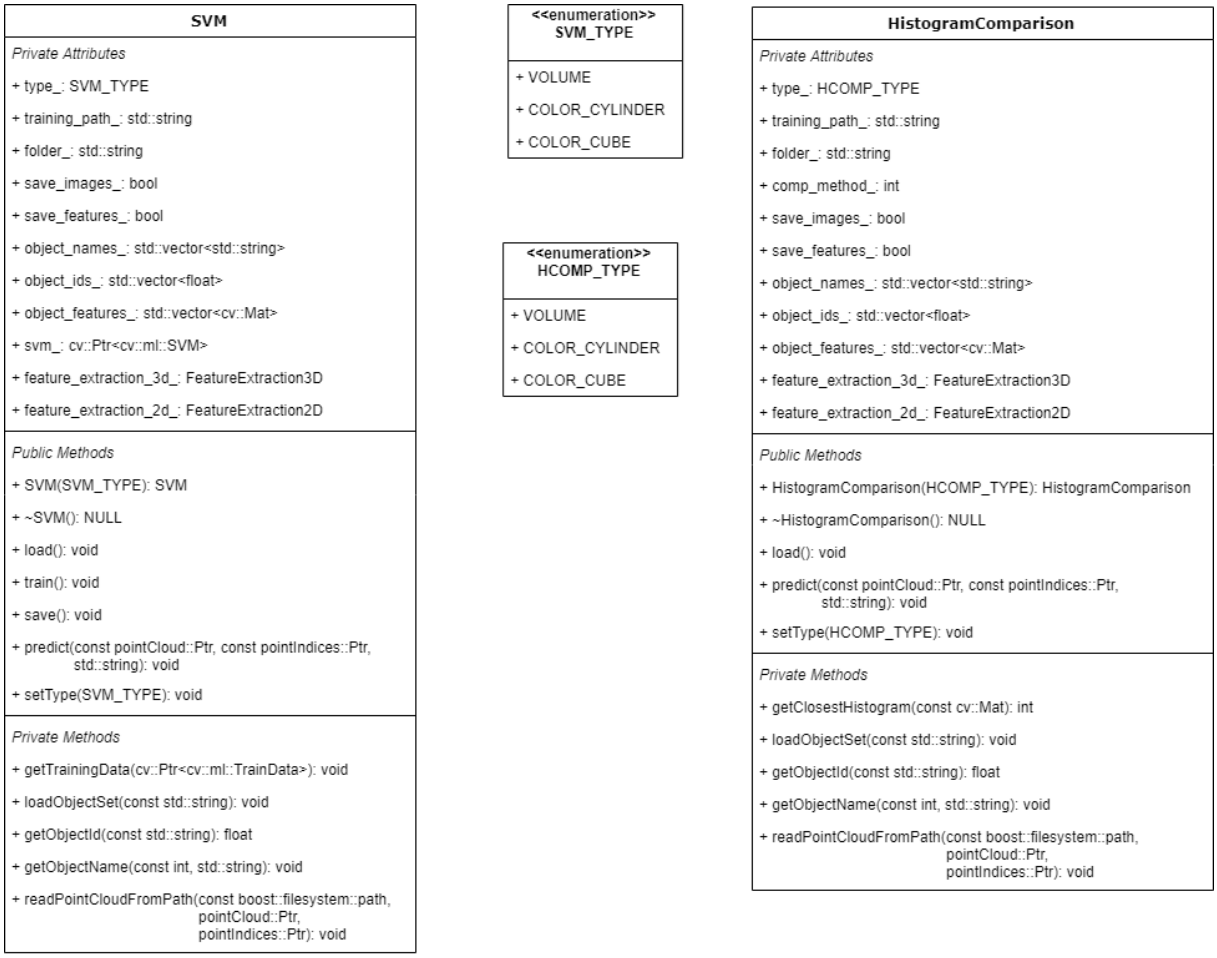


Figure 10.13: SVM and Histogram comparison class structures.

descriptors has also been tested in the project.

The implementation of the descriptor extractors is very similar. In fact, both 2D and 3D extraction methods inherit from a base class called *FeatureExtraction*. This simplifies the code and also enables to rapidly switch between one method or another with minimal code intervention. The class structures can be seen in Figure 10.14.

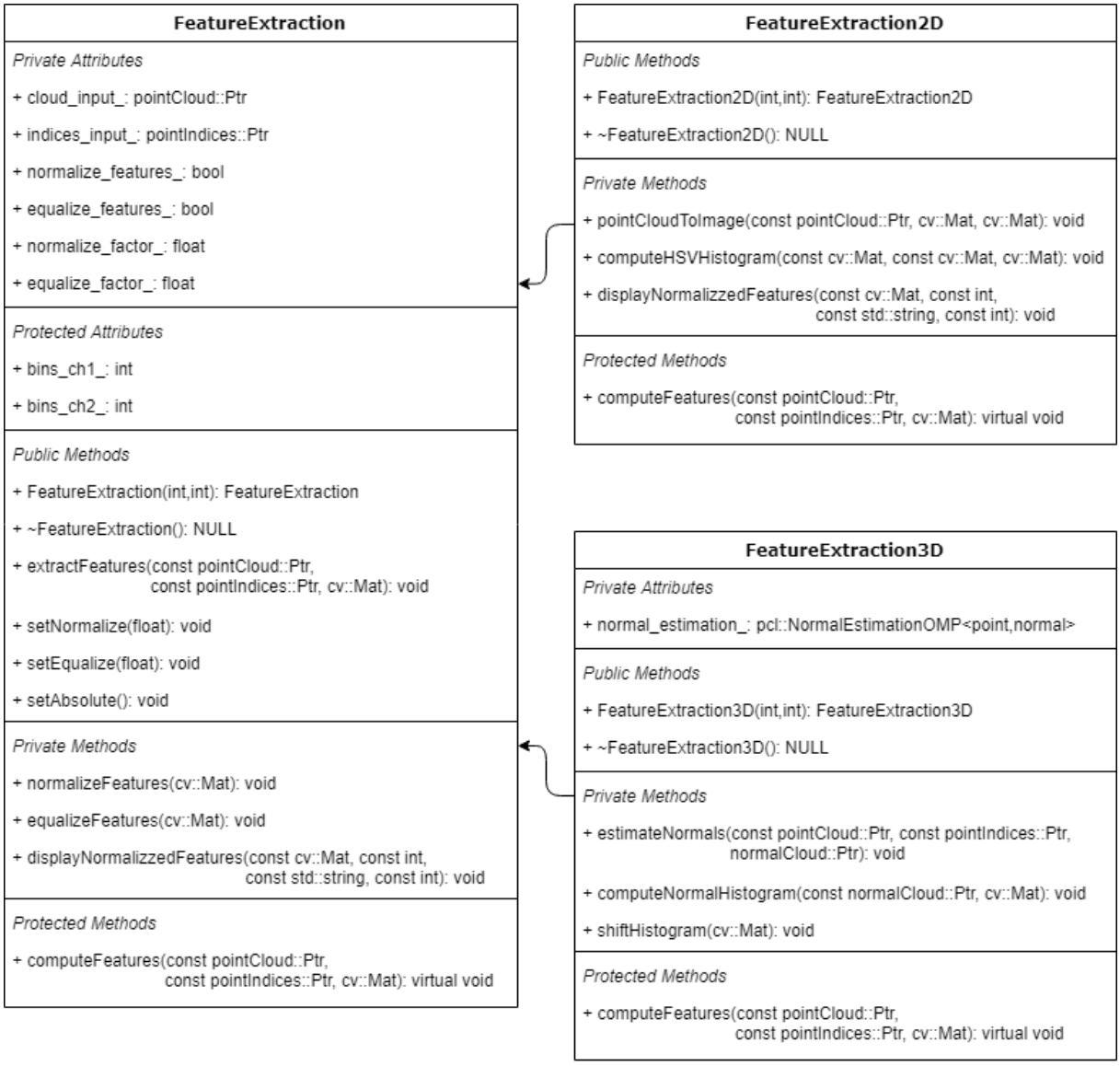


Figure 10.14: FeatureExtraction class structures.

It can be seen how the features are stored as *cv::Mat* elements with two channels or dimensions. In the case of the 2D color descriptor, these two channels correspond to Hue and Saturation channels, as the Value channel is not used for computing the descriptor.

The image is first switched from RGB to the HSV space and then a two dimensional histogram is computed using the number of bins per channel defined earlier in the class. These 2D histogram is then concatenated into a single channel histogram.

As for the 3D descriptor, a variant of the VFH descriptor has been used. First, the normals of each of the points of a cluster are estimated using a default PCL method. Then the normals are transformed to spherical coordinates and grouped into a 2D histogram whose channels correspond to ranges in the inclination and azimuth spaces. Due to the fact that only one side of the object is viewed by the cameras, the range considered for both angles is from 0 to 180 degrees. Finally, the histograms are shifted by the following rule:

1. **Get the bin with highest value** - this will be set as the local (0,0) value for both inclination and azimuth ranges
2. **Shift the other bins** - shift the bins in the 2D space by moving them relative to their original distance to the bin with highest value.

In this way, the bin with highest value will always be the first bin of the histogram. This is done to make the descriptor orientation invariant, so that the same object in different poses has always a similar histogram shape due to the shifting. In this way, for example, a box should have high values in the bins of 90 degrees azimuth and 90 degrees inclination relative to the highest bin; and a cylinder should have a row/column of very similar values at 90 degrees azimuth/inclination. The histogram is also concatenated into a one dimensional histogram before being passed to the classification method.

Finally, both descriptors have the option to be normalized (all the bin values divided by the sum of all the bins, and then multiplied by a factor if desired) or equalized (all the bin values divided by the highest bin value, and the multiplied by a factor if desired) before the classification. There is also a function available for visualizing the histograms as 2D grayscale images.

10.6.5 Two stage classification

While trying different classification methods to achieve the best results, two problems have arisen. The first one is that due to the low point cloud density given by the Kinect One cameras and the plane distortion of small objects such as the ones analyzed, using only 3D descriptors for classification proved to be unsuccessful. The second one is that the similarity of colours between objects made the use of only 2D colour descriptors not

specific enough to classify all the objects. Two approaches were tested in search of a solution to these problems.

1. **Combine 2D and 3D descriptors into a unique one** - the first and most obvious approach is to combine both 2D/3D descriptors into a single one. This, however, after being tested proved to be even less discriminant than using 2D or 3D descriptors alone. Mixing descriptors cannot be done just by concatenating them but they have to be scaled appropriately in order to adjust the relevance of each bin in the classification stage.

2. **Divide the classification in two stages** - this approach suggests that one of the descriptors might be able to correctly differentiate between to subgroups of objects, and that the second descriptor might be able to correctly discriminate all the objects inside these two subgroups.

After testing both approaches, the two stage classification process proved to be the most reliable method. By observing the objects in the database, the conclusion is drawn that almost all of them fall under the category of "box" or "cylinder". Thus it seems logical to have a first stage of classification that separates boxes and cylinders using 3D descriptors. Then the 2D color descriptor can more easily discriminate the objects by only looking at the ones of the volume subgroup. It has been considered that our database does not contain two objects with the same shape and colour, so the implementation of this approach has shown good results in the long run. This two stage classification is the final one used in the project, which explains why the Recogniser class has three models: a volume model, a color cube model and a color cylinder model.

An example of the output of the recognition node for all three camera inputs can be seen in Figure 10.15.

The descriptor used for volume classification is derived from the aforementioned VFH descriptor. Its implementation is simpler and only uses the histogram of the angles between the upwards vertical vector in world reference frame and the normals of the object point cloud. The angles are expressed in spherical coordinates and thus the histogram obtained is two-dimensional. Finally, to achieve pose invariance, the origin of the spherical coordinates is placed at the bin with the highest value, being the rest of bins relative to this one. This descriptor has proven its usefulness for discriminating between simple geometric shapes such as cubes, spheres and cylinders.

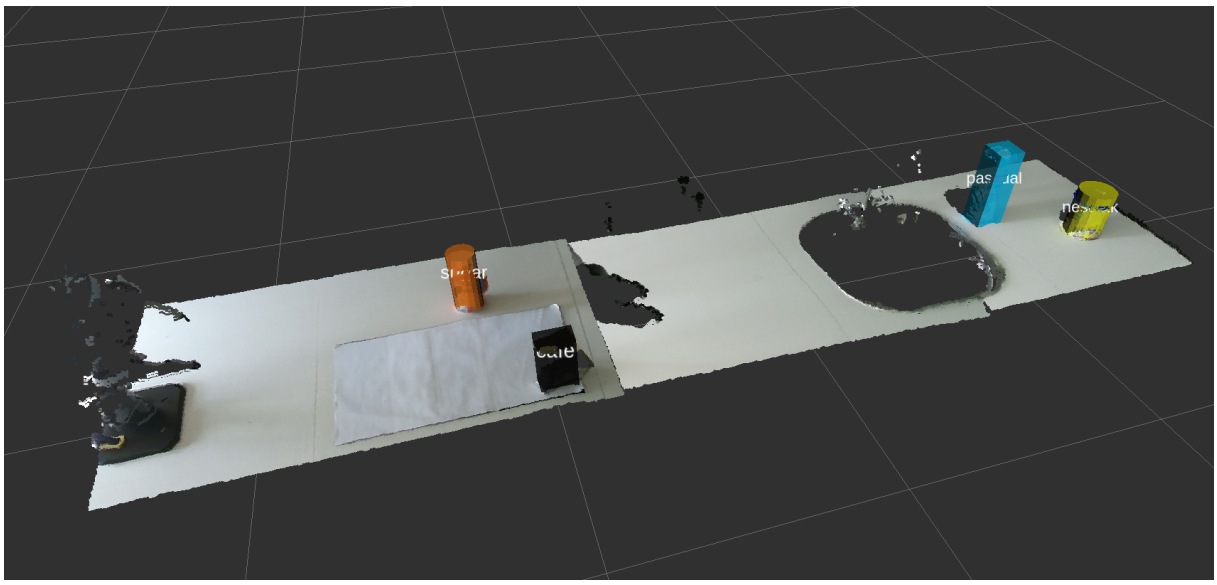


Figure 10.15: Bounding boxes with recognition names of objects in Figure 10.11.

10.7 Dispatcher Node

Up to this point, all the previous steps are implemented as nodelets that load into a nodelet manager. There is one nodelet manager per camera, so as shown in Figure 10.1, there are three Object Recognition pipelines running in parallel. The Dispatcher Node, which is not implemented as a nodelet, has the task of gathering the recognition information outputted by the three pipelines in order to *dispatch* the information as a single message to the rest of the system elements. Therefore, this node has three inputs: the recognized object arrays of each of the three cameras. It outputs the gathered information into a data topic (objects_data_publisher) as well as to three other topics for visualization. The characteristics of this node are shown in Table

Input topics (Subs)	cam1_objects_subscriber[object array], cam2_objects_subscriber[object array], cam3_objects_subscriber[object array]
Output topics (Pubs)	objects_data_publisher[Markers], objects_cloud_publisher[point cloud], objects_names_publisher[Markers], occupancy_marker_publisher[Marker]

Table 10.10: Dispatcher node characteristics.

The Object data contained in each object array is passed to a MarkerArray object where each Marker contains the name of the object, its location and its size. The topics *ob-*

ject_cloud_publisher and *object_names_publisher* are just for visualizing the results of the recognition with RViz, and the topic *occupancy_marker_publisher* publishes a Marker of type CUBE_LIST which represents a discretization of the table surface showing which cells are occupied by objects and/or robots and which cells are free. This grid is needed by the robot planner in order to avoid collisions during the robot movements.

The workflow of the node is the following

1. **Receive** the input object arrays of each camera processing pipeline
2. **For each** object in the compound object array
   (a) **Update** the occupancy grid with the cells occupied by the object
3. **Add** to the occupancy grid other predefined spaces (e.g. the Mico robot is always at the same location, and the table has a sink which is considered as non available space)
4. **For each** object in the compound object array
   (a) **Generate** the object data marker and the object name marker
5. **Publish** the generated messages and the occupancy grid to the ROS master

The duplication of the loop through all objects is coded in this way in order to maintain the same node processing structure throughout the whole developed package. The processing steps of each node have already been explained, showing that the data processing and the message generation are two different methods. This will to follow the structure might seem inefficient in this case as the Node has to loop through the object array twice. However, the maximum number of objects in the worst case scenario is less than a hundred as more would not fit on the table, so the processing time penalty of using this node structure with a duplicated loop has no effect in the algorithm processing time.

An example of the occupancy grid output of the dispatcher node is shown in Figure 10.16.

10.7.1 Message Synchronization

As the three recognition pipelines run in parallel, it may occur that the output of each of them arrives at a slightly different time to the Dispatcher node. Therefore a message synchronization procedure is require to avoid merging messages that correspond to images

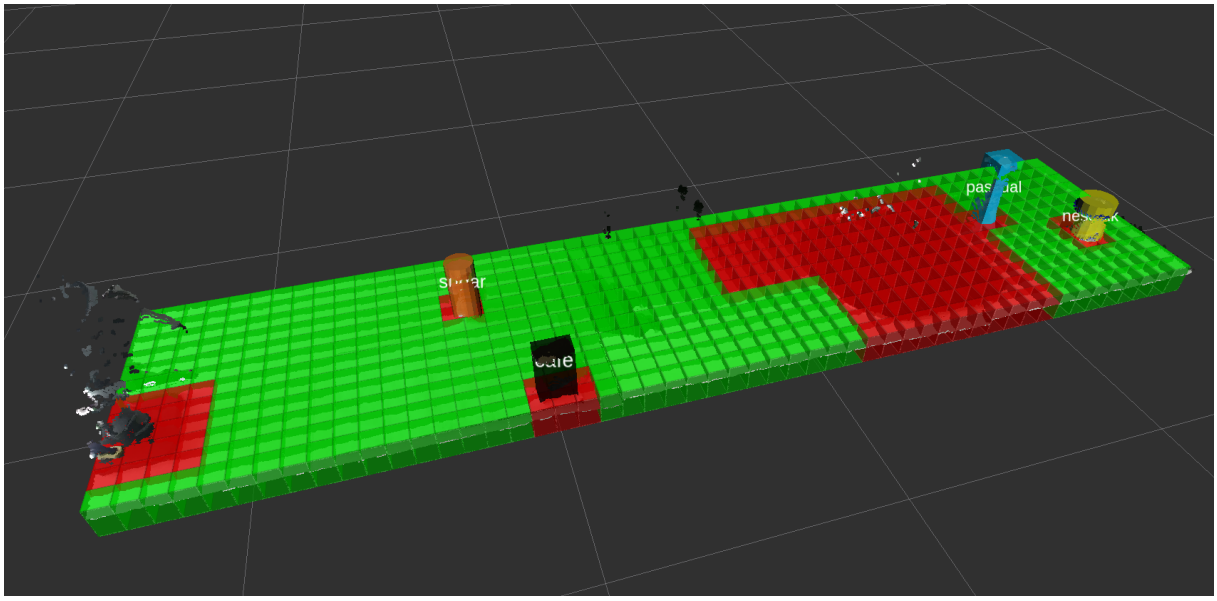


Figure 10.16: Occupancy Grid outputted by the Dispatcher node.

taken at different times. This procedure is implemented using a ROS package called *message_filters*.

The *message_filters* package provides a Synchronizer object that links several subscribers together and makes sure that the message processing is performed using input messages which have the same time stamp. There are several policies to define what is considered as the same time stamp, and in this project the Approximate Time Policy is used. This policy uses a parameter free algorithm to find the best match for each of the input topics to create a set of messages which have been received at an approximately equal time.

By using this message synchronization method with the selected policy we minimize the risk of merging together object data from different cameras that belong to different times, which may result (if the output rate is low enough) in bad robot planning and action recognition results.

10.8 ROS Graph

In the end, all the nodes and their connections can be visualized using a tool provided by ROS under the *rqt* package, called *rqt_graph*. Figure 10.17 shows the ROS graph of the full Object Recognition pipeline, with added colors to indicate groups of nodes. The colors correspond to

- **Green** - nodelet manager and child nodelets for each of the three cameras' processing pipelines.
- **Blue** - input/output topics of each nodelet manager.
- **Yellow** - Dispatcher node and its active topics.
- **Magenta** - nodes and topics related to publishing and/or receiving relative transformations from world to camera.

In the graph it can be seen how each of the above described nodes that are implemented as nodelets do not connect directly to their listeners, but instead are connected through their manager. This responds to the explained structure of nodelets and their managers: it is the manager the one who takes care of receiving messages from its child nodelets and forwarding them to the corresponding listeners that are waiting for them.

The graph also presents two nodelets which have not been described in this chapter: the *camX_bridge* and the *camX_points_xyzrgb_qhd* of each camera. These nodelets are spawned by the IAI-Kinect package in order to establish a communication bridge between ROS and the Kinect, and to deliver the XYZ-RGB point cloud resulting from retrieveing the Kinect images and applying the preprocessing computations on them. More information about how these nodelets work can be found in the official repository of the package.

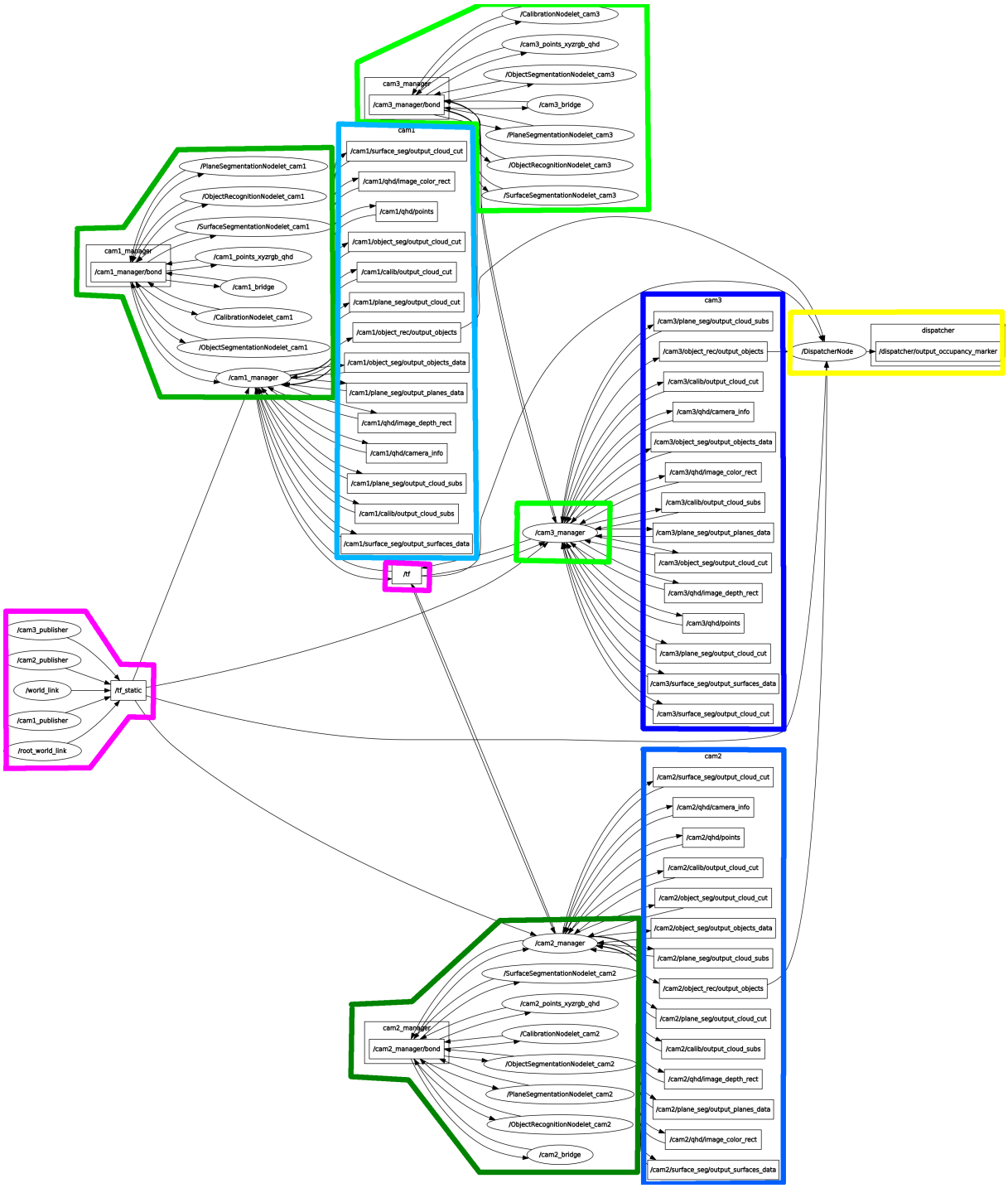


Figure 10.17: ROS graph of all the Object Recognition nodes and their connections (colored by hand).

Part V

Action Intention Recognition

Historically, efforts on recognizing human actions using computer vision data have focused either on skeletal analysis or on full body movements. Among the many conclusions drawn by the authors of papers on the subject, one underlying conclusion can be found common to them all: detecting and recognizing human actions is not an easy task.

The complexity of such a task relies on the high degree of similarity between different actions, on the intra-action and inter-subject variability, and on the difficulty of defining a feature or descriptor to precisely identify and discriminate actions. Be it either 2D image or 3D skeleton analysis, all of the reviewed papers provide unsatisfactory results or highly refined ad-hoc solutions with very low scalability properties.

Moreover, nearly all of the studies on the subject are carried out by analyzing the whole action after it is performed (i.e. analyzing a video sequence which is known to represent a whole action) and not by analyzing it in real-time while it occurs. This presents another challenge to our goal, as the references on real-time online action recognition algorithms are scarce.

The review on the state of the art of this subject has led to a slight change of perspective on how to focus the action intention recognition algorithm of this project. Unlike many of the previous studies, it has been decided that this algorithm will almost completely disregard human movements and only focus on object states and movements for recognizing actions. This is suitable to the project due to the fact that the actions to analyze always involve the interaction of the user with kitchen objects. In fact, human body movements without any kind of object interaction do not define kitchen tasks. This approach might seem restrictive, but it can also be applied to many daily situations such as manual activities.

Furthermore, as the goal of the project is to recognize action intentions, one requisite of the algorithm is that it must work online in real time. As it has already been discussed, this imposes many computation restrictions to the object recognition algorithm, but also to the action recognition part. This recognition must be able to perform without the use of computationally heavy processing or classification methods.

A more detailed analysis on the algorithm requirements leads to the following objectives, stating that the action intention recognition algorithm should

- run online in real time (at a rate equal or faster than the one required for the object recognition algorithm)
- detect intentions before the action is finished
- perform robustly using only object data from the object recognition
- be able to discriminate between a number of different tasks
- be able to correctly identify the action intention even if the user changes actions during the running time of the algorithm

After searching for previous work on approaches which fulfill all or some of the previous objectives, two publications were found relevant for the project development.

11. Activation Spreading Networks

ASNs in the context of action recognition were introduced by Saffar et al. [26] as a step towards online intent understanding, approaching the problem by dissecting high level tasks into lower level actions. ASNs are based on Hierarchical Task Networks as coded networks which represent tasks in a hierarchical structure. A summarized description of ASN architecture follows in the next subsection.

11.1 Structure

Using the hierarchical approach of HTNs, ASNs transform them into tree-like acyclic graphs where the leaves of the tree are the lowest level actions (*operators*), and the top of the tree is the main high-level task (*compound tasks*). Intermediate level nodes represent mid-level tasks (*methods*) which are needed for the high-level task. Each one of these methods is connected to a handful of leaves, indicating that there are several operators involved in carrying out a methods. In the same way, compound tasks are connected to several methods indicating the decomposition of the task. There are thus three task levels in an ASN network:

1. **Operators**: operators are the lowest-level actions, such as *grab object X*, *move object Y*. They make up the mid-level tasks, called *methods*.

2. **Methods**: methods are made up of operators, and they are connected to *compound tasks*. They represent mid-level actions.

3. **Compound tasks**: compound tasks are the highest level nodes of the graph. They are decomposed into methods and can be in turn treated as methods if they are children of another compound task. Compound tasks represent the highest level actions performed by the user, and they are the object of intention recognition.

11.2 Activation and recognition procedures

Once the ASN is built, it is used to the detect actions in the following way:

- When a graph is built, all nodes start with an activation value of zero.
- As the user starts executing a task, the object recognition algorithm uses user-defined functions to update the values of the operator nodes which are active.
- At each clock tick, all graph nodes update their activation values by adding the activation values of their child nodes multiplied by the edge weights.
- To take into account time flow, the nodes also multiply their activation value by a decay factor $d_f \in (0, 1)$.

Moreover, the authors define two types of edge connections; *sum edges* and *max edges*. Their definition is simple: a node with several child nodes connected via sum edges updates its activation value by adding the values of the child nodes multiplied by the edge weights. A node with several child nodes connected via max edges updates its activation value by replacing it with the maximum value of its child nodes which are connected via sum edges. This reasoning leads to the following conclusion: operators and methods are connected with sum edges, as the probability of a method occurring should increase when a larger number of its operators are active; whereas methods and compound tasks are connected via max edges, indicating that the probability of compound task occurring is equal to the highest probability of its child methods.

Last but not least, during the runtime of the program there must be a decision function constantly comparing the values of all nodes in order to decide whether the program

can conclude that the user has a specific intention with a large enough confidence value. This decision function must be defined with caution, as several factors can influence the outcome of the decision:

- *Node Depth* - When comparing activation values of nodes, their depth in the graph has to be taken into account, as comparing a node in the first level with another of the last level would always yield a higher result for the lower level node.
- *Relative Value* - Apart from searching for the highest node value of all graphs, it must be compared relative to the value of the best node of other graphs. This means that the decision function cannot be based exclusively on the absolute value of nodes, but also on the relative value of the best node with all the rest; as the goal of the function is to provide an intention with a certain degree of confidence. This avoids the situation of making a decision when several nodes have very similar values; thus providing a "confidence" decision threshold.
- *Minimum Threshold* - Finally, in order to avoid naive detections, the node values are not considered if they lie under a certain threshold. This additional confidence measure ensures that the detected actions have a minimum node value in order to be considered as good candidates.

11.3 Performance indicators

In order to compare and evaluate the performance of ASNs, three indicators are proposed: *early detection rate (EDR)*, *confidence of detection (COD)* and *average COD (ACOD)*.

1. **EDR**: is computed as $\frac{t_i^*}{T_i}$, where T_i is the total action runtime for intention i and t_i^* is the earliest time at which the correct intention is consistently detected.
2. **COD**: is computed as $\frac{av(i)}{max[av(j)]} \forall j \neq i$, where $av(i)$ is the activation value of the correct action and $max[av(j)]$ is the maximum activation value between the rest of actions.
3. **ACOD**: the average COD for the correct action during the whole action runtime.

A good algorithm performance should yield EDR values as close to zero as possible while providing COD values higher than one, as well as providing and ACOD value higher than one for the whole segment runtime. The authors of ASN present results of EDR between 20 and 60 percent, and ACOD values between 1.15 and 1.30.

11.4 Limitations and improvements

There are, however, two main shortcomings to this ASN implementation. The first one is that the graphs used are built beforehand based on human knowledge, as stated by themselves "[...] we assume that information about how particular activities can be performed is given in the form of HTNs [...]" This means that for each action that we want to detect, we have to manually create a new graph to represent that specific action. The second drawback is that the activation values, weights and thresholds used in the ASN are hard-coded and determined by trial and error. For a system to be autonomous, these characteristics greatly reduce the adaptability of the ASN structure to new situations and the system autonomy from human supervision.

As an effort to solve the aforementioned ASN flaws, this project presents an improved version of them: the Autonomous Activation Spreading State Network (AASSN or A2SN). The improvements proposed to the previously described ASN are described the next section.

12. Autonomous Activation Spreading State Networks

12.1 A2SN structure

The most relevant difference between ASN and the proposed A2SN structure is the codification and representation of the action graph trees. Whereas in ASN the nodes represent either operators, methods or compound tasks, in A2SN the nodes represent *system states*. A system state is defined as "the list of events occurred" up to that system state. Moreover, in ASN the graph edges represent merely connections between nodes, while in A2SN edges have much more relevance in the graph. Each A2SN edge has an associated *trigger*, being the trigger the event that transitions from a state to another. In this way, the A2SN edges can be seen as the ASN operators, which are no longer nodes but edge triggers. One must note that, unlike ASNs, in the A2SN formalism max edges do not exist and all edges are treated as sum edges.

In fact, A2SN can be seen as a Finite State Machine (FSM), with the particularity that

it has one-directional weighted transitions and that it is acyclic. That is why it is called "Autonomous Activation Spreading *State* Network".

This reconfiguration of the A2SN simplifies the structure of ASNs, by reducing the categories of nodes from 3 to 1. The hierarchy of the network is then not translated to node categories but to state transitions, and the various ways of performing an action are no longer represented as multiple children nodes but rather as different graph paths (different state sequences) which lead to the same node. The main reason behind this new structure is to allow the autonomous creation of graphs by just recording sequences of state transitions and carefully merging sequences which lead to the same state.

Another benefit of simplifying the graph structure to only one type of node and one type of edges is that it has a closer resemblance to Neural Networks. The similarity between suggests that NN learning methods might be applicable to A2SN in order to train the weights and decision thresholds of this new graph structure. Thus this decision of proposing A2SN aims to solve the two ASN limitations described before.

12.2 Autonomous creation of A2SN

In order to overcome the restriction of previous knowledge of action development in order to create an ASN, we propose a supervised learning procedure for autonomously creating ASNs. The procedure is the following:

1. Generate a small database of labeled state sequences, repeating each action a small number of times (i.e. from 2 to 10 times)
2. For each action
   (a) Generate a state transition graph for each action sequence
   (b) Merge the generated state sequences, detecting corresponding states, and merging the beginning and final state of each sequence

The result of this procedure is a A2SN state graph with one single initial "zero state", one single final state and several state paths which connect the initial and the final state. One could argue that this approach does not completely remove the previous knowledge requirement, as it requires the user to label the performed action. While true, it is certain that the proposed structure greatly reduces the time and effort required to create an ASN as it can be done by just by recording and labeling action sequences. It also allows an

unexperienced user to train the system to recognize new actions by applying the same procedure.

Figure 12.1 shows a sample sequence of the building process of an A2SN graph. The sequence used is *coffee*, which has two steps (in each steps all events can occur in a random order):

1. Mug present (91) — Coffee present (41) — Milk present (61) — Sugar present (31)

2. Coffee moving (42) — Milk moving (62) — Sugar moving (32)

As the steps have respectively 4 and 3 events that can occur in any order, the number of possible variations for action *coffee* is $4! \cdot 3! = 24 \cdot 6 = 144$.

12.3 A2SN decision function

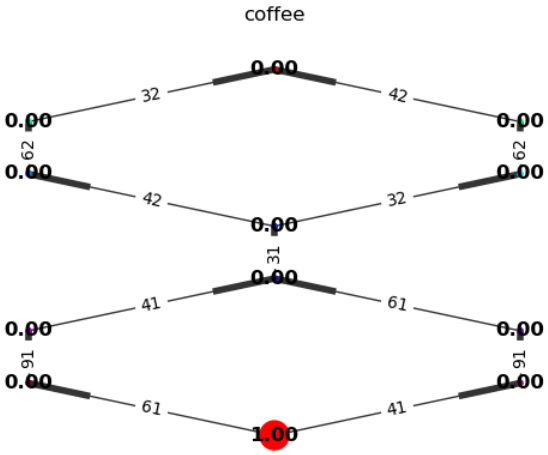
In an Action Intention Recognition run, there are several A2SN graphs which compete in parallel to be recognized as the action being performed. Each A2SN represents an action, and all of them receive the same events at the same time from the Object Recognition algorithm. In order to decide which graph is the most likely to represent the ongoing action, a decision function for recognition must be defined. This function is also critical for training the graphs, as it will provide an error metric for updating the learning parameters.

It has been mentioned before that this function must meet several requirements: take into account the Node Depth, consider the relative value between best nodes of all graphs, and have a minimum value threshold.

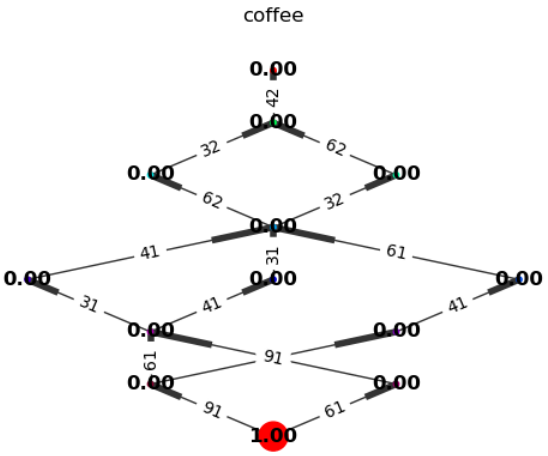
12.3.1 Node Depth

The depth of the nodes is taken into account in the following way: each node value is divided by its node depth. This criterion is fixed and cannot be changed by training, and this correction reduces the impact of deeper graph nodes on the recognition decision.

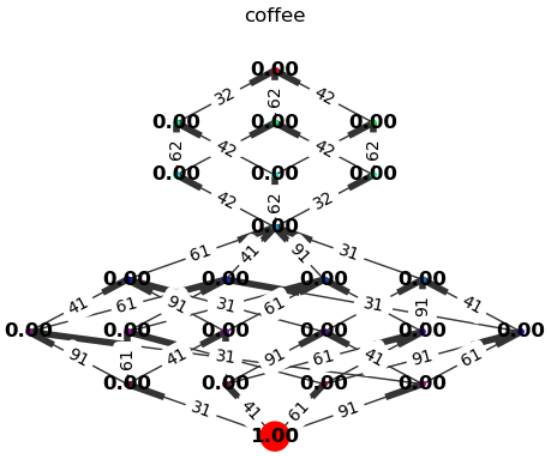
(a) 1 action repetition.



(b) 2 action repetitions.



(c) 3 action repetitions.



(d) 100 action repetitions.

Figure 12.1: Sample A2SN graph building sequence with increasing number of action repetitions.

12.3.2 Relative Value between Graphs

The relative value between graphs is used to ensure that the action detected is over a minimum confidence of detection relative to the other graphs. This metric is computed as a percentage in the following way (where $maxNodeVal_i$ refers to the maximum depth corrected node value of graph i)

$$RV_{A,B} = \frac{maxNodeVal_A - maxNodeVal_B}{maxNodeVal_B} \tag{2}$$

A positive value means that graph A has a higher node value than graph B, and a value of 1 means that the maximum node value of graph A is twice the maximum node value of graph B. The Relative Value of each graph is compared to all the others and in order for a graph to be detected as the ongoing action, the following condition must be fulfilled

$$RV_{i,j} \geq \alpha \qquad \forall j \neq i \tag{3}$$

where $1 \geq \alpha \geq 0$ is a learning parameter. This is a necessary but not sufficient condition for recognition, as it will be explained in the next section.

12.3.3 Minimum Value Threshold

The second and final condition for a graph to be a recognition candidate is that its maximum node value must be greater than a minimum threshold. This minimum threshold $\beta > 0$ is introduced to avoid early naive recognitions, and is also learned during training.

12.3.4 Final Decision Function

Taking into account these conditions, the final decision function for A2SN action intention recognition is defined as follows (where $maxNodeVal_i$ refers to the maximum depth corrected node value of graph i)

Action i is recognized subject to:

$$\begin{aligned} RV_{i,j} &\geq \alpha \qquad \forall j \neq i \\ maxNodeVal_i &\geq \beta \end{aligned} \tag{4}$$

The effect of the parameters α and β can be seen in Figure 12.2. A *cereals* action sequence is tested with a fixed value of $\beta = 10$, whereas by changing the value of α from 1 to 2 completely changes the recognition outcome.

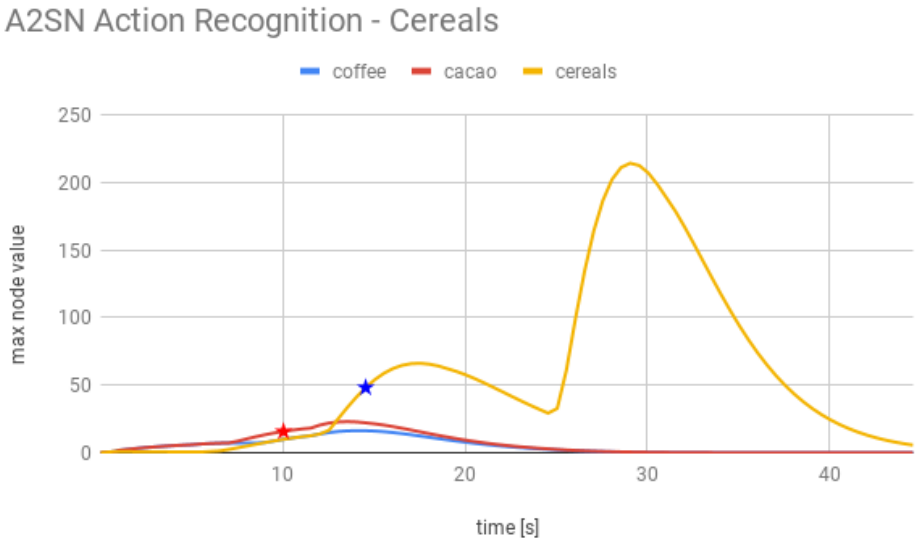


Figure 12.2: Evolution of the maximum node values for three A2SN graphs during the recognition of a *cereals* action. The red star marks the recognition outcome with $\alpha = 1$ and the blue star the recognition outcome with $\alpha = 2$.

12.4 Training A2SN networks

After having defined the decision function, one last objective of A2SN Intention Recognition must be pursued. One of the reasons behind the modifications proposed is that the algorithm parameters should be able to be trained in a Neural Network fashion. There are three parameters that can be trained, one internal and two externals to A2SN graphs:

1. **Edge weight** - weight of each edge event on each A2SN graph [internal]
2. **Relative Value threshold** α - decision function value which ensures a dominant action over the others [external]
3. **Minimum Value threshold** β - decision function value which avoids early naive detection [external]

In the current project we have focused on the possibility of external parameter learning. The reasons for focusing on training external parameters are that they have a greater influence on the detection outcome than internal parameters as it has been proved in the tests performed.

12.4.1 Performance Metrics

In order to build any training process, several metrics need to be used to asses it. The first metric is computed over the whole training set whereas the two latter are computed using data only from the correct detections.

- Percentage of correct detection (True/False Positives)
- Average EDT
- Average COD

These metrics correspond to the action intention recognition goals: an action should be recognised between other actions (percentage of correct detection) while it is being performed (average EDT). The average COD value gives a valuable insight on the quality of the detection, as it represents the confidence of the correct detections. A high detection percentage with a very low COD value would indicate that the algorithm was able to recognise the database actions but with little to none robustness.

These metrics must be included in any training process. As the proposed action recognition algorithm is a three objective optimization problem (maximise COD and percentage of correct detection while minimizing EDT), both three have to been taken into account in any training procedure applied, and they will have an influence on the trained parameters α and β. If a training method is chosen for optimizing the recognition performance, its goal would be to find the most suitable decision function for the particular set of A2SN graphs that are used in the recognition process.

13. Algorithm Implementation

In the current project, the A2SN formalism is implemented in python using the NetowkX library for graph management. Unlike the Object Recognition algorithm, the A2SN implementation for Action Intention Recognition does not have an expensive computation demand and can be implemented and run using an interpreted language. Moreover, the development of graph structures in python is considerably easier and faster than in C++. Finally, due to the fact that ROS provides python wrappers for writing nodes, both algorithms can be connected even though they are written in different programming languages.

13.1 Auxiliary classes

In order to implement A2SNs in a flexible, robust and structured way, some auxiliary classes have been defined: a State class, a KitchenObject class and a SequenceGenerator class. They provide functionalities for creating and comparing states, for creating and using representations of kitchen objects and finally for generating action sequences for creating and training A2SNs without the need of real world data, in order to speed up the implementation and tuning the A2SN creation and training procedures.

13.1.1 State

As stated before, a A2SN State is defined as "the list of events occurred" up to that system state. The first approach to implement this state description is to keep track of all the events in order. There is however a theoretical conflict with this approach when it comes to comparing states: if two states have the same events but in different order, they will not be detected as coincident states. As A2SN should cover the possibility of reaching the same state from different sequences, specially in the case where the sequences are just the same events but in different order, the approach to represent states must be slightly changed.

The representation of a state is then defined as follows: a state is the list of previous events, duplicated the number of times they have occurred. This representation overcomes the

ordering problem of the previous approach. Then in order to compare states, the program needs to check if two states have the same events with equal number of occurrences.

Besides the functionality to check for equal states, there are several extra methods that provide additional functionality tot the State class:

- **State.__init__(int list = [])**: instantiate the state object with the events in the list
- **State.copy()**: returns a deep copy of the state
- **State.hash()**: returns a hash string that identifies the state[3]
- **State.__add__(State)**: adds one state to another
- **State.__sub__(State)**: used to obtain a single event that differentiates two states, this function considers the following cases
  - State 1 has more than one difference with State 2: return -1
  - State 1 is exactly equal to State 2: return null
  - State 1 has exactly one difference with State 2: return the id of the different event
- **State.__contains__(int)**: used to check if an event is present in a State, returns true or false
- **State.__eq__(State)**: used to check if two states are equal, returns true or false

The structure of the State class is shown in Figure 13.1.

13.1.2 KitchenObject/s

The objects considered for Action Recognition are represented in code using the KitchenObject class. This class provides an id to each new object type, as well as unique identifiers for the two object events considered in the project: *object is present* and *object is moving*. This unique object event ids are then used in the representation of States and for labeling graph triggers. The class KitchenObjects is a placeholder list which holds all the

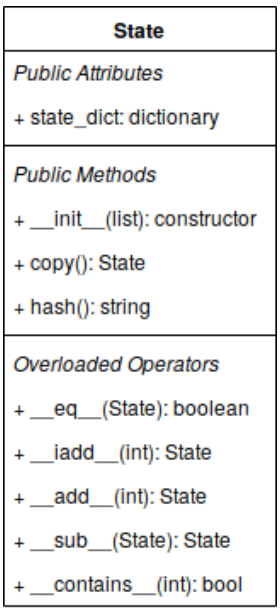


Figure 13.1: State class structure.

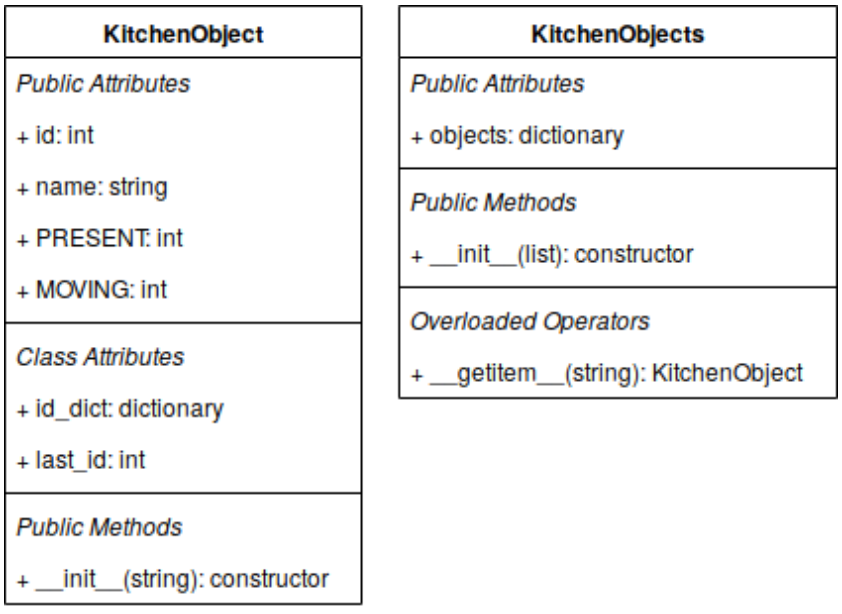


Figure 13.2: KitchenObject/s class structure.

KitchenObject instances used in the project. A representation of the KitchenObject and KitchenObjects class structure can be seen in Figure 13.2.

The KitchenObject class has just one method (its constructor) and four attributes, as well as a class attribute. They are briefly explained below

- **KitchenObject.__init__(string)**: first of all check if an object with the desired name already exists; if not assign a unique ID to the newly created object with the desired name
- **KitchenObject.id_dict**: class variable which contains "id":"name" pairs of the previously created objects
- **name**: the object name
- **id**: unique ID of the object
- **PRESENT**: unique ID representing the event *object is present*
- **MOVING**: unique ID representing the vent *object is moving*

The KitchenObjects class has two methods and one attribute, which are explained below

- **KitchenObjects.__init__(string list = [])**: create all the objects with the given or default names and add them to the objects dictionary
- **KitchenObjects.__getitem__(string)**: get the KitchenObject instance corresponding to the given object name
- **objects**: a dictionary with "object_name":KitchenObject pairs

13.1.3 SequenceGenerator

In order to speed up the development of the project, a SequenceGenerator object has been implemented. This object makes use of the KitchenObject instances and additional information about time and event relationships to generate plausible action sequences

[3] Due to the way that python has for comparing lists, two lists with the same elements can be considered as not equal when compared to each other. Thus providing a hash method is useful for comparing and identifying states.

which can be used to create and train A2SNs without the need of real world action sequence data. Obtaining real world data is a time consuming task, but the main reason to implement a sequence generator is not the time needed to record action sequences.

The objective behind this approach is that in order to record real world action sequences, the Object Recognition algorithm must be finished, and if that is the case then it is impossible to develop the Action Recognition algorithm in parallel. Therefore we decided to work with artificially generated action sequences in order to develop both algorithms in parallel and speed up the development process.

The Sequence Generator object first builds up a "master sequence". This master sequence is made up of *steps*, where each step is a list of events that may occur in any order. For example, for making a coffee we would define two steps: first bringing in a mug, a milk brick and some coffee; and then pouring the milk into the mug, adding the coffee and mixing. These two steps must be executed in order (one cannot mix, pour milk in a mug or add coffee if none of the objects is present in the scene), but the events inside the steps can appear in any order. In order to add more flexibility and realism to the Sequence Generator, each event has a minimum and maximum execution time. This time limits are used to randomly generate the timestamps and duration of each event in the sequence.

Therefore a master sequence is made up of ordered steps, and each step is made up of unordered events; where each event has a minimum and maximum execution time. A sample master sequence can be represented in the following way

- MasterSequence "Make coffee"
  - Step 1
    - Event "Mug is present", mintime = 5s, maxtime = 20s
    - Event "Milk is present", mintime = 10s, maxtime = 25s
    - Event "Coffee is present", mintime = 5s, maxtime = 15s
  - Step 2
    - Event "Milk is moving", mintime = 10s, maxtime = 40s
    - Event "Coffee is moving", mintime = 15s, maxtime = 20s

Once the "master sequence" has been defined, the user can call the *generate* method of the SequenceGenerator object. This method uses the mater sequence to generate an instance of the action sequence, selecting randomly the event order of each step and also determining the duration of each event randomly inside the specified time limits.

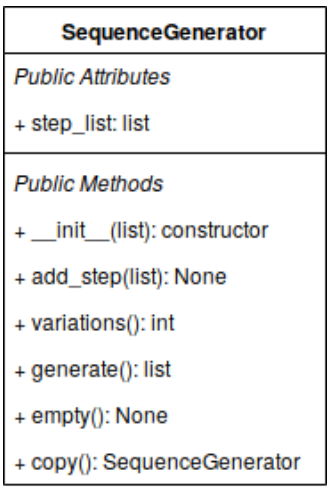


Figure 13.3: SequenceGenerator class structure.

In this way, a single SequenceGenerator object can be used to generate different sequences of the same action in order to create and train A2SNs. Figure 13.3 shows the structure of the SequenceGenerator class. Its methods and attributes are explained below

- **SequenceGenerator.__init__(nested list = []):** initialize the step list of the master sequence with the given steps
- **SequenceGenerator.add_step(list = []):** add a step to the master sequence
- **SequenceGenerator.variations():** compute the number of possible variations of the master sequence
- **SequenceGenerator.generate():** generate a sample sequence based on the master sequence
- **SequenceGenerator.empty():** remove all the steps from the master sequence
- **SequenceGenerator.copy():** return a deep copy of the object
- **step_list:** nested list of steps of the master sequence

13.2 A2SN implementation

The development of these auxiliary classes is oriented to implementing the A2SN formalism in a simple yet flexible way. It has been decided to divide this implementation into

two different classes: A2SN_BUILD and A2SN_RUN. The first one is used for building A2SNs from a database of action sequences and saving the generated graphs, whereas the second one is used to load the saved graphs and run the Action Recognition in real time.

Due to the fact that these classes share many functionalities, a base class has been created from where they both inherit the common features: A2SN_BASE. The three classes are explained below.

13.2.1 A2SN_BASE

This class holds methods and attributes that are common to all A2SNs, whether they are used for building or running graphs. Its most important attribute is the NetworkX Graph, which stores all the information about states (graph nodes) and transitions (graph edges). Both the graph nodes and edges store all the information needed to run the Action Recognition, information that is generated during the building and training phases. The graph information is stored in the following way

- **A2SN_BASE.graph**: NetworkX Directed Graph
  - **graph.nodes**: they represent the known states that make up the action sequence, and are added to the graph in the building phase
    - *state*: State object that identifies the node
    - *value*: float that stores the activation value of the node
    - *depth*: integer that stores the distance from the node to the start node
    - *color*: float tuple that stores a unique color for plotting
    - *end*: boolean which indicates if the node is the last of the graph
  - **graph.edges**: directed edges that represent the transitions between states
    - *weight*: float value used to multiply the activation value of the sender node
    - *factor*: float normalization factor applied to the edge weight
    - *trigger*: integer corresponding to the event that triggers the 'state transition' and passing the activation message

Many of the methods of the base class are oriented to generate a visually appealing plot of the graph. The class also provides methods for saving the graph to a YAML file and loading a previously saved one, as well as utility methods for reordering the graph structure, setting the edge weight factors and generating a node depth map. The attributes needed to provide all these functionalities and the methods themselves are detailed below

- **A2SN_BASE.__init__()**: initialize the graph and add the starting node with an empty State
- **A2SN_BASE.plot(int, string)**: plot the graph to the given matplotlib figure with the desired title
- **A2SN_BASE.relabel_nodes()**: restructure the nodes based on their depth and relabel them so that all the other algorithms work correctly
- **A2SN_BASE.export(string)**: save the graph structure and values to the given YAML file
- **A2SN_BASE.load(string)**: load the graph structure and values from a given YAML file
- **A2SN_BASE.inherit(A2SN_BASE)**: inherit the graph structure and values from an existing A2SN_BASE instance
- **A2SN_BASE.__node_depth_map()**: compute a map from each node to its depth value
- **A2SN_BASE.__set_edge_factors()**: set the factor of each edge based on the number of edges that have the same destination node
- **A2SN_BASE.__node_color_map()**: compute a color map of the nodes based on their depth in the graph
- **A2SN_BASE.__edge_color_map()**: compute a color map of the edges based on their weight
- **A2SN_BASE.__node_label_map()**: compute the a node label map with their activation values as labels
- **A2SN_BASE.__edge_labels()**: return a list of edge labels with the edge trigger events as labels
- **A2SN_BASE.__node_size_map()**: compute a node size map proportional to the node activation values
- **A2SN_BASE.__node_position()**: compute the position of each node to untangle the graph for plotting
- **graph**: NetworkX DiGraph with all the graph information
- **node_count**: integer that keeps track of the number of nodes present in the graph
- **end_node**: integer that keeps track of the end node ID

- **latest_state**: State used to hold a copy of the latest State of the graph
- **figure**: integer that points to the matplotlib figure used to plot the graph
- **fig_title**: string that stores the title used when plotting the graph

13.2.2 A2SN_BUILD

On top of this base class, the A2SN versions for building and running graphs provide specific attributes and functions in order to perform their respective tasks. The graph building class offers two methods for adding nodes to the graph, one method for merging two graphs together, and a final method to end the graph building process. This subclass does not add any additional attributes to the base class. The functional aspects of the subclass methods are briefly explained below

- **A2SN_BUILD.add_node_by_state(Sate, State, int)**: add a new node to the graph using the provided new state input. Two additional parameters can be given if known: the parent state and the transition event. If these parameters are unknown, the latest state of the graph is used as the parent and the difference between states is computed to get the transition event. If the parent and the new state are the same, if they differ by more than one event or if the new state lacks an event present in the parent state, the function returns an error. Furthermore, if an existing graph node already has the new given state, the nodes are merged.
- **A2SN_BUILD.add_node_by_event(int, State)**: add a new node to the graph using the provided transition event. This function uses the latest state of the graph to build a new state and pass this information to the add_node_by_state function.
- **A2SN_BUILD.__iadd__(A2SN_BASE)**: merges an existing graph to the A2SN own graph. This method is used to incrementally build an A2SN graph from multiple 'simple' graphs generated from only one sequence.
- **A2SN_BUILD.end()**: end the graph building process by relabeling the nodes, setting the end node and computing the edge factors.

13.2.3 A2SN_RUN

Even though it is theoretically simple to run a pre-built graph and update its values with real time data, the implementation of the A2SN class for running the Action Recognition

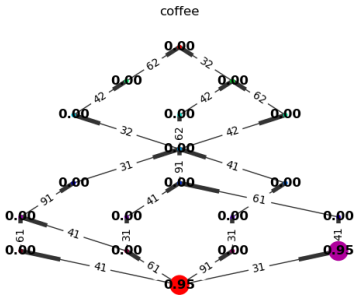
algorithm is critical to ensure a correct outcome of the whole process. Some crucial decisions must be made before writing the code of this class as they will have a great impact on the action recognition procedure. The structural and conceptual differences between ASNs and A2SNs have an effect on the behaviour of the graph when exposed to inputs, and these differences must be taken into account in order to preserve the theoretical background for ASN action recognition when shifting to a A2SN structure.

Input handling

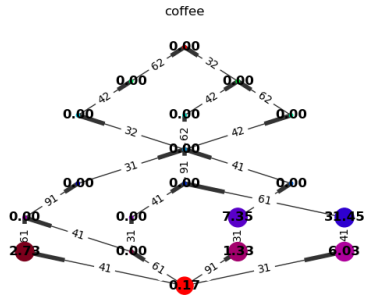
The first difference comes from the fact that with an ASN structure, the inputs to the system are translated into activation values of its operator nodes. This means that if an input occurs constantly over time, the activation value of its correspondent operator nodes will also be constant over time. In the A2SN formalism, however, inputs are related to edge triggers and have no relation to the fluctuation of node activation values. This means that if an input occurs constantly over time, the activation values of nodes will not be affected by it; as they are only affected by the decay factor and the input messages they receive. However, the fact that the node activation values do not change with relation to the input events means that if all nodes start with a value of zero, they will never increase their value throughout the whole run.

In order to solve this issue, a slightly different approach has been taken. In an A2SN graph, the starting node begins with an activation value of 1. This value is only affected by the decay factor, as the starting node does not receive messages and only serves as a sender, so it will decrease at each clock tick. This characteristic adds another time-related behaviour to the A2SN: as time goes by, the nodes closer to the starting point will have less impact on the activation value flow of the graph, eventually passing on their potential to the deeper nodes. This produces a characteristic wave-like spreading pattern of the activation values throughout the graph, which responds to a logical deduction: as time goes by, the nodes deeper in the graph should have higher activation values than the ones at the beginning because it is expected that the user progresses through the different stages of the action.

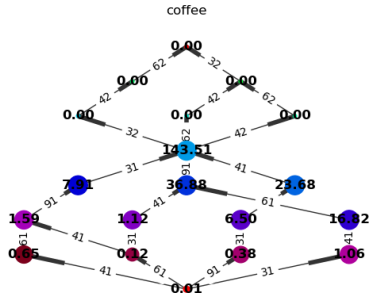
An example of the evolution of graph values over a 60 seconds action sequence can be seen in Figure 13.4, in increments of 8 seconds. The sequence played is made up by the following events: 31-41-91-61-32-62-42. It is worth noticing that this particular *coffee* sequence is not present in the A2SN graph used, which has been trained with only 5 action sequence samples. Despite not having seen this particular sequence before, the graph was able to recognize the action.



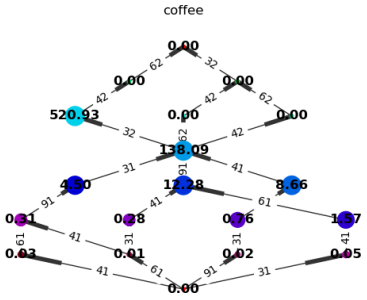
(a) t = 0s



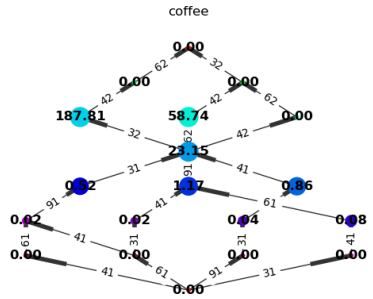
(b) t = 8s



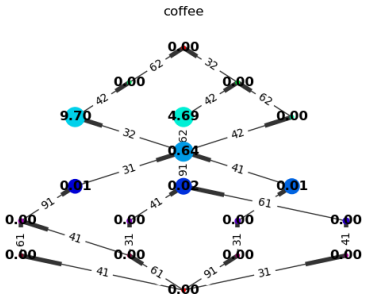
(c) t = 16s



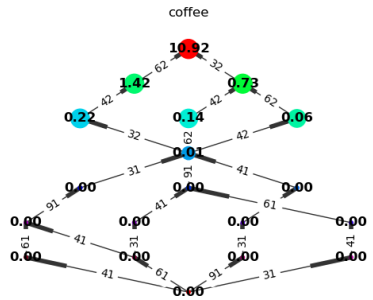
(d) t = 24s



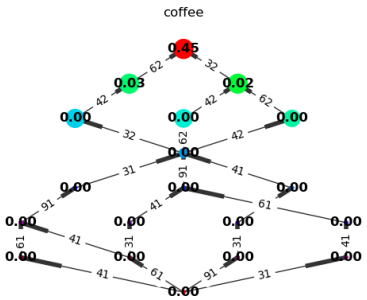
(e) t = 32s



(f) 1t = 40 s



(g) t = 48s



(h) t = 52s

Figure 13.4: Evolution of a *coffee* A2SN graph over the execution of the 60 second coffee sequence: 31-41-91-61-32-62-42.

Activation message flow

The second noticeable difference between both approaches is that an ASN sends messages through all its edges at each clock tick. An A2SN, however, only sends messages through the edges whose trigger event is active at the given clock tick. This provides a behaviour similar to the operator nodes in ASNs: the ASN operator nodes have an activation value of zero when there is not any input related to them (effectively sending a message of value zero to its child nodes); and the A2SN edges are deactivated when their trigger event is not active (thus also stopping the value propagation). A2SN graphs thus give more importance to the order of actions than ASNs, so at first sight it might seem that they are more restrictive when it comes to recognizing actions performed in a way not previously seen by the network. However, this apparent restriction is bypassed by the fact that A2SN graphs can be built incrementally from different sequences of the same action, whereas ASN structures are hard coded by a programmer which results in a more strict and error prone approach for unknown sequence recognition.

Activation values evolution

The third and perhaps most decisive difference between ASN and A2SN comes from the aforementioned *max edges*. In ASN, a max edge represents not only a quantitative but also a qualitative relationship between nodes. As it has been previously explained, these edges connect several *method* nodes with a single *compound task* node, in such a way that the activation value of the compound task is the highest of its child nodes connected via max edges. There are two relevant implications to this characteristic of ASN graphs:

1. The quantitative implication is that max edges work as an slowing factor for activation value propagation, as the receiving node does not increment its value with the incoming message but instead replaces its value with the highest incoming message, thus serving as a sort of saturation gate for all the nodes below.

2. The qualitative implication is that max edges establish a hierarchy between nodes, making a difference between low level greedy nodes which are updated via sum edges and high level discriminant nodes that only accept one of their incoming messages.

The suppression of max edges is one of the characteristics of A2SN graphs, and has two effects that develop from these two implications. Due to the fact that all A2SN nodes are connected with sum edges, there is no saturation in the activation value propagation, meaning that the criteria for comparing activation values in order to decide the action with the highest probability cannot be the same than the ASN rules. Nonetheless, with a proper study of A2SN behaviour suitable criteria can be found.

The main reason for suppressing max edges has to do with the qualitative implication of max edges. The fact that max edges establish a distinction between *methods* and *compound tasks* increases the need of a programmer to generate action graphs. This qualitative difference, required for the correct performance of the ASN, cannot be easily detected by an autonomous algorithm; as there are methods that in some actions behave as compound tasks and vice-versa. In some cases, even in the same ASN the same method appears as a compound task deeper in the graph. That is why a graph structure with only sum edges and that reduces the meaning of nodes to system states is much more suitable for autonomous graph building, one of the objectives of this project.

Finally, this simplified node representation structure has a close resemblance to classic Neural Networks, opening the action recognition algorithm to the wide variety of NN training methods to use for learning. In the end, these learning methods used to train the weights of the edges could also end up creating connections similar to max edges; for example if after the learning phase one node has three children connected with very edge weights close to zero and one child with an edge weight close to one.

Consecutive Action Recognition

The last difference between both approaches is how do they perform against two specific situations: the case where a user stops the action being performed to start a new one, and the case where the user ends an action and starts a new one immediately after.

In long term program executions where one action follows another, A2SN has a core problem: as the initial node value always decreases, at some point no event is able to propagate a message strong enough to activate the recognition and the graph then has a limited life cycle. ASN graphs, however, present high activation values for the operator nodes each time their respective event is active, so their life cycle is infinite.

However, in the case of a spontaneous action change, ASN have a slow response time because the graph keeps propagating activation values until their value is so low that the information from the previous action has no effect on the new coming information. A2SN, however, due to their wave-like behaviour, have the potential of a much faster response time than ASN. The problem is that in order to generate a new activation value wave, the initial node value would have to be reset, and the algorithm does not have the necessary knowledge to determine when to reset it.

The solution proposed to add this feature to A2SN graphs is to fix a life cycle for A2SN based on a comparison between the highest activation value of each graph nodes and the time elapsed since the graph batch started receiving input (called graph *utility*). In fact, the current project proposes to go a step further: initialise a new batch of A2SN graphs

at a constant rate, and monitorize the running graphs to remove those who have finished their life cycles.

In this way, the both cases presented above are solved, as there are new graphs being constantly created which can capture spontaneous action changes or the beginning of new actions. This approach also adds another flexibility factor to the action recognition algorithm: for the case of recognizing actions which have very different runtimes, having graphs starting at different times can capture actions that take a wide range of different times to complete.

A2SN_RUN Structure

Taking into consideration these A2SN properties, the class that implements the graph for running the recognition algorithm is detailed below.

- **A2SN_RUN.update_events(list)** - thread safe function to update the active events used when updating the graph node values
- **A2SN_RUN.reset()** - set the node values to zero and reset the rest of graph variables for a new recognition run
- **A2SN_RUN.plot()** - this method overrides the *plot* method of the base class A2SN_BASE in order to make it thread safe
- **A2SN_RUN.update()** - safe thread function to update the activation value of the graph nodes. It updates the values in the following way: first, the list of active events is retrieved, and for each active event the matching trigger edges are marked as triggered and the activation messages are generated. Then, the function loops through all nodes. If a node has one or more input activation messages in queue, it performs the value update by adding the input messages. Even if a node does not have any input message, all nodes end the update function by multiplying their value by the decay factor.
- **A2SN_RUN.decay_factor** - class attribute which stores the decay factor (common to all graphs) as a float, between zero and one.

13.2.4 Running the Action Intention Recognition

In order to run the Action Recognition, the tasks are divided into three threads, hence the need to make the A2SN_RUN methods thread safe. As the input to the algorithm comes from a ROS topic, the main thread is the one that handles the ROS node which subscribes to the Dispatcher information.

This node has two attributes that are a pool of A2SN_RUN objects to update their node values and an object in charge of plotting the graphs if desired. These two objects span their own running threads, so in the end the Action Intention Algorithm has three processes running in parallel

1. A ROS node process which listens to input messages from the Object Recognition and updates the active event list of the graphs when a new message is received
2. A A2SN_pool process which updates the node values of the graphs at a given rate (equal or higher to the real time specification)
3. A plotter object that references the aforementioned A2SN_pool in order to plot the graphs at a desired rate (preferably slower than the update rate)

These task distribution makes it possible to have the A2SN update loop running at any desired fixed rate, and also reducing the impact of displaying figures on the node value update process time. As it has been explained before, the graphs update their node values at each clock tick, which is a constant rate. If they updated their values with each incoming message, this rate would be variable (as the object recognition publishing rate is variable) and the use of the decay factor would introduce unwanted behaviour in the system.

Two more execution modes are added for simulation purposes: a mode to simulate an event sequence in real time (which replaces the ROS inputs for simulated events), and a mode to simulate a large number of virtual sequences using Discrete Event Simulation. This last execution mode enables testing A2SN Action Recognition with a large number of virtual sequences in a short amount of time. Figure 13.5 shows a representation of the processes that run in parallel in each of the three execution modes.

Main Thread (Real Time Simulation)		
Updater Thread	**Plotter Thread**	**Sequence Thread**
Loop (2 Hz)	*Loop (1Hz)*	*Simulation Loop*
-Update A2SN values -Check action recognition conditions	-Plot graphs	-Wait until next event -Update A2SN active events list

Main Thread (Execution)		
Updater Thread	**Plotter Thread**	**ROS Thread**
Loop (2 Hz)	*Loop (1Hz)*	*ROS Loop*
-Update A2SN values -Check action recognition conditions	-Plot graphs	-Listen to dispatcher messages -Update A2SN active events list

Main Thread (Discrete Event Simulation)
Simulation Loop
-Get new event from sequence -Until event time is reached: +Update A2SN values +Check action recognition conditions +Increment simulation time by 0.5 seconds -Update A2SN active events list

Figure 13.5: Structure of each execution mode for Action Intention Recognition.

Part VI

Results

After developing both the object recognition and the action intention recognition algorithms, several tests have been performed to study their performance according to the project objectives. Three tests have been performed: one to assess the robustness and performance of the object recognition, one to test the feasibility of the proposed approach to achieve action intention recognition, and a final one to check the overall performance of the coupled systems.

14. Object Recognition Results

As stated in the Introduction chapter, three goals were set when regarding the performance of object recognition:

- robustly detect a defined set of kitchen objects
- precisely position the detected objects in a given world reference frame
- work in real-time

The set of kitchen objects considered is made up of 10 different objects, categorized by volume as either *cube* or *cylinder* shaped objects. The list of objects and their categorization is shown in Table 14.1.

The classification works in two steps, a first step to categorize by volume and then a second step to categorize by colour only taking into account objects of the same volume category. This decreases the difficulties posed by a colour only classification by reducing the number of objects to classify.

In order to perform the classification, two different methods have been tested: a multi class SVM classification and a Histogram Distance comparison using different metrics (Intersection, Cross Correlation, Chi Square and Battacharyya). Both SVM and Histogram

Object name	Volume category
acorsugar	cube
pascual	cube
marcillacafe	cube
lletnostra	cube
nesquik	cylinder
colacao	cylinder
nescafe	cylinder
yogurt	cylinder
sugar	cylinder
mug	cylinder

Table 14.1: Object list categorized by volume.

Comparison have been tested for volume, cube colour and cylinder colour classification. After performing tests over a video stream of over 1000 frames (10 minutes of video at a frame rate of 1.8 Hz), the conclusions are that SVM outperforms Histogram Classification for categorizing objects by volume, whereas for color classification both methods present similar results.

In the end, it has been decided to use SVM for volume classification and Histogram Comparison with Intersection distance for color classification. The reason to use Histograms instead of SVM for color is that unlike SVM it does not require training, and it is also suggested as the appropriate method for color classification by many of the authors cited in the State of the Art review of this project. The SVM parameters have been defined using an auto-train function provided by the OpenCV codebase that finds optimal parameters for the data provided.

14.1 Recognition by Volume and Color

The final confusion matrix for the 10 objects presented above can be seen in Figure 14.1. Several conclusions can be drawn from an analysis of the matrix.

1. Three items are not recognized a single time: *colacao*, *sugar* and *yogurt*. *Colacao* and *yogurt* are confused with *nescafe*, whereas *sugar* is confused with *mug*.

2. One item is only recognized well 33% of times: *pascual*. It is confused mainly with *nescafe*, a 66% of times.

REAL \ PRED	lletnostra	mug	pascual	marcillacafe	colacao	nescafe	sugar	nesquik	yogurt	acorsugar	
lletnostra	0.99	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
mug	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
pascual	0.00	0.01	0.33	0.01	0.00	0.66	0.00	0.00	0.00	0.00	1.00
marcillacafe	0.00	0.00	0.00	0.99	0.00	0.01	0.00	0.00	0.00	0.00	1.00
colacao	0.00	0.02	0.00	0.00	0.00	0.98	0.00	0.00	0.00	0.00	1.00
nescafe	0.00	0.01	0.00	0.00	0.00	0.99	0.00	0.00	0.00	0.00	1.00
sugar	0.00	0.80	0.00	0.00	0.00	0.19	0.00	0.01	0.00	0.00	1.00
nesquik	0.00	0.00	0.00	0.03	0.00	0.01	0.00	0.97	0.00	0.00	1.00
yogurt	0.00	0.00	0.00	0.01	0.00	0.99	0.00	0.00	0.00	0.01	1.00
acorsugar	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00
	0.99	1.84	0.33	1.03	0.00	3.82	0.00	0.97	0.00	1.01	

Figure 14.1: Normalized Confusion Matrix of the Object Recognition results. Rows represent true labels and columns predicted labels.

3. The two items *mug* and *nescafe* are the ones that almost all of the bad recognitions point to.

After analyzing the results, there is one source of error that is proposed as the cause of two of these problems, and it is the following: both *mug* and *nescafe* have black and brown tonalities. If seen in the HSV color space, these tonalities have very low Value and the same Hue and Saturation that red and orange tonalities (see Figure 14.2). The *yogurt* object is white, so its HSV values can also be confused with any bright color of the whole spectrum. This is the reason why those objects are confused with the black and brown tonality objects.

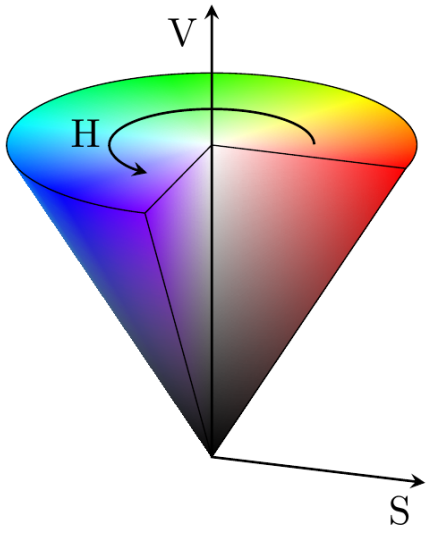


Figure 14.2: HSV color space cone. [9]

In the case of the *pascual* object, the error source is different. Here the problem lies in the fact that the object is recognised as a *cylinder* instead of as a *cube*, and therefore the color classification cannot retrieve the correct class as it uses the *cylinder* database for finding a match. The reason for this confusion comes from the fact that the *pascual* has a hexagonal base. This shape has been categorized as *cube* for training but it is also close to a cylindrical shape, thus leading to the confusion in volume recognition.

As for the objects position, it has been manually measured and for all tests the average position deviation is around 1 centimeter, even when working with cluttered scenes (see Figure 14.3). Given the placement of the Kinect cameras, the algorithm is able to recognize and position objects in a scene with a high density of instances close to each other.

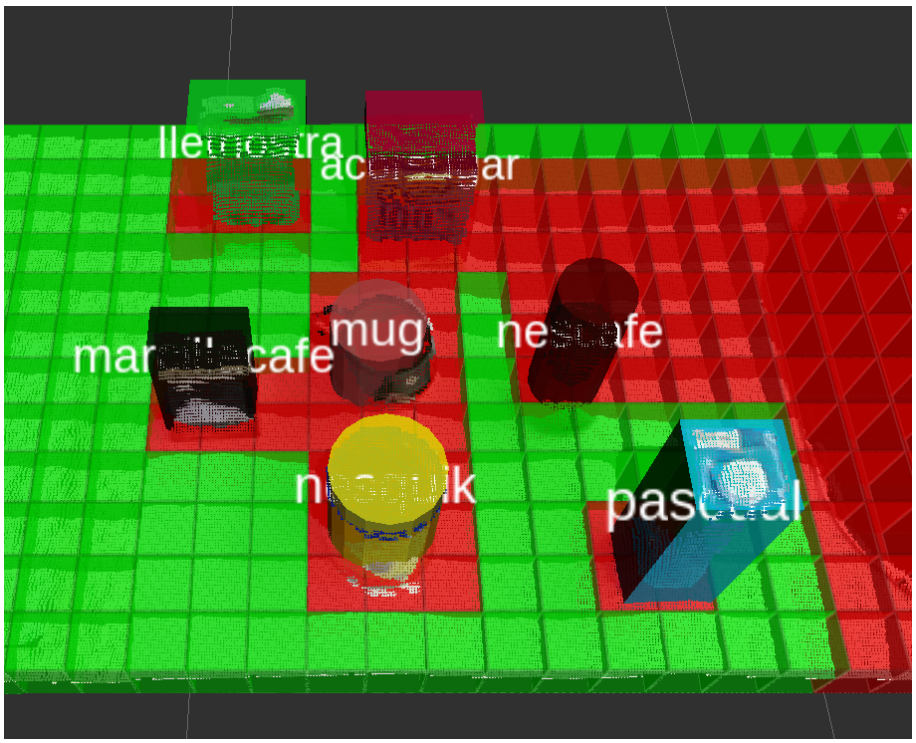


Figure 14.3: Recognition of objects in a cluttered scene.

14.2 Algorithm Performance

Another goal for the Object Recognition Algorithm is that it must perform in real time, as it has to provide information to the Action Intention Recognition which works in real time. This has resulted in many code optimizations throughout the project development, and in the end each step of the object recognition process has been timed to test if the final algorithm can perform in real time or not.

The real time requirement for the project is that the algorithms must provide results at a minimum rate of 1 Hz. A graph showing the processing times of the whole Object Recognition pipeline and each of its steps is shown in Figure 14.4.

It can be seen how the steps which use the subsampled version of the point cloud (Plane and Surface Segmentation) are faster than the ones that use the full cloud data (Object Segmentation and Recognition). In fact, on average the Plane and Surface Segmentation steps are a 25-30% faster.

Moreover, it can be seen how the maximum processing time of the whole process (corresponding to the rate at which the Dispatcher node publishes messages) lasts a maximum

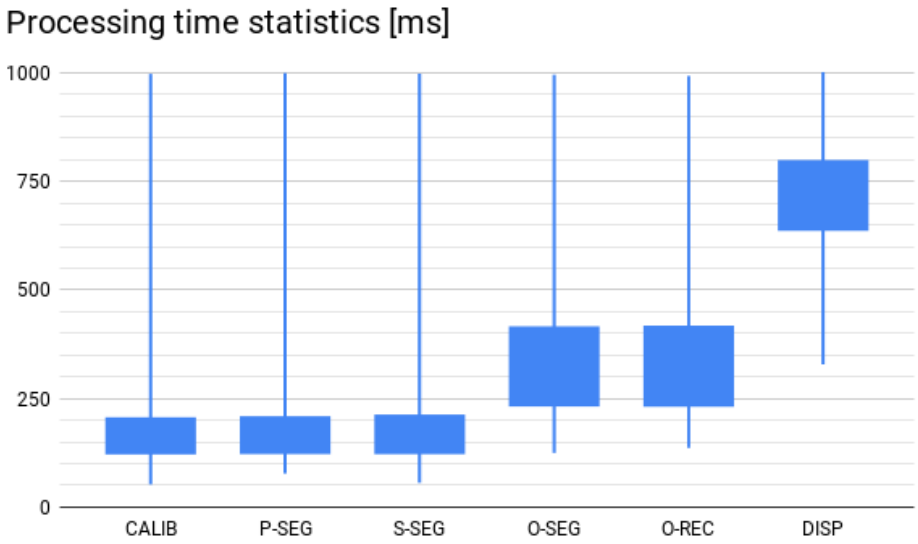


Figure 14.4: Object Recognition processing time.

of 1000 ms, which corresponds to the limit established by the project goals[4].

Finally, an analysis of the average processing time of the object recognition steps gives us the results shown in Table 14.2. The table shows how the average rate for the recognition process is 1,41 Hz, which corresponds to a time of 710 ms.

Step	**Avg. time [ms]**	**Avg. rate [Hz]**
CALIB	253	3,96
P-SEG	256	3,91
S-SEG	264	3,79
O-SEG	368	2,71
O-REC	368	2,72
DISP	710	1,41

Table 14.2: Object Recognition average processing time and rate.

[4] Note that the addition of all the step times is more than 1 second. However, due to the separation of the steps in nodes, they work as different processes and thus they do not run in sequence but in parallel.

14.3 Test Parameters

The test results shown above have been obtained by running the object recognition pipeline with the following parameters, seen in Table 14.3.

Parameter	Value
HSV Hist. - Hue bins	18
HSV Hist. - Saturation bins	3
HSV Hist. - Value bins	0
Normal Hist. - inclination bins	10
Normal Hist. - azimuth bins	10
Training Images per object	30

Table 14.3: Object Recognition test parameters.

Finally, a sample image of each object is shown in Figure 14.5. The image sizes range from 25 to 60 pixels both in width and height.

Figure 14.5: Sample object images.

15. Action Intention Recognition Results

As the capture of real action sequences is a time consuming task and the number of sequences needed to evaluate an algorithm has to be large, a mixed real/virtual approach

has been taken to test the proposed A2SN Action Intention Recognition Algorithm.

15.1 Testing database

From the performance of a small set of actions, a general rule has been extracted that represents an action sequence class. This general rule, named the *master sequence*, is a list of action steps. Each step contains information about the events that happen during the step, which can occur in any order; as well as information about a minimum and a maximum duration for each event. From these master sequence deduced from real action sequences, an infinite number of individual virtual sequences can be obtained which present a strong resemblance to real actions.

To test the A2SN algorithm, three actions have been taken into account, making sure that they contain similar events: coffee, cereals and cacao. All three actions involve the objects *mug*, and *milk*, and while the coffee action also involves the use of *coffee* and *sugar*, the cereals and cacao actions involve the use of *cereals* and *cacao* respectively.

The master sequences of each considered action have the following characteristics:

- **Coffee** - 4 items, 2 steps: the first step has 4 events and the second step has 3 events. 144 unique action sequences can be generated from this master sequence.
- **Cereals** - 3 items, 2 steps: the first step has 3 events and the second step has 4 events. 144 unique action sequences can be generated from this master sequence.
- **Cacao** - 3 items, 2 steps: the first step has 3 events and the second step has 4 events. 144 unique action sequences can be generated from this master sequence.

15.2 Test process

The process for testing the algorithm involves two phases:

1. **Graph Building [Learning Phase]** - for each action, a small number of individual sequences is generated and merged into a unique A2SN graph which represents the action

2. **Action Intention Recognition** [**Testin Phase**] - a random action is selected and an individual sequence of that action is generated. The generated sequence is passed to the action recognition, which uses the A2SN graphs from the Learning Phase for recognition. This is repeated 1000 times.

These tests have been performed by building the A2SN graphs with 5 sequence instances per action: only 3.5 % of the number of variations of each master sequence is used for training. Then these graphs have been used for recognizing 1000 different action sequences, and statistical metrics have been computed.

15.3 A2SN recognition results

Under the conditions presented above, the test results have given the statistical data seen in Table 15.1. It can be seen how the algorithm performs as expected with an 83.6% of correctly recognised actions in an average Early Detection Time of 18.8%.

Undetected Action Sequences	0.2 %
True Positives	83.6 %
False Positives	16.2 %
Average EDT	18.8 %
Average COD	6.85

Table 15.1: Statistical Action Recognition Results.

A sample of the Maximum Node Value evolution for the three considered A2SN graphs during the recognition of a *cereals* action sequence is shown in Figure 15.1, as well as a *coffee* action sequence in Figure 15.2. The EDT of this recognitions are between 20-30 % and the recognition outcome is correct. All the tests used the parameters $\alpha = 5, \beta = 10$.

15.4 A2SN performance results

Finally, the average computation time of the update function for all A2SN graphs has been tested by gradually increasing the number of A2SN graphs that are used for recognition. The number of graphs tested ranges from 3 to over 300, with a total number of nodes ranging from 30 nodes to over 3500 nodes. The results are shown as the average time

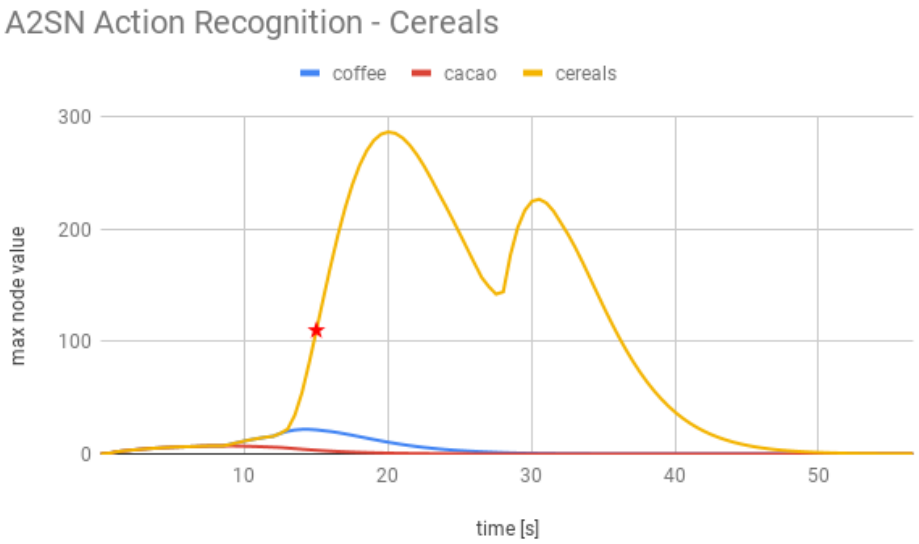


Figure 15.1: Evolution of the maximum node values for three A2SN graphs during the recognition of a *cereals* action.

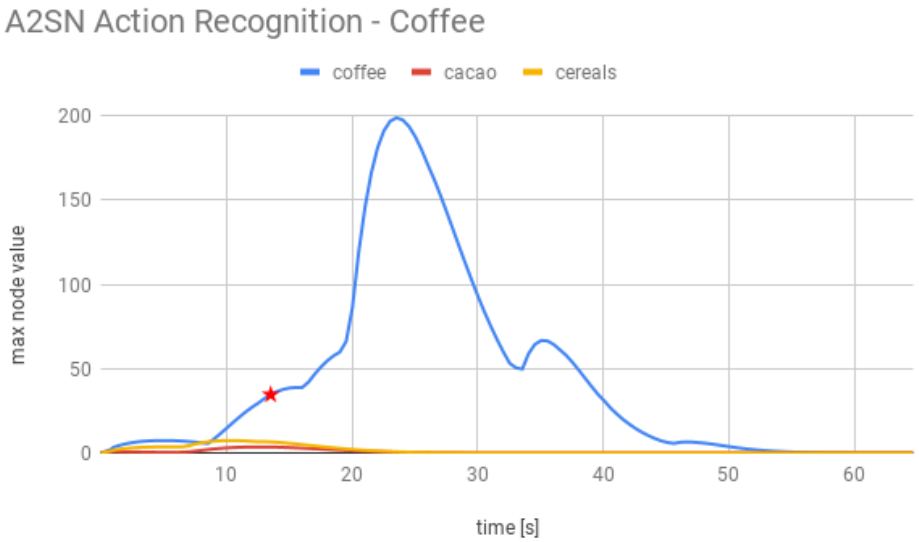


Figure 15.2: Evolution of the maximum node values for three A2SN graphs during the recognition of a *coffee* action.

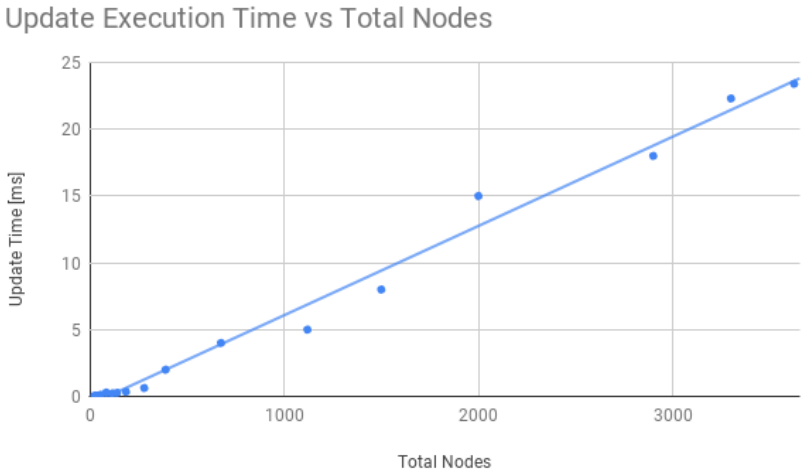


Figure 15.3: Execution time of the update function for all graphs relative to the total number of nodes in the program.

taken to update the graph values for all graphs as a function of the total number of nodes in the program, shown in Figure 15.3.

Given that the real time requirement for the program is to run at least at a frequency of 1 Hz, the developed A2SN implementation achieves this objective with a wide margin. This result also strengthens the scalability properties of A2SN action recognition algorithms.

Part VII

Project Schedule and Budget

After finishing the project, revising the initial schedule and computing an estimate of the project budget is a task that provides relevant feedback on the project outcome. It gives an insight on the overall performance during the project's life and on the economic implications of the work done.

This chapter analyzes the schedule that was programmed at the beginning of the project in order to compare it with the real project development flow and also estimates all the project costs (personnel, hardware amortization and energy consumption) in order to present a final project budget.

16. Schedule

An extended Master's Thesis at the UPC ETSEIB engineering school is comprised of 30 ECTS credits which translates to 750 hours of work. The schedule of this thesis covers eight months from November 2018 to June 2019: by considering 4 weeks per month and 5 working days per week, the work load is divided into 23,5 hours/week which is equivalent to 4 hours and 40 minutes/day. This was defined as the original work plan, and a distribution of time between project tasks was programmed as a Gantt chart seen in Figure 16.1.

After completing the project, a backwards analysis has been performed in order to confront the ideal work plan (750 total hours, 23,5 hours per week) with the real hours worked. In the end, a total of 828 hours have been devoted to the project, 85% of which are spent on researching, developing and testing the solution and the rest 15% are hours spent writing, revising and correcting the project's thesis. All tasks were programmed at the beginning of the project in blocks of 6 hours with each block representing a working day. Then the amount of real hours dedicated to each task has been used to compare the initial plan with the real project development. The differences between both plans are shown in Figure 16.2.

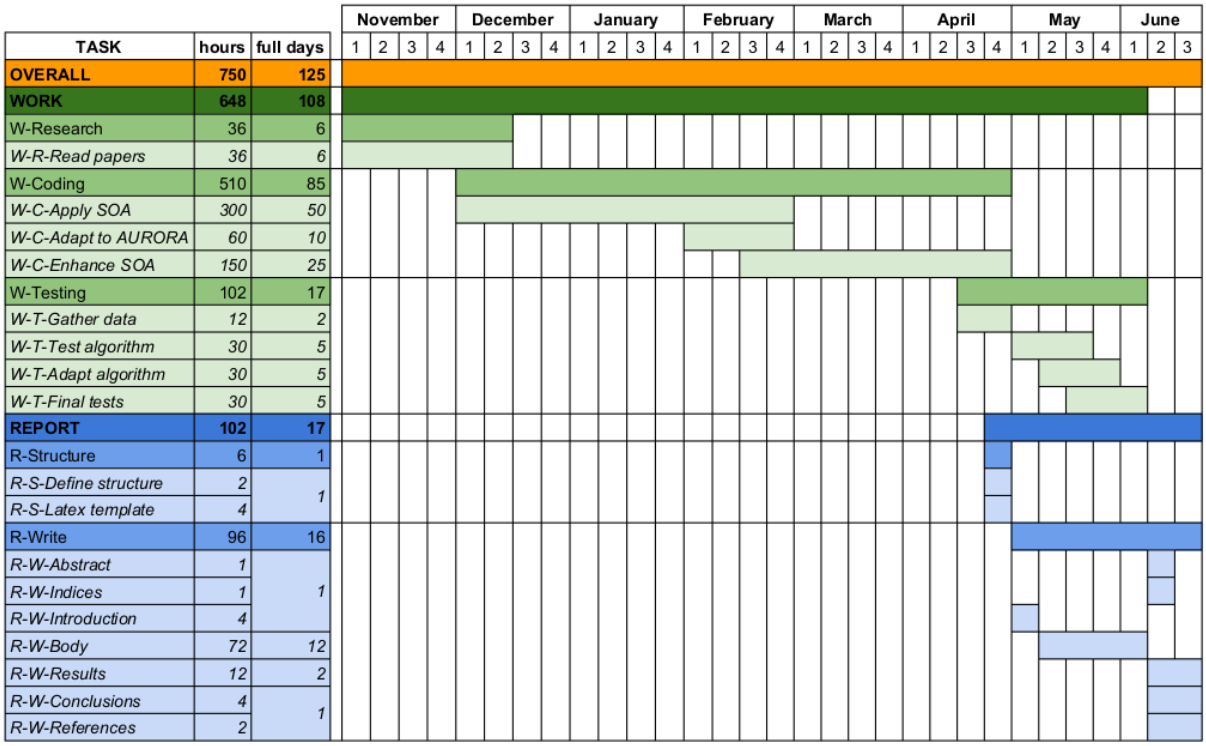


Figure 16.1: Initial project Gantt chart.

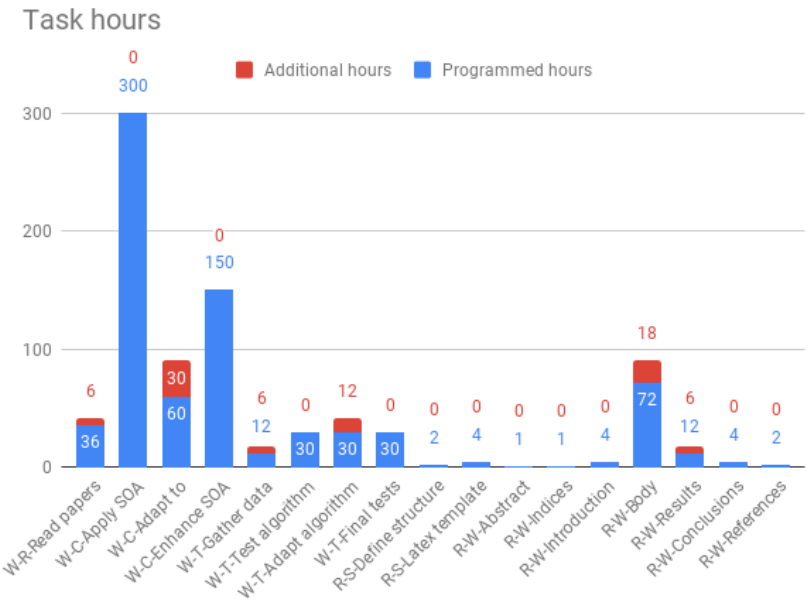


Figure 16.2: Difference between programmed and real hours for each task.

Finally, a comparison between the theoretical work hours per month (750/8) and the real dedicated hours can serve as an insight to learn from mistakes in order to perform better on the next project. Figure 16.3 shows the ideal work hours as well as the real hours per month.

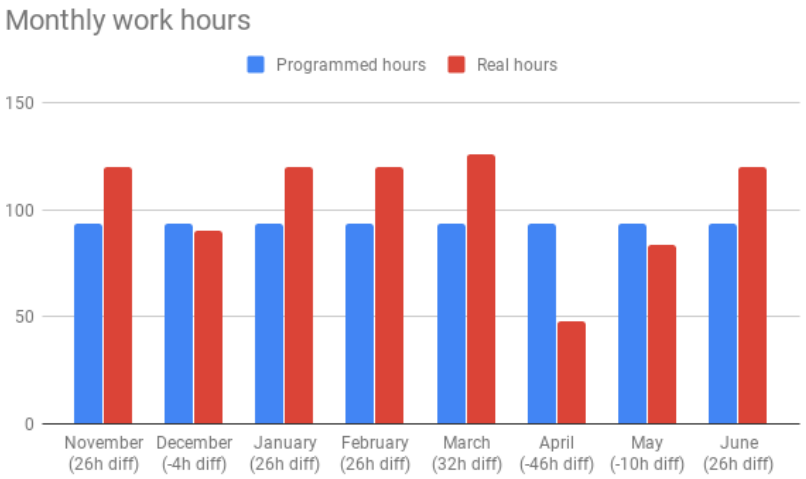


Figure 16.3: Difference between programmed and real hours for each month.

17. Budget

Taking into account the work hours and the material used for the project, a budget estimation can be performed in order to evaluate the costs related with it. The project budget is divided into personnel costs, amortization and energy consumption. As a reference for personnel costs, the salary of a Technician Engineer with a Master's Degree university level has been looked up in the latest approved Spanish BOE[5]. It consists in 23600 annual gross euros for a maximum of 1800 yearly work hours. This leads to an average wage of 13.1 €/h.

In order to add the personnel costs for the project director, the double of a technician wage has been considered: 26.2 €/h, assuming that the amount of hours the director

[5] Resolución de 30 de diciembre de 2016, de la Dirección General de Empleo, por la que se registra y publica el Convenio colectivo del sector de empresas de ingeniería y oficinas de estudios técnicos. *Boletín Oficial del Estado, 542*, de 18 de enero de 2017, 4356 a 4382.

devotes to the project is 10% of the hours dedicated by the technician. Thus in this project the personnel costs are computed as follows, where PC refers to Personnel Cost:

$$PC = 828h \cdot 13.1€/h + 82.8h \cdot 26.2€/h = 13016€ \tag{5}$$

As for the amortization costs, several hardware components have to be taken into account. No software amortization is considered in this project as all the software used is either free or open source. The hardware used in the project consists of:

- A personal computer with two screens, keyboard and mouse
- Three Kinect One cameras
- A server computer with an NVIDIA Titan X graphics card
- A Plasma television screen connected to the server

None of the robots is considered because even though they are part of the AURORA system they have no been used for any task related with the development of this project. After estimating the value of each hardware component and its useful life, the amortization can be computed as the time fraction of that life used for the project (8 months, 240 days). The final cost is then the total purchase cost multiplied by the amortization of each item, , showing the results in Table 17.1. The total amortization cost is 290.6 € .

Item	Cost	Useful Life	Amort.	Final cost
Personal Computer	400 €	1825 days	13.15%	52.6 €
Kinect One (x3)	420 €	3650 days	6.57%	27.6 €
Server	800 €	1825 days	13.15%	105.2 €
NVIDIA Titan X	1000 €	3650 days	6.57%	65.7 €
Plasma TV	600 €	3650 days	6.57%	39.5 €

Table 17.1: Amortization cost of project hardware.

Finally, the energy consumption of the project covers two main energy groups. The first one is the electric power consumed by the hardware equipment, and the second one is the cost of light and heating/cooling the laboratory. These costs have been estimated using average values multiplied by the work hours of the project. The computation of energy costs is shown in Table 17.2, adding up to 549.7 € .

The final project budget is obtained by adding all the previously considered costs. The result is shown in Table 17.3, with a total budget of 13856.3 € .

Concept	**€ / kWh**	**Project kWh**	**Final cost**
Light	0,1127930	270	30.4 €
Heat & A/C	0,1127930	3354	378.4 €
Hardware power	0,1127930	1249	140.9 €

Table 17.2: Energy cost of the project.

Concept	**Cost**
Personnel	13016.0 €
Hardware amortization	290.6€
Energy consumption	549.7 €
TOTAL	**13856.3 €**

Table 17.3: Project Budget.

Part VIII

Environmental Impact

A two part environmental analysis has been performed for the current project. On one hand, the direct impact that the hardware has on the environment has been studied from the point of view of potential hazards and CO2 emmision. On the other hand, the future impact that applications based on this project can have on the environment has also been taken into account.

The main hardware used during the project development has been three Kinect One cameras, two computers and a Plasma television. The Kinect cameras are designed for human use, and although they work with IR emitters, the harm they can cause is identical to the one of a traditional IR LED; and this light emission does not harm the environment. The Plasma television has been proved to be less harmful than the nowadays mostly used LCD displays, and if state of the art recycling techniques are used most of its materials can be used for further manufacture processes. As for the personal computers, during their usage life they only emit heat but they must be also recycled properly to avoid the waste of materials.

The second environmental aspect of this project is how it can affect future applications based on it. Regarding the kitchen environment, an intelligent assistive system can perform recycling tasks which are sometimes not done by people. For example, a robotic system can take care of trash handling and thus ensure that all the kitchenware and food is recycled in a proper way. this example shows that the project can also have beneficial effects for the environment if applied properly in future applications.

Part IX

Conclusions

The field of Assistive Robotics has an enormous growth potential as it tackles a wide amount of daily problems which affect at least ten percent of the world population [28]. The broadness of its object of study is also the cause of the challenges for finding robust, scalable and viable solutions to these problems. Despite this obstacle among many others the research community has held a steady pace of improvement in this area over the last 20 years. One of the specific issues studied in the context of Assistive Robotics is Action Intention Recognition.

Action Intention Recognition has several particularities which make it a challenging problem to solve. Action Classification approaches need to deal with high inter and intra class variability as well as with spatio-temporal differences between similar actions. Moreover, the objective to detect human intentions adds a real time requirement and a predictive aspect to the problem to solve. This project proposes a novel Action Intention Recognition based on RGBD Object Recognition, parting from state of the art approaches and making slight modifications to improve such methods.

The object recognition algorithm has proven its capabilities to run in real time with the input from three cameras while recognizing and positioning the objects present in the region if interest. In spite of the proposed volume descriptor providing desirable results on the object dataset, the performance of color classification with HSV histograms has not achieved enough robustness. The low image resolution and the real time requirement make the task of finding an adequate classification method a difficult and time consuming task.

When it comes to action recognition, the proposed improvements over the ASN formalism have proven to be effective for autonomous addition of new action sequences to the database. Moreover, the applied changes provide a suitable structure that can be subject of traditional Neural Network training methods, although this is a field that has been left for future studies. Finally, the tests preformed under similar conditions to the ones carried out by the ASN authors and with a higher degree of similarity between actions show that the performance cost of introducing the proposed changes is minimal.

After analyzing the development of the project and the results of the tests performed, it can be concluded that all of the project have been fulfilled: an object recognition algorithm that recognises and positions objects from a kitchen set in real time and an

action intention recognition algorithm that differentiates between a set of kitchen tasks and recognizes them while they are being performed have been developed, implemented and tested. However, there is still room for improvement which has been gathered as future work proposals.

18. Future Work

Several aspects of the project development and implementation have opened new ways of improving the proposed solution. The most relevant ones have been gathered in the following proposals for future work on the subject:

- Improve the detection of object movement by adding a lightweight object tracking algorithm to the object recognition process. In order to avoid confusions by volume when objects are overlapped, a color based tracking should be used.
- Explore and test the capabilities of A2SN action intention recognition in the case of consecutive actions and changes mid-action. The proposed method of using A2SN batches and monitorizing their *utility* can be a starting point for this analysis.
- Build a large enough real action sequences database to validate the experimental results of this project without the use of virtual data.
- Improve the color object recognition by refining the descriptors used. Adding texture information and segmenting the object images in color clusters before computing the histograms are two widespread approaches to improve the classification results.

References

[1] A Alexandre. A Comparative Evaluation of 3D Keypoint Detectors in a RGB-D Object Dataset. *2014 International Conference on Computer Vision Theory and Applications (VISAPP)*, pages 476–483, 2014.

[2] Joan Aranda and Manuel Vinagre. Anticipating human activities from object interaction cues. In *2016 25th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 58–63. IEEE, aug 2016.

[3] Manuel Blum, Jost Tobias Springenberg, Jan Wülfing, and Martin Riedmiller. A learned feature descriptor for object recognition in rgb-d data. In *2012 IEEE International Conference on Robotics and Automation*, pages 1298–1303. IEEE, 2012.

[4] Liefeng Bo, Xiaofeng Ren, and Dieter Fox. Unsupervised feature learning for rgb-d based object recognition. In *Experimental Robotics*, pages 387–402. Springer, 2013.

[5] Dr. Willie Brink. *Computer Vision Course Notes*, chapter 5. Stellenbosch University, Western Cape, South Africa, 2017.

[6] Olivier Chapelle, Patrick Haffner, and Vladimir N. Vapnik. Support vector machines for histogram-based image classification. *IEEE Transactions on Neural Networks*, 1999.

[7] Saurabh Gupta, Ross Girshick, Pablo Arbeláez, and Jitendra Malik. Learning rich features from RGB-D images for object detection and segmentation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8695 LNCS(PART 7):345–360, 2014.

[8] [Image]. Docker logo - Docker official website. `https://www.docker.com/sites/default/files/social/docker_facebook_share.png`. Accessed: 2019-05-30.

[9] Image. HSV color space cone. `https://i.stack.imgur.com/uSbIT.png`. Accessed: 2019-06-16.

[10] [Image]. IAI-Kinect logo - IAI Github repository. `https://avatars1.githubusercontent.com/u/4136974?s=400&v=4`. Accessed: 2019-05-30.

[11] [Image]. MoveIt! logo - MoveIt Github repository. `https://github.com/ros-planning/moveit`. Accessed: 2019-05-30.

[12] [Image]. OpenCL logo - AnandTech. https://images.anandtech.com/doci/9039/OpenCLLogo_678x452.png. Accessed: 2019-05-30.

[13] [Image]. OpenCV logo - Wikimedia. https://upload.wikimedia.org/wikipedia/commons/thumb/3/32/OpenCV_Logo_with_text_svg_version.svg/1200px-OpenCV_Logo_with_text_svg_version.svg.png. Accessed: 2019-05-30.

[14] [Image]. OpenKinect logo - OpenKinect Github repository. https://avatars0.githubusercontent.com/u/478332?s=400&v=4. Accessed: 2019-05-30.

[15] [Image]. Pinhole camera model - OpenMVG official documentation. https://openmvg.readthedocs.io/en/latest/openMVG/cameras/cameras/. Accessed: 2019-04-17.

[16] [Image]. Pointcloudlibrary logo - PCL official website. http://www.pointclouds.org/assets/images/contents/logos/pcl/pointcloudlibrary_vert_large_pos.png. Accessed: 2019-05-30.

[17] [Image]. ROS logo - Generation Robots. https://www.generationrobots.com/blog/wp-content/uploads/2016/03/Logo-ROS-Robot-Operating-System1-687x241.jpg. Accessed: 2019-05-30.

[18] [Image]. RViz logo - RViz Github repository. https://raw.githubusercontent.com/ros-visualization/rviz/melodic-devel/images/splash.png. Accessed: 2019-05-30.

[19] International Federation of Robotics. Global industrial robot sales doubled over the past five years. *World Robotics - Industrial Robot Report 2018*, October 2018.

[20] Richard Kelley, Alireza Tavakkoli, Christopher King, Amol Ambardekar, Monica Nicolescu, and Mircea Nicolescu. Context-based Bayesian intent recognition. *IEEE Transactions on Autonomous Mental Development*, 4(3):215–225, 2012.

[21] Kevin Lai, Liefeng Bo, Xiaofeng Ren, and Dieter Fox. A large-scale hierarchical multi-view rgb-d object dataset. In *2011 IEEE international conference on robotics and automation*, pages 1817–1824. IEEE, 2011.

[22] Pier Luigi Mazzeo, Luciano Giove, Giuseppe M. Moramarco, Paolo Spagnolo, and Marco Leo. HSV and RGB color histograms comparing for objects tracking among non overlapping FOVs, using CBTF. In *2011 8th IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS 2011*, 2011.

[23] Yannick Morvan. *Acquisition, Compression and Rendering of Depth and Texture for Multi-View Video*. PhD thesis, Eindhoven University of Technology, January 2008.

[24] Roland Smeenk. Kinect FOV explorer tool. `http://www.smeenk.com/webgl/kinectfovexplorer.html`. Accessed: 2019-03-20.

[25] Radu Bogdan Rusu, Gary Bradski, Romain Thibaux, and John Hsu. Fast 3D recognition and pose using the viewpoint feature histogram. *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, pages 2155–2162, 2010.

[26] Mohammad Taghi Saffar, Mircea Nicolescu, Monica Nicolescu, and Banafsheh Rekabdar. Context-based intent understanding using an Activation Spreading architecture. *IEEE International Conference on Intelligent Robots and Systems*, 2015-December:3002–3009, 2015.

[27] Giovanni Saponaro, Giampiero Salvi, and Alexandre Bernardino. Robot anticipation of human intentions through continuous gesture recognition. *Proceedings of the 2013 International Conference on Collaboration Technologies and Systems, CTS 2013*, 1(Cts):218–225, 2013.

[28] Laura Sminkey. World Report on Disability. *World Health Organisation*, 2011.

[29] Shiva Soleimanizadeh, Dzulkifli Mohamad, Tanzila Saba, and Amjad Rehman. Recognition of Partially Occluded Objects Based on the Three Different Color Spaces (RGB, YCbCr, HSV). *3D Research*, 2015.

[30] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. DeepFace: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2014.