Design and Implementation of a Computer Vision System for Robotic Disassembly of Electric Vehicle Battery Pack.

By:

Eduard Marti Bigorra

Supervisors:

Martin Marie Hubert Choux

Ilya Tyapin

This Master's Thesis is carried out as a part of the education at the University of Agder and is therefore approved as a part of this education. However, this does not imply that the University answers for the methods that are used or the conclusions that are drawn.

University of Agder, Spring 2020
Faculty of Engineering and Science
Department of Mechatronics

Preface

This report is written as part of the Master's program in Mechatronics at the University of Agder. Being part of this project has been an excellent opportunity for me to grow professionally and personally.

I want to thank my supervisors, associate professor Martin Marie Hubert Choux and associate professor Ilya Tyapin at the University of Agder for all their support and opportunities since I contacted them for the first time. I appreciate their implication and the trust placed in me during all the project.

I also wish to thank PhD candidates Atle Aalerud and Dipendra Subedi for their assistance and useful recommendations.

Lastly, I would like to make a special mention to my dad, my mum and especially my sister. Thank you for all your unconditional support, without you this would not have been possible. Gràcies de tot cor.

Grimstad, June 23, 2020

Abstract

The increasing demands for greener transport solutions results in a foreseeable increase of Electric Vehicles (EV). The development of EVs puts strong demands on the development of Lithium-Ion Batteries but also into its dismantling process, which today is made manually and therefore is time-consuming. LIBRES, Lithium-Ion Battery Recycling, is a project owned by Norsk Hydro with the goal of finding fully-automated solutions for complete LIB batteries dismantling. The work presented in this thesis is a part of the LIBRES project and its aim is to develop a task planner for EV LIB battery pack dismantling to a module level through the design and implementation of a computer vision system. The designed task planner integrates the communication between the primary system elements (i.e. the robot, the 3D camera and the computer). The aim of the task planner is (1) to scan the dismantling scene, (2) to identify the different LIB components and their locations in the battery, (3) to create a dismantling order and lastly (4) to move the robot to the detected dismantling positions. The object detection algorithm "You Only Look Once" combined with the projects pose estimation part is used to find the different components in the disassembly scene and determine their position. To implement the designed task planner, a Volkswagen Hybrid LIB pack is considered. Results show that the method has a good accuracy with errors lower than 5 mm. In addition, the system is able to perform steps (1), (2), and (3) mentioned above, in an average time of 40 seconds.

Contents

Preface i

Abstract ii

1 Introduction 1

1.1 Industrial Motivation 1

1.2 Project objectives and scope 2

1.3 Project limitations 2

1.4 Report Outline 2

2 Theory on Disassembly 4

2.1 Challenges in disassembly 4

2.2 Disassembly Automation 6

2.2.1 Mechanical design review 7

2.2.2 Perception systems in automated disassembly 10

2.3 Product analysis 12

2.3.1 Main components of a LIB pack 13

2.3.2 Manual disassembly of a Volkswagen Hybrid LIB pack (25 Ah) 14

3 Robot Operating System communication 16

3.1 Robot Operating System (ROS) 16

3.1.1 ROS distributions 16

3.1.2 ROS background 17

3.1.3 ROS in Python (rospy) 18

3.2 Moveit! 19

3.2.1 MoveIt! Setup Assistant 21

3.2.2 ABB IRB4400 + track IRBT4004 MoveIt! package creation and configuration 22

3.2.3 Essential coding to move the robot . . . 25
3.3 ROS implementation . . . 26
3.3.1 Zivid 3D camera ROS driver . . . 27
3.3.2 ABB ROS-Server . . . 29
3.3.3 Catkin Workspace . . . 31

4 Guidance: UiA Robotics Lab ROS set up manual 32

5 Object detection 37
5.1 Image capture: Zivid one parameter adjustment . . . 37
5.2 Object detection: YOLOv3 . . . 44
5.3 YOLOv3 Algorithm training . . . 47
5.3.1 System's nomenclature . . . 47
5.3.2 Labeling . . . 47
5.3.3 Training Images acquisition . . . 47
5.3.4 Training stage . . . 48
5.3.5 YOLOv3 training results . . . 50
5.3.6 Training data creator (function) . . . 50

6 Pose estimation 52
6.1 Structural light camera: Zivid one 3D camera . . . 52
6.1.1 Pointcloud concept . . . 53
6.1.2 Structured Light Imaging . . . 53
6.2 Point cloud and depth images storage . . . 54
6.3 Intrinsic and Extrinsic Parameters . . . 56
6.3.1 Pinhole Camera Model . . . 57
6.3.2 Intrinsic Parameters . . . 58
6.3.3 Extrinsic Parameters Calibration . . . 59
6.4 World coordinates of the detected components . . . 62

7 Methodology, task planning 68
7.1 Task planning methodology analysis . . . 68
7.2 Fully automated dismantling system . . . 69
7.2.1 Agent emulating human behaviour . . . 69

7.2.2 Semi-destructive disassembly . . . 71
7.3 Task Planner Design . . . 71
7.3.1 Task Planner overview . . . 72
7.3.2 Integrated Functions and scripts . . . 75
7.3.3 Integrated Functions and scripts flow charts . . . 81
7.3.4 Running the task planner . . . 87
7.3.5 Conclusion . . . 88

8 Project results **89**
8.1 Object detection results . . . 89
8.2 Time analysis . . . 90
8.3 Decision-making: Optimal path . . . 92
8.4 Accuracy . . . 94

9 Conclusions, future work and discussion **97**

Bibliography **100**

A Hardware Specifications **103**
A.1 Zivid 3D Camera . . . 104
A.2 IRB4400 . . . 105
A.3 IRB4400 . . . 106

B Code Documentation **107**
B.1 [main()]: Task planner core . . . 107
B.2 [CamR_pos()]: Find the positions in camera reference base frame . . . 109
B.3 [WR_pos]: World reference base frame position detection: . . . 110
B.4 [remove()]: Removal Operation . . . 111
B.5 [num_components]: Components analysis . . . 112
B.6 [what_component and has_comp_over]: Decision-making . . . 113
B.7 [training_data_autogenerator()]: Automatic YOLO training data creator . . . 115
B.8 [merge_detection()]: Merge detection . . . 116
B.9 [move_to_pict.py]: Image capturing . . . 118
B.10 [move_pos.py]: Move the robot TCP to a specific position . . . 123
B.11 [move_to_cal.py]: Calibration images capturing . . . 124

B.12 Moving the robot basic code . . . 127
B.13 [training_data.py]: Training images capture . . . 128
B.14 [obtain_calfiles.py]: Obtain calibration files . . . 131

C Extrinsic calibration **133**
C.1 Extrinsic calibration (Zivid Command Line Interface tool) . . . 133
C.2 Final calibration transform: calibration_transform.yalm . . . 134
C.3 Calibration positions . . . 135
C.3.1 pos01.yaml . . . 135
C.3.2 pos02.yaml . . . 135
C.3.3 pos03.yaml . . . 135
C.3.4 pos04.yaml . . . 136
C.3.5 pos05.yaml . . . 136
C.3.6 pos06.yaml . . . 136
C.3.7 pos07.yaml . . . 137
C.3.8 pos08.yaml . . . 137
C.3.9 pos09.yaml . . . 137
C.3.10 pos10.yaml . . . 138
C.3.11 pos11.yaml . . . 138
C.3.12 pos12.yaml . . . 138

D Task planner feedback **140**

E Guidance Code **142**
E.1 moverobot.py . . . 142

Chapter 1

Introduction

Global warming is today one of the major concerns in the society being the greenhouse effect one of the major causes. The emission of too much carbon dioxide (CO2) by combustion engines to the air is said to cause accelerated global warming.

In fact, transport is responsible for nearly 30% of the European Union's total CO2 emissions and while greenhouse gas emissions decreased in the majority of sectors between 1990 and 2018, CO2 emissions increased by 172 Million tonnes (CO2 equivalents) in transport [2]. For that reason, updating and changing the traditional fuel-based powered vehicles for zero-emission vehicles (EV) is getting every day more common among personal car users. This change is already becoming a reality in Norway where in 2017 the 21% of the new passenger cars sold in the country were zero-emission cars (or 40% if rechargeable hybrid vehicles are included).

In 2018, the global electric car fleet exceeded 5.1 million, up by 2 million since 2017, almost doubling the unprecedented amount of new registrations in 2017. In Europe, 1.2 million electric cars were sold during this year. In terms of electric car market share, Norway remained the global leader at 46% of its new electric car sales in 2018, more than double the second-largest market share in Iceland at 17% and six-times higher than third-highest Sweden at 8% [15].

1.1 Industrial Motivation

Due to the increasing market share of the EV and the necessity of recycling the lithium-ion battery (LIB), the project LIBRES (Lithium-Ion Battery Recycling project) was launched in 2018. Different projects conform the LIBRES project, and this master project is one of them. LIBRES project main goal is to have, by 2022, a developed LIB recycling pilot plant in Norway large enough to handle commercial volumes in 2024. Norsk Hydro ASA is the project owner, and the University of Agder (UiA) is one of the research and development partners for the project. One of the leading associates linked with battery dismantling work package is BatteriRetur AS.

Today, the automotive LIBs dismantling process is mainly carried out manually and the use of robotics in this process is limited to simple tasks or human assistance [37]. Automated systems are more robust, have a lower-cost and are lower-time consuming compared to manual systems. Given the increasing number of new electric cars registrations the implementation of robotics and automation is the near future of this sector of the industry [38].

Thus, this master project is a part of the current research project LIBRES and the main goal of this thesis, is to develop a battery dismantling process chain using computer vision and the robot

to partially disassembly an electric vehicle LIB from pack to module level.

1.2 Project objectives and scope

This project aims to automate the dismantling process of Electric Vehicles Lithium Ion Batteries from battery pack to module level through the design of a computer vision system.

The task planner to be designed needs to coordinate the robot, camera and computer in order to capture images and point cloud data, analyse the images through object detection, find the world reference frame positions (pose estimation), decide the operations order and move the robot to the desired points. Thus, communication between the main system elements (i.e. robot,...) is considered as one of the main challenges.

In order to accomplish the above-mentioned objective, the scope of the work presented in this thesis is to:

- Study the disassembly process and its automation.
- Setup communication between the system elements using ROS-Robot Operating System.
- Create robot and linear track MoveIt! package.
- Define task planner methodology.
- Draft a ROS setup manual for UiA Robotics Lab.
- Detect object (Using 2D Images).
- Estimate pose (Using 3D point clouds).
- Integrate all the parts in the task planner design.

1.3 Project limitations

The objective of the project is not to a complete and functioning dismantling system prototype and some of the parts, i.e. the tools should be done in future stages of the LIBRES project.

The task planner will be verified using only one Volkswagen LIB pack. However, other battery types should also be tested in order to ensure the robustness of the proposed task planner.

1.4 Report Outline

The rest of this report is organized as follows. Chapter 2 (Theory on disassembly) contains theory on disassembly automation and product analysis. Next, chapter 3 (ROS communication) includes all the essential Robot Operating System configurations and concepts relevant for the project. MoveIt! package creation is also presented in chapter 3. Then, it is important to highlight that chapter 5 (Object Detection) and chapter 6 (Pose Estimation) are highly interconnected. The final goal of object detection + pose estimation is to obtain the positions of the components in world reference base frame (see Figure 1.1). To do so, chapter 5 aims to explain how the algorithm detects the components in the scanning images of the battery pack. Then, chapter 6 shows how

the information extracted from chapter 5 is used to find the positions of the components in the world reference base frame, solving the pose estimation problem.

In chapter 7 the task planner is explained. The task planner is the main core of the proposed dismantling system and is the responsible to integrate all the knowledge and the functions presented in chapters 2, 3, 5 and 6 (see Figure 1.1). Lastly, Chapter 9 presents conclusions and future work.

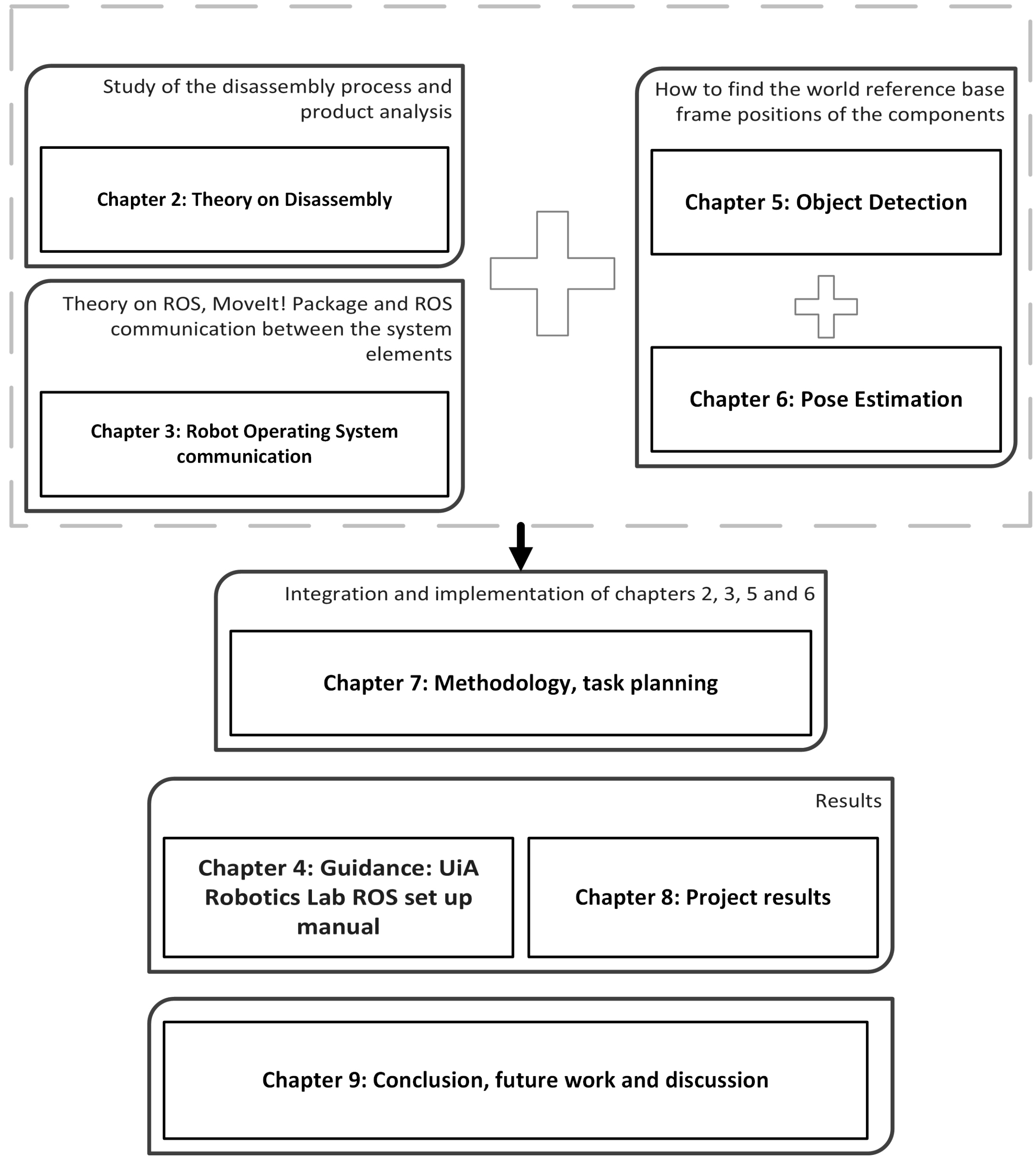


Figure 1.1: report outline

Chapter 2

Theory on Disassembly

In the life cycle of a product, End Of Life (EOL) treatment is one of the most important stages to achieve high sustainability. Since disassembly is a key operation to achieve a successful EOL treatment, its automation using robotics has received particular attention in the research communities, the last few years, together with the increased popularity of circular economy. The aim of this chapter is hence to present theory on the robotic dismantling processes and in disassembly automation.

Designing products according to the Design for Disassembly (DfD) guidelines is the final solution to accomplish the economical feasibility for the disassembly of products. In the particular case of the EV LIBs, the standardization of the models is beginning to happen and it is going to become a reality in the next years.

This chapter presents the main challenges and uncertainties in disassembly, explains disassembly automation and analyses the product to be dismantled (LIBs).

2.1 Challenges in disassembly

Generally, the disassembly process of EoL products cannot be considered as the reverse of its assembly, mainly because of the presence of several uncertainties.

These uncertainties are mainly caused by uncertainties within models, model-related variations and operational difficulties.

Uncertainties within models

- **Component defects:**

  Component defects can cause problems during the disassembly process. This defects are mainly caused by the use of the product during its lifespan. For example: damaged batteries, broken screws or fasteners, etc..

- **Upgrading or downgrading during usage:**

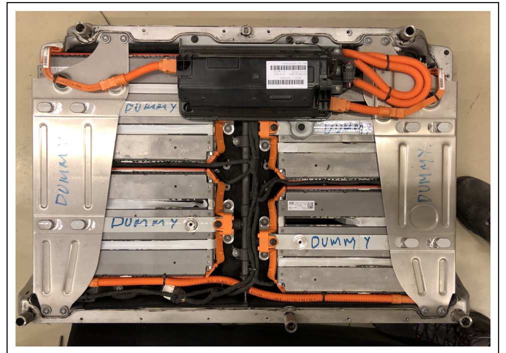
  During maintenance operations a change in the configuration of the product can happen, specially in products with exchangeable components.

- **Damage during the disassembly operation:**

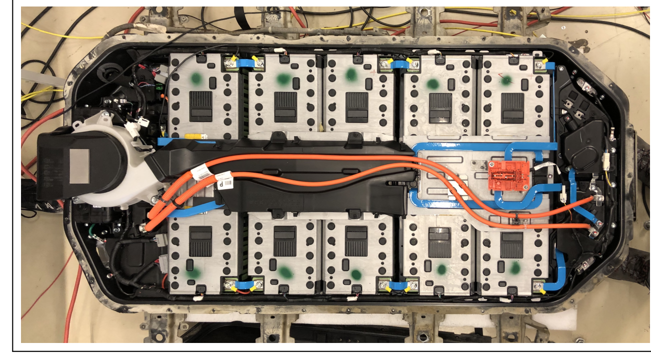
Disassembly consists of several operations realised in a time sequence. Prior disassembly operations can damage some components to be disassembled at later stages, specially in fragile products. In the specific case of LIB, it should not be a major problem.

Model-related variations

In the same product family (for this project LIBs), depending on the producer, the year/month and the car model, different versions can be found. Within these models there can be different characteristics, components, materials, fasteners or internal configurations. Thus, incomplete information is one of the main challenges to deal with during the design of a dismantling system. These differences are clearly visible between the Volkswagen and the Mitsubishi battery packs shown in the Figure 2.1.



(a) Volkswagen battery pack



(b) Mitsubishi battery pack

Figure 2.1: Volkswagen and Mitsubishi battery packs

Operation difficulties

Many papers are dedicated to qualify the difficulty of the disassembly process of a product. The main criteria are ideas are summarised below:

- In [16], the term disassemblability is quantified by assessing in five major criteria: (a) component accessibility, (b) precision in locating the component, (c) force required to perform tasks, (d) additional time, and (e) special problems.
- Similarly in [18], the disassemblability is quantified by summarising the following characteristics: (a) Minimal force exertion, (b) quick operation without excessive manual labour, (c) simple mechanism of disassembly, (d) minimal use of tool, (e) minimal part repetition, (f) easy recognition of fasteners, (g) simple product structure and (h) avoidance in usage of toxic material.
- Finally the authors in [13] state that the development of optimal disassembly plans relies on four key phases: (a) product analysis, (b) assembly analysis, (c) usage mode and effect analysis and (d) dismantling strategy.

The disassemblability of industrial batteries, as described above, can be improved either by modifying their design, or by developing new technologies to ease and eventually remove some of the above-mentioned challenges, as for example, making the recognition of fasteners an easy task. This second route, i.e. making the disassembly process smarter and more efficient is the one chosen in this study.

2.2 Disassembly Automation

In great measure the economic feasibility of the disassembly process goes through an automated disassembly. Nowadays, the disassembly process is done manually, causing high operating costs, specially in developed countries. The main restriction for the success of the automated systems in dismantling are the variations and uncertainties in used products.

To be automated, a dismantling system must include sensors and an artificial intelligence (AI) agent controlling the mechanical operation units. The sensors give the flexibility to perceive important information about the precise state of the treated product at each moment of the process.

When talking about the degree of autonomy in the dismantling process, this can go from manual disassembly to semi-automatic disassembly and fully-automatic disassembly [21].

The design of this kind of system involves different engineering fields like: computer vision, robotics, mechanical systems, control systems, disassembly process planning and product analysis.

Why automation?

Automation has proved to be more cost-effective, more accurate, and faster to handle repetitive works than human workers. Moreover, robots are able to work on hazardous tasks and in undesirable (for humans) environments. For these reasons, automation plays an important role in the modern manufacturing industry. But, some similar issues are faced in the disassembly process. The implementation of manual disassembly is difficult to justify (in an economical point of view) due to the generally-low economic returns from EOL treatment. That is because the process is usually laborious, it has high costs and it is affected by the variations and uncertainties of the used products. " The process may also be hazardous depending on the subject of disassembly. For example, the disassembly of electric vehicle batteries deals with risks due to the chemical composition and any charge remaining in the cells. Automation has a high potential to reduce these problems in the disassembly process." [29]

The design of an automated dismantling system, sets technical and economical challenges such as the variety of manufacturers and different product types, variety in design and product structure, small lot sizes and insufficient collection logistic, varying product condition after the usage phase, changes in standard components, insufficient disassembly tools, changes in legislation and changes in market demand and prices. So all these points should be taken into account in the design of an automated dismantling system.

Given its high flexibility (due to the perception, the dexterity and the intelligence) of humans in front of the variations in returned products, manual labour has traditionally been used rather than automated systems.

- **Perception:** refers to the sensor systems ability to perceive information about the process.
- **Dexterity:** relates to the proficiency of carrying out the physical operation.
- **Intelligence:** relates to the ability to plan and control the process according the knowledge of the product.

In practice, to limit the variations encountered in the process, each system is designed for disassembling a particular product type or family. Therefore, the technical requirements of the dismantling tools, relating to e.g. the type of techniques, product size and weight, can be limited. A specifically-designed system can satisfy the cost constraint given a sufficiently-large lot size. However, the lot

size of returned EOL products is unpredictable and may lead to difficulty at the strategic planning level. In addition, the profit depends on the market value of the recycled and reused parts, which changes according to demand and legislation. Therefore, strategic planning regarding these aspects should be done before developing the automated system.

2.2.1 Mechanical design review

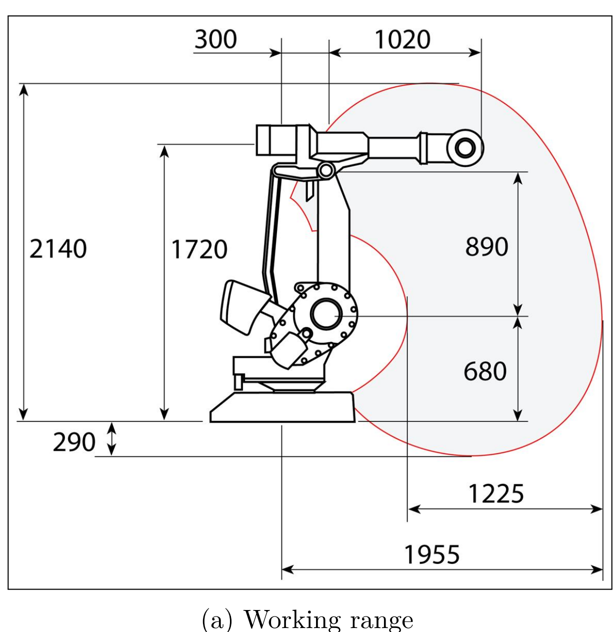
In the disassembly process it is necessary to interact physically with the components and the products to be disassembled. Mechanical design refers here to the development of the tools required to realize the disassembly operations. Tools can be categorized in three types: manipulators, disassembly tools and handling devices.

Manipulators

Manipulators are the robot or other devices that perform the movement in the disassembly process. Using sensors, disassembly tools, and other devices, they carry out the dismantling. A disassembly system can consist of one or various manipulators working cooperatively.

Robot arms are usually the selected manipulators given their versatility. A low-level control (force-torque control, path planning and motion control) is generally included in industrial robots. The selection of the specific manipulator depends basically on the task. The robot needs to have sufficient degrees of freedom (DOF) to perform the operations.

In this project, the **ABB IRB 4400** robot has been assigned as the manipulator (see Figure 2.2b). "IRB 4400 is an extremely fast, compact robot for medium to heavy handling. It has exceptional all-round capabilities which makes it suitable for a variety of manufacturing applications. The load capacity of 60 kg at very high speeds usually permits handling of two parts at a time." [3]



(a) Working range

(b) ABB IRB 4400

Figure 2.2: ABB IRB 4400

Some of the most important specifications of this manipulator are: (see the all the specifications

datasheet in appendix A)

SPECIFICATIONS	
Reach	1.95 m
Payload	60 kg
Number of axis	6
Controller	IRC5 Single Cabinet
Position repeatability RP	0.06 mm
Path repeatability RT (at 1.6 m/s)	0.09 mm

Disassembly tools: connective components

- **Screw automatic removal:**

  As it is known, screws are one of the most important categories of fasteners and they can be found in almost all the products, specially in LIB. The removal of the screws can be done by three different means: non-destructive, semi-destructive and destructive [5].

  In the **non-destructive removal**, a screwdriver attached to an electric motor equipped with a potenciometer is used for measuring and monitor continuously the torque.

  In the **semi-destructive removal**, using a pneumatic tool it is created a new slot in the head of the screw (see figure 2.3) . This allows to remove different types of screw and it also works for damaged screws. Tools developed by Seliger [30] and Feldmann [10].

  In the **destructive** removal, the connection is disestablished intentionally damaging the head of the screw by using various methods of material removal (grinding, drilling, chisel, milling, etc..)

  In addition to that, in a project for BatteriRetur, a flexible screwdriver was designed in the University of Agder. As can be seen in Figure 3, the tool is capable of fitting in different types of screws of various sizes.

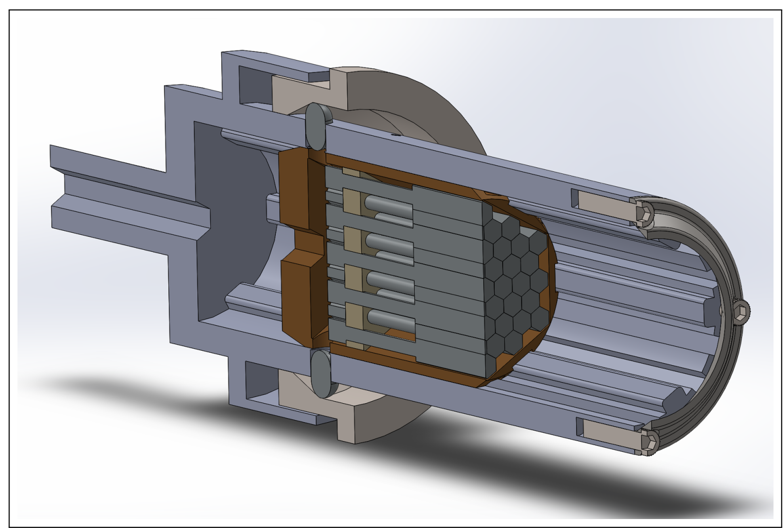
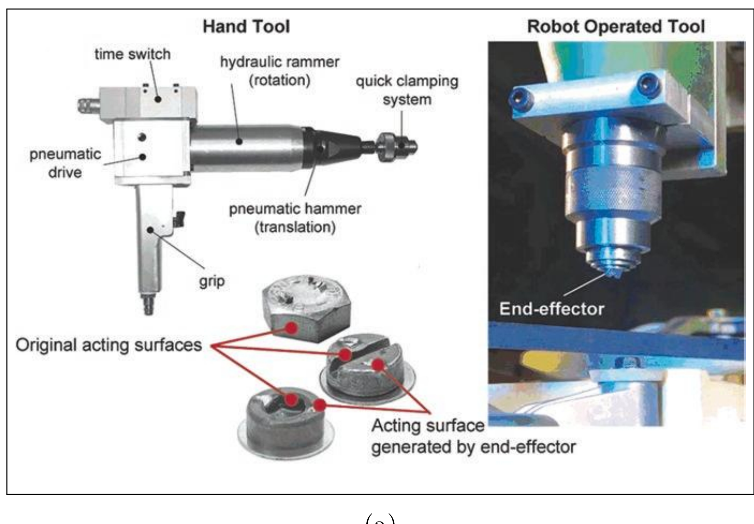


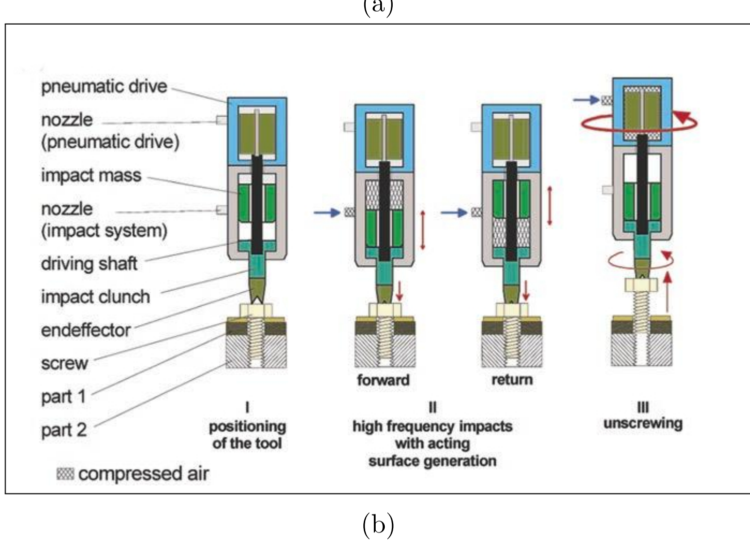
Figure 2.4: UiA screwdriver concept tool

- **Snap-fits automatic removal:**

  Snap-fits are used as fasteners according to the principle of design for assembly (DFA). It is an assembly method used to attach flexible parts, usually plastic. The components are interlocking together thanks to the hook-like elements.



(a)



(b)

Figure 2.3: Screws semi-destructive removal

There are some studies in **Snap-fits non-destructive automated disassembly** that demonstrate that is only possible in certain conditions [17].

Disassembly tools: Handling devices

Handling devices refer to fixtures, logistics systems and grippers used to control the motion of the parts and products through the steps during the dismanling process.

- **Product fixtures:**

  The aim of the fixtures (Figure 2.5) is to ensure the location and orientation of the product in a specific place. This is essential to have highly accurate physical operations. In certain operations, the reaction forces would cause undesired displacement of the object to be dismantled, if not properly fixed.

  The design of the fixture should be able to fix different product models and it should not obstruct the detection and the removal of the different product components or connective components.

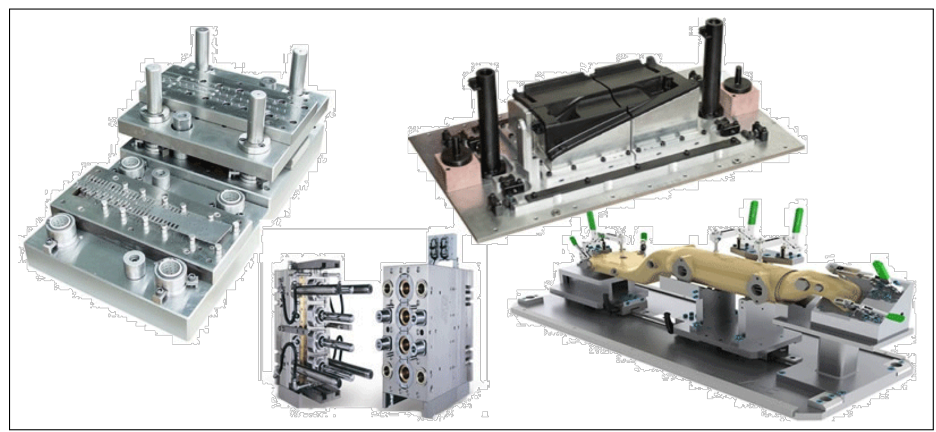


Figure 2.5: Product fixtures

- **Grippers** The main objective of the grippers is to help the manipulator grasping and handling the desired components and objects. Grippers can be classified between their use: to disestablish connections and to remove the components that have been already disconnected

  From one side, for removing the disconnected components, the gripper type depends especially on the geometry of the component. Visual input, together with force feedback, may be used to determine the size and shape of the object. From the other side, the "pull" action is usually associated with the operation of the disestablishing connection, for example, pulling electric cables.

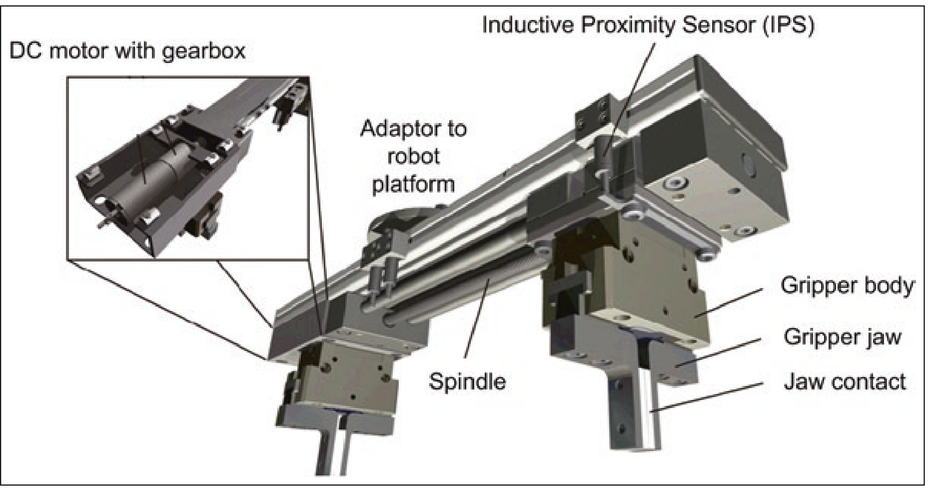


Figure 2.6: Grippers for disassembly of Lithium-Ion Battery [35]

2.2.2 Perception systems in automated disassembly

In general, when a system needs to make dynamic decisions (depending on a variable state or situation), it requires a perception system. Thus, the current situation should be recognised before any action in order to act in the proper way. Therefore, the term perception system encompasses the recognising and localising of the objects of disassembly and its fastening elements.

An accurate world model has a great importance for robotic assembly and disassembly. If all the positions and tolerances could be guaranteed, a robotic blind system could be feasible for dismantling applications. But it is usually inefficient and uneconomical to design a disassembly

system for a particular product. For this reason, flexible methods, like computer vision based systems, gain importance in this field.

In general, perception systems consist of:

- Geometric data obtention.
- Reading of the raw data provided by sensors: In order to convert this data into semantic information useful for scene creation, location of the components, etc...

Within the strong points of computer vision based systems can be distinguished:

- **High flexibility to positioning errors:** The automatic localisation of the components enables the system to reduce positioning errors. This is because the detections are done at the moment, just before realizing the operations.
- **Easy setting up for arrival products:** Given that the positions are found for each specific product, the position of the system or other specifications have not the importance that would have in a blind system.
- **Robust against product modifications:** Sometimes during the lifespan of the product, the dismantling object can have been undergone to modifications. These modifications are motivated for multiple reasons like upgrades, reparations, etc...
- **Verify the process by "seeing":** During the dismantling process, the system is capable of checking the current state.
- **Possibility of disassembling unknown products:** Using the generic structure of a product type.

Quick sensor technologies review

For this project, Zivid One 3D camera has been assigned. The Zivid One is a high precision structural light camera. Anyways, since for future stages of the project, the incorporation of other sensory technologies should be reconsidered, an analysis of these technologies has been done.

- **Stereo cameras:**

  Conventional cameras, sense a scene by measuring the reflection of ambient lighting in the visible spectrum by the objects to be disassembled. Mainly, conventional cameras capture 2D images, but a 3D scene can be constructed by using several cameras. Using this principle, the stereo vision cameras can capture 3D scenes.

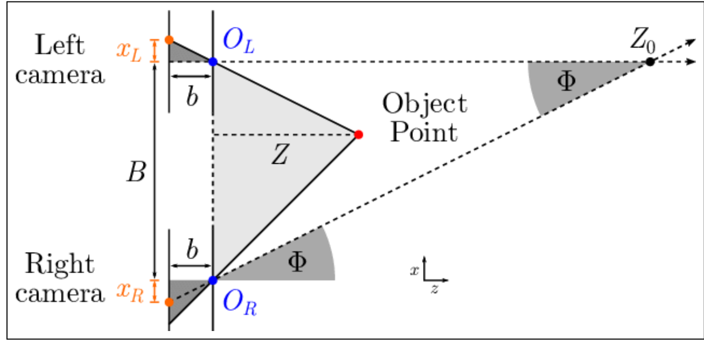


Figure 2.7: Stereo Cameras principle

- **Light Detection and Ranging (LiDAR) Scanner/Pulsed Time of Flight (ToF)**
  This kind of cameras uses active sensors, which emit energy to illuminate the target object. This energy is emitted in from of a not visible for humans pulsed laser. The same emitted laser pulse is received, and the distance can be derived using the return time and the wavelength of the laser. This operation is repeated point by point, determining the depth image. LiDARS can determine high precision and high-resolution depth maps.

- **Continuous Wave Time of Flight (ToF) camera:**
  Due to the expensive cost of LiDAR cameras, the Continuous Wave Time of Flight cameras were created. To find the depth, these type of cameras iluminate the whole scene with a continous wave modulated light and measure the phase shift of the received light wave (see figure 2.8).

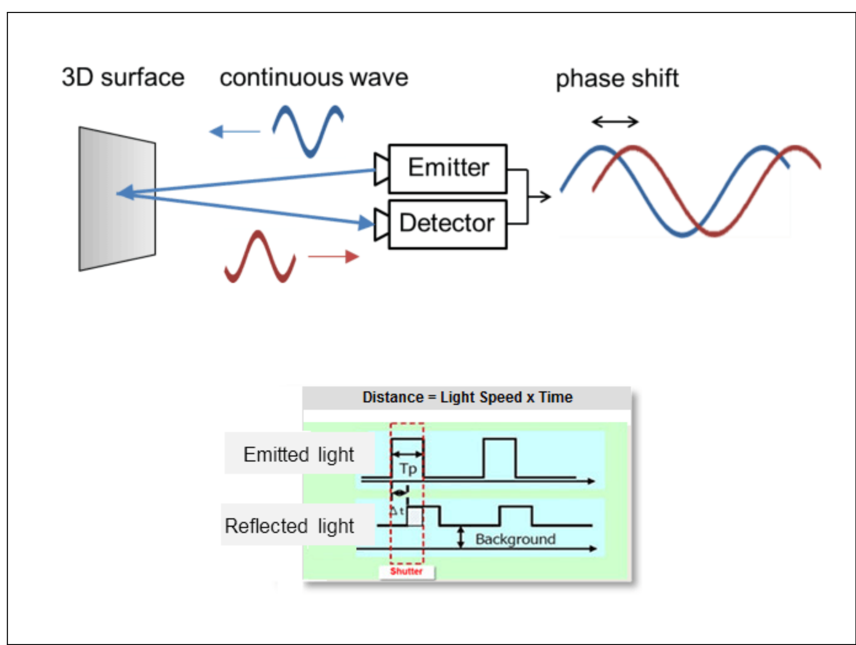


Figure 2.8: Continuous Wave Time of Flight (ToF) camera

- **Structured Light Camera:**
  Structured light cameras project a known light pattern onto the surface. The features of the surface distort the pattern. Analysing these distortions the surface topography can be reconstructed. See the full explanation in section 6.1.2.

2.3 Product analysis

The Hybrid and electric vehicles high voltage battery pack consist in individual modules and cells organized and connected in series and parallel." A cell is the smallest, packaged form a battery can take and is generally on the order of one to six volts. A module consists of several cells generally connected in either series or parallel. A battery pack is then assembled by connecting modules together, again either in series or parallel." [1]

Some of the most important and general components are:

- Battery modules.

– Battery management system (BMS).

– Cooling system.

– High voltage and information wires.

– Connective components.

2.3.1 Main components of a LIB pack

Battery modules (main component to be removed)

As mentioned in the introduction, the aim of the designed dismantling system is to disassembly all the components of the electric battery pack in order to obtain the battery modules. Battery modules are composed by a combination of a fixed number of cells, and they are the main component of the battery pack.

Battery modules usually incude a CMC (Cell Module Controller), to monitor and control the temperatures of the cells, the SOC (State Of Charge) and balance them so that their SOC is as equal as possible.

Battery management system (BMS)

The battery management system (BMS) is an electronic system that manages a rechargeable battery (cell or battery pack), such as by protecting the battery from operating outside its safe operating area, monitoring its state, calculating secondary data, reporting that data, controlling its environment, authenticating it and / or balancing it.

The battery management system (BMS) is an electronic system that controls a rechargeable battery (cell or battery pack). Its main functionalities are; to protect the battery from operating outside its safe operating area, to monitor its state, to calculate and report secondary data, to control the battery environment, and to balance the current states of the battery.

One of the essential parameters that are required to ensure safe charging and discharging is SOC. SOC is described as the present capacity of the battery stated in terms of its rated capacity. SOC provides the current status of the battery and allows batteries to securely be charged and discharged at a level proper for battery life enhancement. Consequently, SOC helps in the administration of batteries. Nevertheless, estimating SOC is not direct because it involves the measurement of the battery voltage, temperature, current and other information that pertains to the battery under consideration.

BMS is a separate entity with hardware and firmware and is connected to a battery charger rather than integrated within the charger. BMS consists of a number of sensing devices for monitoring battery parameters that will be used in the algorithm for SOC estimation.

Cooling system

The cooling system of the LIB is the responsible to keep the batteries cool. It is recommended to keep LIB at around 15°C and 45°C, if the appropiate temperature is not mantained, the life cycle of the LIB will shorten. Given the quite poor thermal stability and the excessive charging and discharging of these batteries during their lifespan, the cooling system has a very important role in this kind of batteries controlling the temperatures.

High voltage and information wires

From the one side, the **high voltage wires** are the responsible to carry the electric current. From the other side, the **information wires** transfer the diferent sensors information from the battery modules to the BMS:

Connective components

Finally, one of the most relevant components of the battery pack are the connective components. These are responsible for attaching and holding the other components.

Into a battery pack, many different connective components can be found. These can range from the simplest (i.e. screws) to special fasteners designed especially for a particular LIB model.

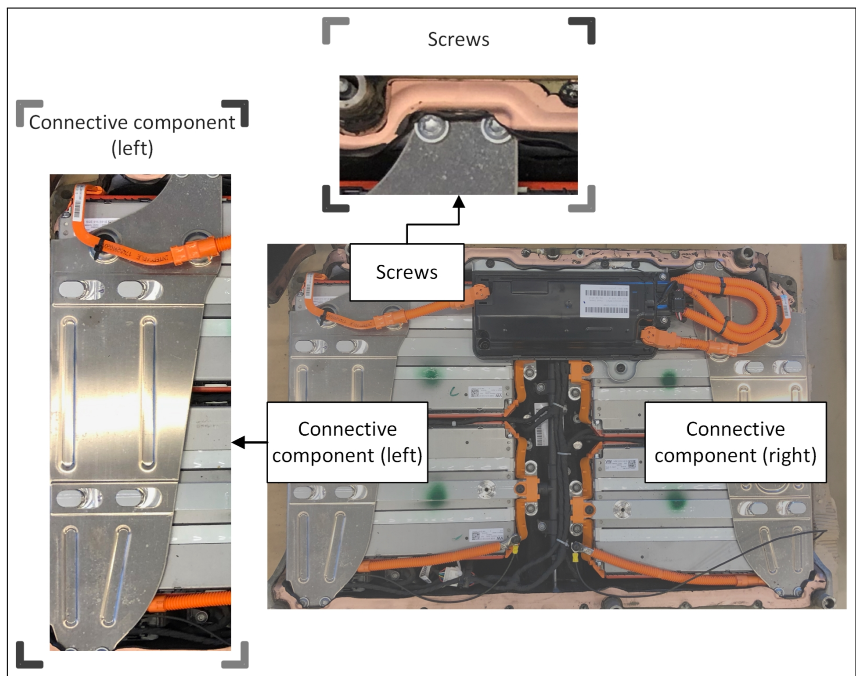


Figure 2.9: Example of connective components

2.3.2 Manual disassembly of a Volkswagen Hybrid LIB pack (25 Ah)

In order to better understand how the disassembly process should be, and given that the object detection has to be able to detect the majority of the components, a manual dismantling of a battery pack together with a visit to the company BatteriRetur (specialized in battery dismantling company) have been done.

It is essential to know all the main components of the battery and recognize how they are attached to the battery pack to identify the basic dismantling steps within the different pack models. Thus, as it can be seen in the Figure 2.10, a manual disassembly and a part listing of the Volkswagen battery pack have been done.

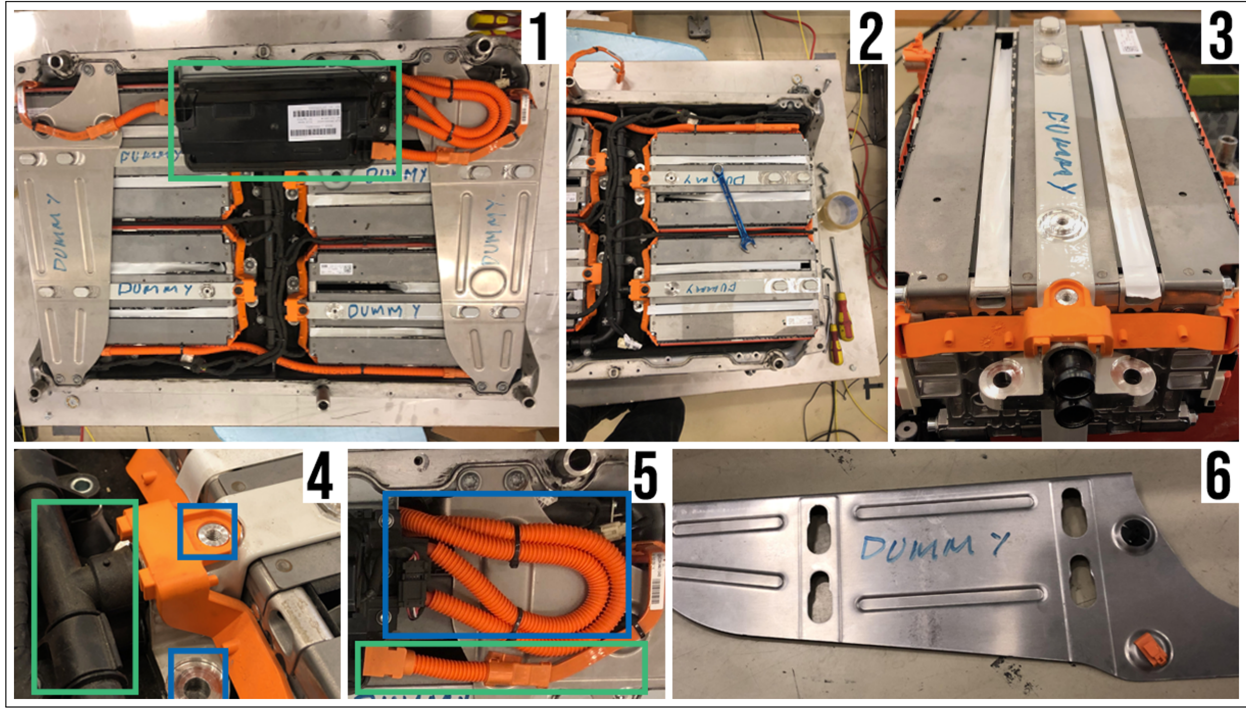


Figure 2.10: Manual dismantling of Volkswagen Hybrid LIB battery pack

Part listing

During the manual disassembly process, the main components described in section 2.3, have been observed. Thus the pack part listing is:

Part number	Part name	Location in Figure 2.10
1	LIB modules	Image 3
2	BMS	Green label in image 1
3	Screws	Blue labels in image 4
4	Connective elements	Image 6
5	High Voltage wires	Green label in image 5
6	"Information" wires	Blue label in image 5
7	Cooling system	Green label in image 4

Chapter 3

Robot Operating System communication

This chapter aims to present the basic concepts about ROS and the main ROS packages used in this project. It explains MoveIt! and the created MoveIt! package and finally shows how ROS has been implemented in this project.

3.1 Robot Operating System (ROS)

Although it is called Robot Operating System (ROS), it is not an operating system. ROS is a middleware for developing robotics platforms.

Technically, ROS is an open-source, meta-operating system for robots, and provides the services expected from an operating system like hardware abstraction, low-level device control, implementation of most used functionalities, communication between processes and package management. Tools and libraries, to ease the building, running and writing code across, are also provided [24].

3.1.1 ROS distributions

Every year in May, a ROS distribution is released. A ROS distribution is a version of a collection of ROS packages. The publication of the new ROS distributions goes according to the Linux distributions. The distributions released in odd-numbered years have standard ROS support for two years, instead the distributions released in even-numbered years have support for five years.

Due to the UiA robotics lab conditions, the ROS Kinetic Kame distribution has been used in this project. The ROS Kinetic release is stable for Ubuntu 16.04.

Distribution	Release date	EOL date
ROS Melodic Morenia	May 23rd, 2018	May, 2023
ROS Lunar Loggerhead	May 23rd, 2017	May, 2019
ROS Kinetic Kame	May 23rd, 2016	April, 2021
ROS Jade Turtle	May 23rd, 2015	May, 2017
ROS Indigo Igloo	July 22nd, 2014	April, 2019

Table 3.1: Last ROS Distributions

3.1.2 ROS background

ROS architecture is based on the publish-subscribe mechanism. Publish-subscribe is basically a messaging pattern. The basic idea is that the message posted by the publishers, is not programmed to be sent directly to the subscriber. Instead, the sent message is categorized into classes without knowing any information of the future subscribers. The same with the subscribers, they subscribe into a topic without knowing any information about the sender (publisher).

The basic ROS elements are:

- **Nodes:** A node is a process that performs computation. Nodes use the ROS client API and they are identified in the Master by its graph resource name. For this project, the Zivid camera and the robot are nodes. [27]

- **Topics:** Topics are named buses over which nodes exchange messages. [28]

- **Master or core:** The ROS Master provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer. [25]

- **Message:** A specific data structure, based on a set of built-in types, 8 used as type for topics. [26]

- **Services:** In the publish-subscribe mechanism the request-reply interaction is done with a Service. Basically, a service is a synchronous remote procedure call. The node that wants to call the service, sends a request message and waits for the reply. The service is offered by a node under a defined string name.

The following Figure 3.1, extracted from the chapter [6], shows the idea of the publish-subscribe (ROS topic) mechanism. Supposing two nodes, where the Node A is a publisher to a topic (Figure 3.1 a) and the Node B is a subscriber to the same topic (Figure 3.1 b), the roscore sends the network address to the subscriber (Figure 3.1 c), The subscriber contacts directly to each publisher, which results in a direct data exchange within the subscriber and the publisher (Figure 3.1 d. Many nodes can publish and subscribe to the same topic, which results in a N:M relation (Figure 3.1 e).

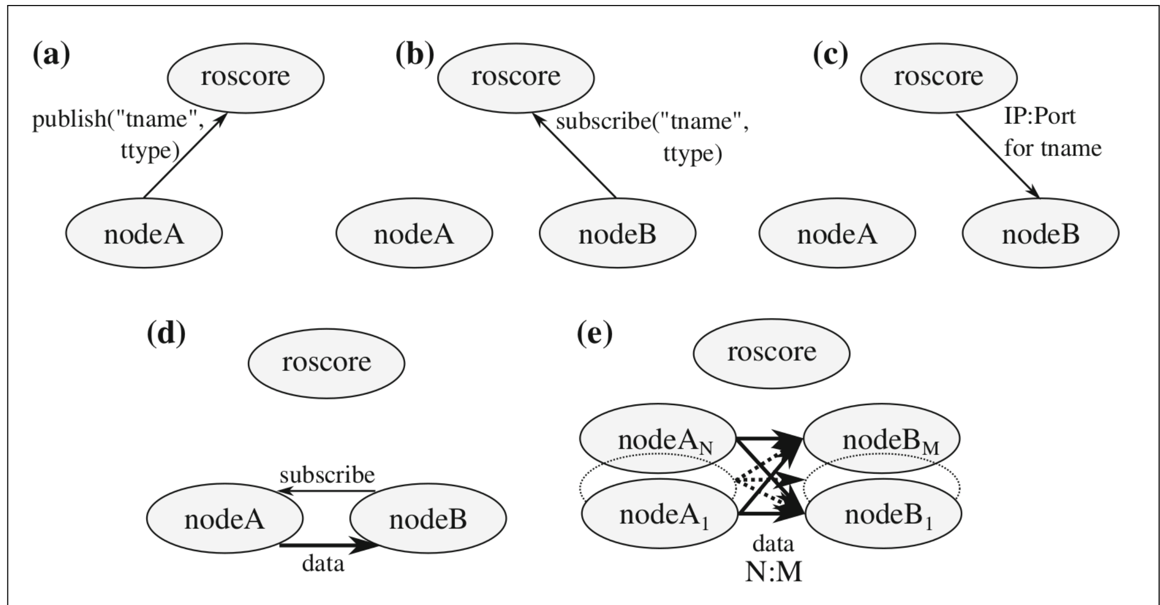


Figure 3.1: ROS topic mechanism

3.1.3 ROS in Python (rospy)

The whole code in this project has been done with the programming language Python (task planning, object detection, etc..). Python is a well supported language for the ROS-community. In this project, the Python library rospy has been used in order to realize the communication within the different nodes.

The aim of the rospy library is to create a client library for ROS using Python. Rospy enables the Python programmers to interface with ROS and eases the access to the ROS topics and services.

ROS Publisher

Initialisation of the Publisher.

```
pub = rospy.Publisher('topic_name', std_msgs.msg.String, queue_size=10)
```

To publish data.

```
pub.publish(std_msgs.msg.String("hello world"))
```

ROS Subscriber

To subscribe to a topic:

```
rospy.Subscriber("chatter", String, callback)
```

3.2 Moveit!

'MoveIt! is a software for mobile manipulation, incorporating the latest advances in motion planning, manipulation, 3D perception, kinematics, control and navigation. It provides an easy-to-use platform for developing advanced robotics applications, evaluating new robot designs and building integrated robotics products for industrial, commercial, Research and Development and other domains. MoveIt! is the most widely used open-source software for manipulation and has been used on over 65 different robots.' [7]

The aim of Moveit! is to provide the core functionality for manipulation in ROS.

- **Architecture:**

  The central node of the Moveit! architecture is the move_group. The main objective of the move_group is to manage different capabilities, integrate kinematics, planning and perception.

  The architecture of the move_group consists in an adopted from ROS plugin-based architecture [1].

  To interact with the move_group, users can use three API: C++, Python and through GUI.

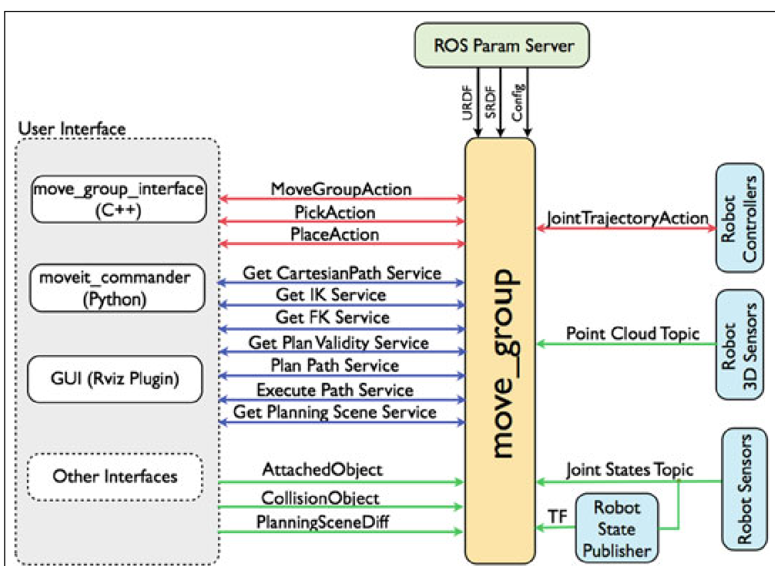


Figure 3.2: Moveit! Architecture scheme

- **Collision Checking:**

  In this case, FCL [20], which is a general-purpose library that integrates several techniques for fast and accurate collision checking and proximity computation, is the main package for native collision checking. The collision checking architecture follows a plugin architecture, allowing any collision checker to be integrated.

  Collision checking is usually (computationally) the most expensive part of motion planning. It lasts 80%-90% of the motion plan calculation time.

- **Kinematics:** The aim of inverse and forward kinematics is to calculate the variable joint parameters needed to place the end effector or the tool centre point in a certain position and vice versa. [8]

[1] A plugin architecture is an architecture that will call external code at certain points without knowing all the details of that code in advance.

MoveIt! uses a plugin achitecture for solving inverse kinematics as well. A numerical solver (useful for any robot) is used as a native solver for inverse kinematics. Nevertheless, users are free to add their own custom solvers. For forward kinematics, MoveIt! provides a native implementation.

Specifically, analytic solvers are much faster than the native solvers. I.e. IKFast is one of the most popular analytic solvers for industrial arms [7].

- **Motion planning:**

  The plugin architecture allows MoveIt! to use different motion planners from multiple libraries. A ROS action or service, offered by the ros node move_group, allows the interface to the motion planners. Generally, the configuration of the default motion planners is done through MoveIt! setup assistant. [7]

  One of the most used motion planner libraries is the Open Motion Planning Library (OMPL). OMPL consists of many state-of-the-art sampling-based[2] motion planning algorithms and it is aimed at three different audiences: Motion planning researchers, robotics educators and end-users in the industry. Within the different algorithms collected in the OMPL can be found [34]:

  - Probabilistic Roadmap Method (PRM)
  - Rapidly-expanding Random Trees (RRT, RRTconnect, lazy RRT)
  - Kinodynamic Planning by Interior-Exterior Cell Exploration (KPIECE, bidirectional KPIECE, bidirectional lazy KPIECE)
  - Single-query bi-directional probabilistic roadmap planner with lazy collision checking (SBL)

- **Planning scene:**

  The planning scene is the representation of the world around the robot, the last state of the robot is represented as well. It is stored in the planning scene monitor placed inside the move_group node.

  The planning scene monitor listens to:

  - The joint_states topic and the tf (transform tree). To find the robot state information.
  - Using a world geometry monitor creates the 3D occupancy information using the information received from the sensors.
  - From user input or other sources, the world geometry information.

[2] The fundamental idea of sampling-based motion planning is to approximate the connectivity of the search space with a graph structure. The search space is sampled in various ways, and selected samples end up as the vertices of the approximating graph. [34]

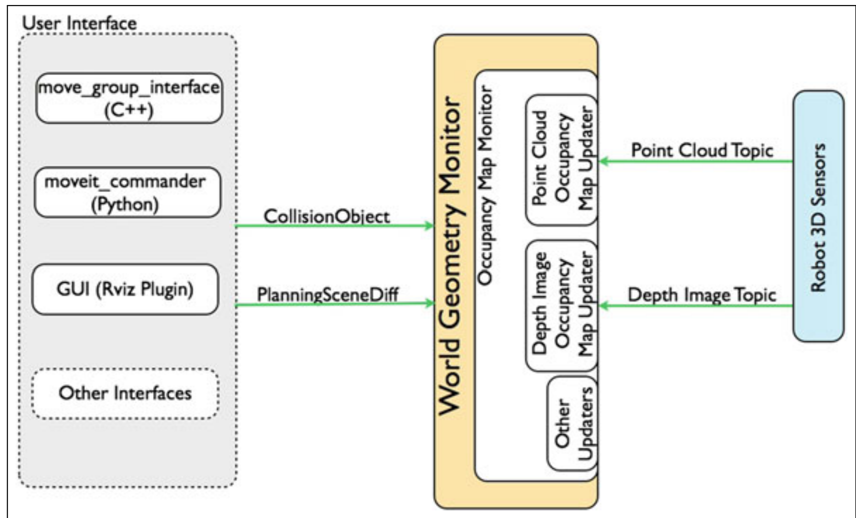


Figure 3.3: Moveit! Architecture scheme

3.2.1 MoveIt! Setup Assistant

The MoveIt! Setup Assistant is a GUI software designed to allow and help users creating new MoveIt! packages for different robot models. The main function of the setup assistant is to generate a Semantic Robot Description Format (SRDF) file for the robot. [7]

Semantic Robot Description Format (SRDF)

Basically, an .srdf file is a representation of the semantic information about robots. The intention of this files is to represent and add some information not included to the .urdf files such as collisions between the links. Thus, add information that has a semantic aspect to it. [31]

The main tags in SRDF are:

- **<robot>**: Root tag, all the information is included in this tag.
- **<group>**: Links and joints representation.
- **<group_state>**: Define a specific state for a **<group>**.
- **<link>**: Specify that a link is part of a **<group>**.
- **<joint>**: Specify that a join is part of a **<group>**.
- **<chain>**: Kinematic representation of a chain.
- **<end_effector>**: Information regarding an end effector.
- **<disable_collisions>**: Disable the collision checking between a specific pair of links.
- **<passive_joint>**: Change the default active state for a specific **<joint>**.

MoveIt! Setup Assistant

The main parts or steps for creating a new MoveIt! package using the Setup Assistant are:

- Launch the application:

  The first step is starting the setup assistant.

  ```
      $~ rosrun moveit_setup_assistant moveit_setup_assistant
  ```

  This action opens a window where the user has to select between two options: Create a new MoveIt! configuration package and Edit existing MoveIt! configuration package. Then, the robot configuration (URDF file or a xacro file) should be loaded.

- Generating the Self-Collision Matrix:

  The self-collision matrix is generated using a random sampling method. The sampling settings are selected by the user. Using high sampling values gives better results but slows down the process of generating the MoveIt! configuration package. A typical value is 10.000 samples.

- Add virtual joints:

  Virtual joints are needed to specify where is the robot in the world reference. There are two types of virtual joints: fixed and related to a motion of a mobile frame. When adding a virtual joint, there have to be a source of information about the transform (i.e a tf static transform publisher).

- Planning groups:

  The planning groups creation is the main purpose of the SRDF. The aim of planning groups is to connect (semantically) different parts of the robot. Usually, move groups are defined by a set of joints but they can be defined as a chain (specifying the first and the tip link).

  The kinematics of the created group can be solved by the MoveIt! default kinematic solver, KDL Kinematics solver (in the Kinematics Dynamics Library package *KDL*). Anyways, a specific kinematics solver can be defined.

- Robot poses:

  User-defined fixed poses of the robot can also be defined in the SRDF file. I.e home position.

- Passive joints:

  The passive joints are those that cannot be used for planning or controlling the robot. In this step, these joints are specified.

- Configuration Files:

  The last step, is to generate the MoveIt! configuration package. Once this is done, a simulation of the robot can be launched (demo.launch file).

  ```
  $~ roslaunch <moveit_config_package_name> demo.launch
  ```

3.2.2 ABB IRB4400 + track IRBT4004 MoveIt! package creation and configuration

For this project, an specific MoveIt! package for the righty robot has been created. In this case, the righty robot is an ABB IRB4400 robot mounted on an ABB track IRBT4004. Both, are controlled

by the IRC5 controller and the created MoveIt! package is capable to move the robot around the combined workspace [3].

During all this section, the righty robot refers to the group robot plus track.

Creation of the URDF robot model

URDF files and the xacro files are different formats to save and load the robot descriptions. In this case, the whole MoveIt! package is configured to use the URDF file of the righty robot. Note that working with both formats gives at the end the same results.

In this case the URDF file has been created from the righty.xacro file. The righty.xacro file has been provided by Atle Aalerud (wp3_robots package). Thus, the URDF file has been created running:

```
~$ rosrun xacro xacro --inorder -o model.urdf model.urdf.xacro
```

The provided righty.xacro file loads all the information regarding to the IRB4400 manipulator and the IRBT4000, defines the relative position between them and the position of the track in reference to the world.

The independent xacro files for the manipulator and the track are provided in the ROS-Industral ABB support on Github [4].

To include the track description file:

```
<xacro:include filename="$(find abb_irbt4004_support)/urdf/irbt4004r_macro.xacro"/>
<xacro:abb_irbt4004r prefix="${prefix}"/>
```

To include the manipulator description file.

```
<xacro:include filename="$(find abb_irb4400_support)/urdf/irb4400_60_macro.xacro"/>
<xacro:abb_irb4400_60 prefix="${prefix}"/>
```

Tool definition

As explained before in the project objectives and scope section 1.2, the design of the tools is not included in the scope of the project. Anyways, a simulated tool has been defined. Note that the dimensions and the orientations of the tool have been invented.

Thus, to define a new tool, the URDF file has to be modified. First of all, a new link should be defined. For instance, to define the tool 1:

```
<link name="tool1"/>
```

Given the dimensions of the tool, and the link where the tool is attached, the next step is to define a joint. The rpy and the xyz values, are referenced from the parent frame (in this example the righty_tool0).

```
</joint>
    <joint name="righty_tool0−tool1" type="fixed">
```

[3]The combined workspace is defined by the reachable positions by the robot combining the moves done by the track plus the moves done by the own robot.

[4]Link: *https://github.com/ros-industrial/abb*

```
    <parent link="righty_tool0"/>
    <child link="tool1"/>
    <origin rpy="0 0 0 " xyz="1.1 0.0 0.0"/>
</joint>
```

In case of having to define a real tool to work on real removal operations, it is recommended to enable the tool collisions and the tool visuals. To do so, the collision and visual meshes should be added.

To enable the tool visuals:

```
<link name="tool_name_here">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package:path_to_package.stl"/>
      </geometry>
      <material name="">
        <color rgba="0.9254902 0.9254902 0.9058824 1"/>
      </material>
    </visual>
</link>
```

To enable tool collisions:

```
<link name="tool_name_here">
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package:path_to_package.stl"/>
      </geometry>
      <material name="">
        <color rgba="0.9254902 0.9254902 0.9058824 1"/>
      </material>
    </collision>
  </link>
```

Move group creation

The move group is one of the basis of the ROS communications with the robot. It is declared in the srdf file (righty.srdf). More than one move group for the same manipulator can be defined in the same srdf file.

In this case, the move group 'righty_tcp' has been created. It has been defined as a chain where the base link is the righty track and the tip link is the simulated tool. Using this move group, the whole set (manipulator + track) moves to reach the desired positions.

```
<group name="righty_tcp">
    <chain base_link="righty_track_right" tip_link="tool1" />
</group>
```

In case of wishing a move group chain to move only the manipulator, the base link should be changed for the IRB4400 base link.

Camera configuration (pose and collision checking)

The definition of the MoveIt! camera settings, it is very similar to the tool definition.

Fistly, the new link for the camera has been defined.

```xml
<link name="zivid_optical_frame"/>
```

Then, using the extrinsic calibration of the camera (explained in section 6.3.3, the joint between the tool1 and the camera link has been declared.

```xml
<joint name="tool1-zivid_optical_frame" type="fixed">
    <parent link="tool1"/>
    <child link="zivid_optical_frame"/>
    <origin rpy=" -1.55510702 0.00474244 -1.76689492" xyz="-1.0406646482792733
        0.28582285515849640 0.043799123724683504"/>
</joint>
```

Finally, it has been downloaded (from the official Zivid documentation web-page [43] the camera 3D mesh in .stl format. After that, the collisions and the visuals have been set up.

```xml
  <link name="zivid_optical_frame">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://wp3_robots/meshes/zivid_camera/zivid_one_cad.stl"
            scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="">
        <color rgba="0.9254902 0.9254902 0.9058824 1"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://wp3_robots/meshes/zivid_camera/zivid_one_cad.stl"
            scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="">
        <color rgba="0.9254902 0.9254902 0.9058824 1"/>
      </material>
    </collision>
  </link>
```

3.2.3 Essential coding to move the robot

To move the robot has been and is one of the most important and basic tasks of the project.

To run the robot using Python, ROS and MoveIt!, the moveit_commander class plays an important role.

Some of the most important class functions are (see the full reference in shorturl.at/rEIJ9):

- RobotCommander() -> Instantiates a RobotCommander object.

- PlanningSceneInterface() -> Instantiates a PlanningSceneInterface object. This object is an interface to the world robot environment.

- set_planner_id() -> Sets which motion planner use when motion planning.

- plan() -> Calculates the motion plan within the current state and the goal state using the specified motion planner.

- execute() -> Executes the plan.

- stop() -> Ensures that there is no residual movement.

In the appendix B.12 a basic code based on Python, ROS and MoveIt! (MoveGroupCommander(), roscpp, etc..) has been presented.

3.3 ROS implementation

In this project, ROS has been used to integrate the system elements. As it can be observed in Figure 3.4, the Zivid camera, the ABB IRB4400 robot, the rack pc, and the laptop where the system is running, are continuously communicating through the ROS Master.

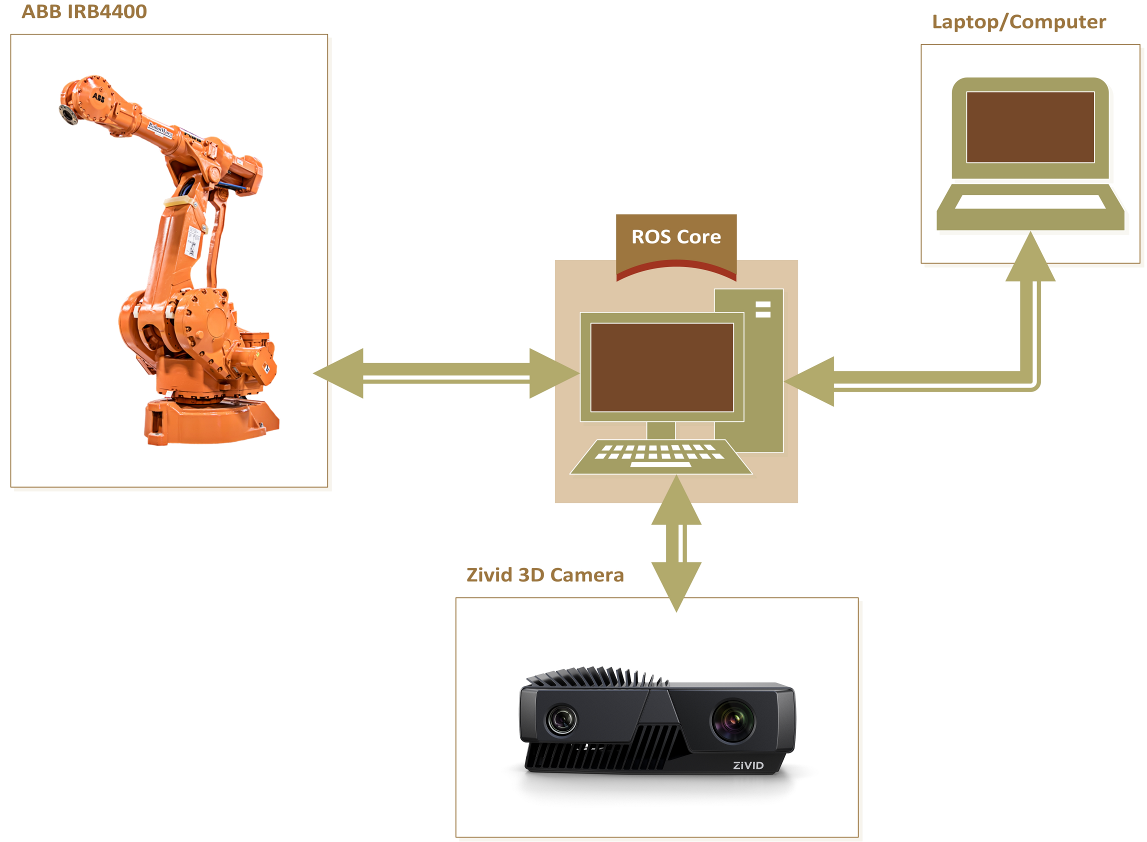


Figure 3.4: System elements scheme

3.3.1 Zivid 3D camera ROS driver

To realise the integration of the camera, the official Zivid ROS package has been used.

Launch the camera node

After connecting the camera to an USB3 port, to launch the camera node using the predefined parameters:

```
$~ ROS_NAMESPACE=zivid_camera rosrun zivid_camera zivid_camera_node
```

But, the following parameters can be redefined when launching the driver:

- *file_camera_path* (string, default: ""): In case of using a file camera, not the real one. This can be used for developers without access to hardware.
- *frame_id* (string, default: "zivid_optical_frame"): Specify the frame_id used for all published images and point clouds.
- *num_capture_frames* (int, default: 10): Specify the number of dynamic_reconfigure capture/frame_<n> nodes that are created. This number defines the maximum number of frames that can be a part of a 3D HDR capture

Services

The main ROS services offered by the Zivid ROS driver are:

- **Capture_assistant/suggest_settings** (*zivid_camera/CaptureAssistantSuggestSettings.srv*): Analyze the scene and suggest settings for the particular scene, camera distance and other conditions.
- **Capture** (*zivid_camera/Capture.srv*): Launches a 3D capture. The resulting point cloud and the color image are published to the corresponding topics.
- **Capture_2d** (*zivid_camera/Capture2D.srv*): Useful to trigger a 2D capture. The resulting image is published to the topic *color/image_color*.
- **Is_connected** (*zivid_camera/IsConnected.srv*): Returns the camera state from the prespective of the ROS driver (connected/disconnected).

Topics

The data is published in the following topics:

- *color/camera_info* (message type: CameraInfo): Camera calibration and metadata [5].
- *color/image_color* (message type: Image): RGB color image. In the capture.srv service, the image is encoded as rgb8 and for the capture2d.srv service the image si encoded as rgba8.
- *depth/image_raw* (message type: Image): Depth image. Instead of containing the RGB information, each pixel contains the z value (NaN in case of a missing value).
- *points* (message type: Pointcloud2): Point cloud data. The output is in camera frame. The published information for each point is: x(m), y(m), z(m), c(contrast value), r,g, and b (colors).

Set the frame settings

To realize the set up and change the different camera parameters (see section 5.1):

- Brightness (double)
- Enable (bool)
- Exposure_time (int)
- Gain (double)
- Iris (int)

For instance, to configure the parameters defined in section 5.1 for the frame 0 [6], the next Python code would be used:

[5] data that provides information about other data

[6] The frame 0 configuration is the one used for the 2D images.

```
frame0_config_client = dynamic_reconfigure.client.Client("/zivid_camera/capture/frame_0")
frame0_config = {"enabled": True, "iris": 17, "exposure_time": 60000, "brightness":1.0, "gain":
    4.0}
frame0_config_client.update_configuration(frame0_config)
```

For 3D HDR captures, with more than one frame, each frame can be configured the in the same way.

3.3.2 ABB ROS-Server

In order to operate and communicate with the robot using ROS, the ABB ROS-server has been installed in the robot controller. First, the controller needs to meet some requirements, and then the ROS-server can be installed and configured.

Controller prerequisites

The ABB ROS Server code is written in RAPID, using a socket [7] interface and multiple parallel tasks. So, the controller needs to have the next specific packages installed to allow the ROS communication.

- **Multitasking (623-1):** Multitasking gives the possibility of executing up to 20 programs (tasks) in parallel, including the main program.
- **Socket Messaging (672-1):** Socket Messaging, allows the RAPID program to exchange TCP/IP messages over a network, with a C/C++ or Python program on another computer or rapid program running into another robot controller. It is constantly receiving messages over the Ethernet channel of the controller (IRC5).

Installing the ROS-Server

Then, the next files have been downloaded and copied to the robot controller. they have been saved into the ROS sub-directory (HOME/ROS/*).

- **ROS_common.sys**: Global variables and data types shared by all files.
- **ROS_messages.sys**: Implementation of specific message types.
- **ROS_motion.mod**: Issues motion commands to the robot.
- **ROS_motionServer.mod**: Receive robot motion commands.
- **ROS_socket.sys**: Socket handling and simple_message implementation.
- **ROS_stateServer.mod**: Broadcast joint position and state data.

Download link: https://github.com/ros-industrial/abb/tree/indigo-devel/abb_driver/rapid

[7]'Sockets are a generalization of the Unix file access mechanism that provides an endpoint for communication, either across a network or within a single computer. A socket can also be thought of as an extension of the named pipe concept that explicitly supports a client/server model, wherein multiple clients may be attached to a single server.' [32]

ROS-Server configuration

1. The **tasks** have been created.

Name	Type	Trust Level	Entry	Motion Task
ROS_StateServer	SEMISTATIC	NoSafety	main	NO
ROS_MotionServer	SEMISTATIC	SysStop	main	NO
T_ROB1	NORMAL	-	main	YES

2. Definition of the following **Signals**.

Name	Type of Signal
signalExecutionError	Digital Output
signalMotionPossible	Digital Output
signalMotorOn	Digital Output
signalRobotActive	Digital Output
signalRobotEStop	Digital Output
signalRobotNotMoving	Digital Output
signalRosMotionTaskExecuting	Digital Output

3. **Tie Signals** to the system output:

Signal Name	Status	Arg 1	Arg 2
signalExecutionError	Execution Error	N/A	T_ROB1
signalMotionPossible	Runchain OK	N/A	N/A
signalMotorOn	Motors On State	N/A	N/A
signalRobotActive	Mechanical Unit Active	ROB_1	N/A
signalRobotEStop	Emergency Stop	N/A	N/A
signalRobotNotMoving	Mechanical Unit Not Moving	ROB_1	N/A
signalRosMotionTaskExecuting	Task Executing	N/A	T_ROB1

4. **Link** modules with the tasks.

File	Task	Installed	All Tasks	Hidden
HOME:/ROS/ROS_common.sys		NO	YES	NO
HOME:/ROS/ROS_socket.sys		NO	YES	NO
HOME:/ROS/ROS_messages.sys		NO	YES	NO
HOME:/ROS/ROS_stateServer.mod	ROS_StateServer	NO	NO	NO
HOME:/ROS/ROS_motionServer.mod	ROS_MotionServer	NO	NO	NO
HOME:/ROS/ROS_motion.mod	T_ROB1	NO	NO	NO

Robot Motion Task: Manual and auto modes

When running the robot, the T_ROB is the responsible task for launching the robot motion. Depending on the controller mode (**auto** or **manual**) the task has a different behaviour.

While running in manual mode (used mode during all the tests done in this project):

– The robot moves at a reduced speed.

– The enable switch must be held during all robot movements.

– Since the movement execution can take longer than expected, ROS may try to cancel the move.

While running in automatic mode:

– The robot moves at full speed.

– It is very important that the workspace is clear of personal and obstacles.

3.3.3 Catkin Workspace

A catkin workspace is the folder where all the catkin packages are modified, built, and installed. The catkin workspace can contain up to four spaces (source, build, development and install spaces). Each space has a role in the software development process.

- **Source Space:** The source space contains the code of catkin packages. Inside each folder, placed in the source space, one or more catkin packages can be contained.
- **Build Space:** In the build space is CMake is invoked in order to build the catkin packages in the source space. CMake and catkin keep the cache information and other files here.
- **Development (Devel) Space:** The development space (devel) is where built targets are located before being installed. The way targets are created in the devel space is the same as their design when they are installed.
- **Install Space:** Once targets are created, they can be installed into the install space by requesting the install target, usually with make install. The install space does not have to be contained in the workspace.

In the catkin workspace of the project, there have been placed the source files for the ABB robots (*catkin_ws/abb*), the Zivid 3D camera (*catkin_ws/zivid-ros*) and the specifically created files for using the Lab robots (*catkin_ws/wp3_robots-master*).

Chapter 4

Guidance: UiA Robotics Lab ROS set up manual

Note: This guidance has been done using Ubuntu 16.04 and ROS Kinetic, it has not been tested with different Operating System or different ROS versions.

1. **Network and .bashrc file configuration:**

   In the UIA robotics Lab (henceforth lab) it is used a distributed ROS core structure. This means that the ROS core is running in the rack PC an the other computers are nodes that communicate with the ROS core (rack pc in this case). Thus, to use an external computer the Network Address of this computer (ROS node) has to be declared.

   First of all, install ROS following the steps and instructions explained in the official ROS tutorial. ROS installation guidance link

   Once ROS is correctly installed, the network settings of the computer have to be modified. First, plug-in the Ethernet cable to establish a network connection between the computer and the rack PC. Then, go to the top bar and press *Network settings > Edit connections...*. In Network connections select *Add > Ethernet*.

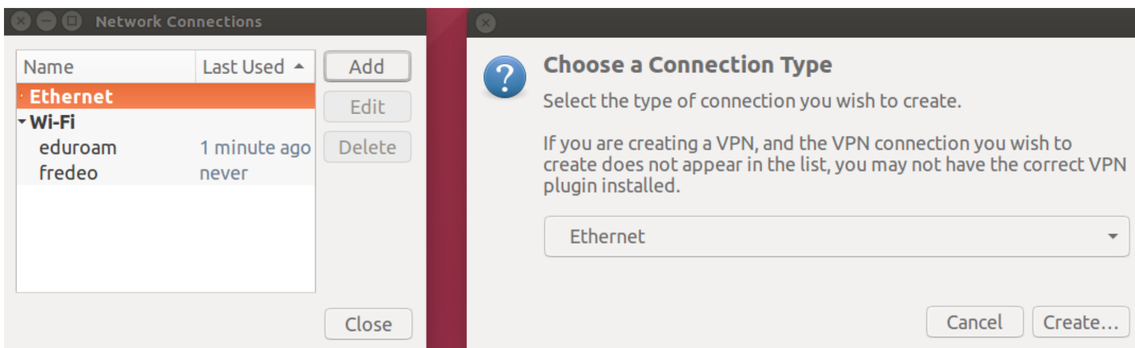
   

   Figure 4.1: Caption

   Set the IPv4 Settings. Modify the Method to Manual (*Method > Manual*) and add a new address for the computer, this address has to be predefined in the rack PC and will be the IP for the ROS Node, in Addresess (*Add*).

   Set the IPv6 Settings. Set Method to Ignore (*Method > Ignore*).

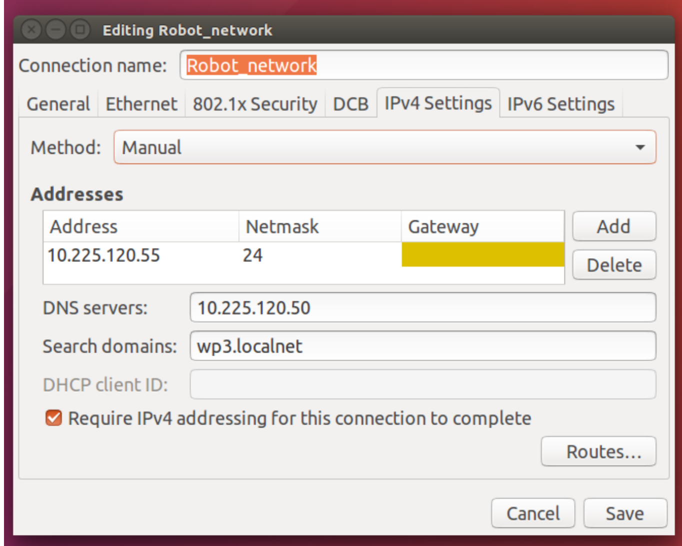


Figure 4.2: IPv4 settings.

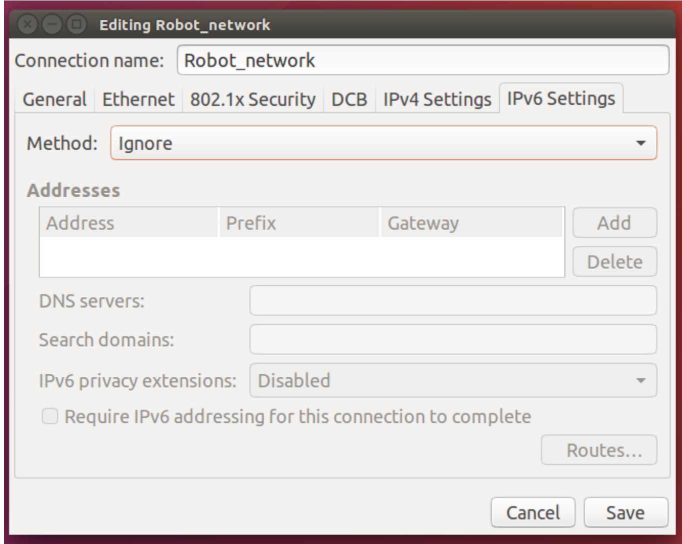


Figure 4.3: IPv6 settings.

After configuring the network properly, the .bashrc file has to be modified. The .bashrc file is a shell script that Bash runs whenever it is started interactively. This file has to be modified in order to set the new ROS MASTER (the rack PC). Open the .bashrc file typing:

```
~$ sudo nano .bashrc
```

Move to the final part of the document and add the following lines:

```
#Define the adress of the rack PC (ROS MASTER)
export ROS_MASTER_URI=http://10.225.120.50:11311
#Define your IP
export ROS_IP=#write_your_IP_here#
#Define the Host Name
export ROS_HOSTNAME=#your_name#.wp3.localnet
```

Now everything is defined and we are able to operate with ROS normally.

2. **Create and configure your catkin workspace**:

   Create your catkin workspace following the official tutorial: How to create a catkin workspace link.

   Once your catkin workspace is created, download onto the *src* folder the abb ROS support.

```
~$ cd catkin_ws/src
~$ git clone https://github.com/SFI−Mechatronics/wp3_abb
```

   Rebuild the catkin workspace:

```
~$ catkin build
```

3. **Launch the demo.launch file and move the simulated robot using Python:**

   MoveIt! configurations for both robots in the lab are located in the folder wp3_robots-master . In each configuration folder there are the config and the launch folders (see Figure 4.4).

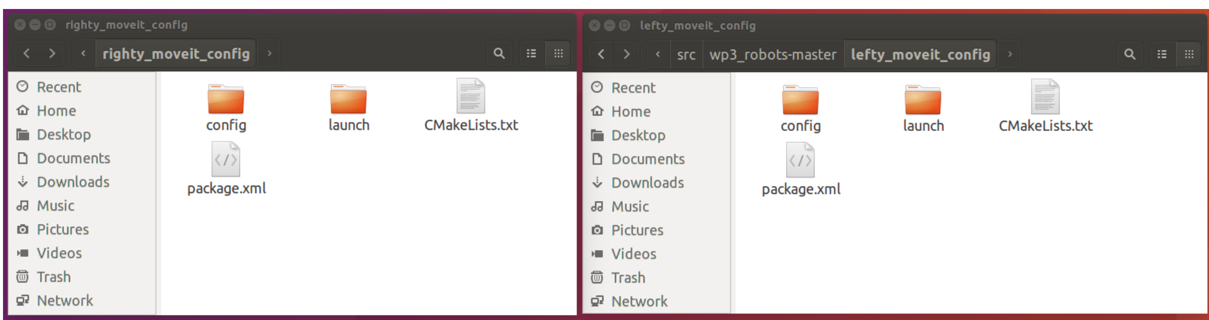


Figure 4.4: MoveIt! Packages for righty and lefty.

To work with these MoveIt! packages, MoveIt! should be installed Link: MoveIt! installation Tutorial.

If it is the first time working with MoveIt! it is strongly recommended to do the Link: MoveIt! Tutorial in order to have a basis in the source.

Take a look into these folders (Figure 4.5)

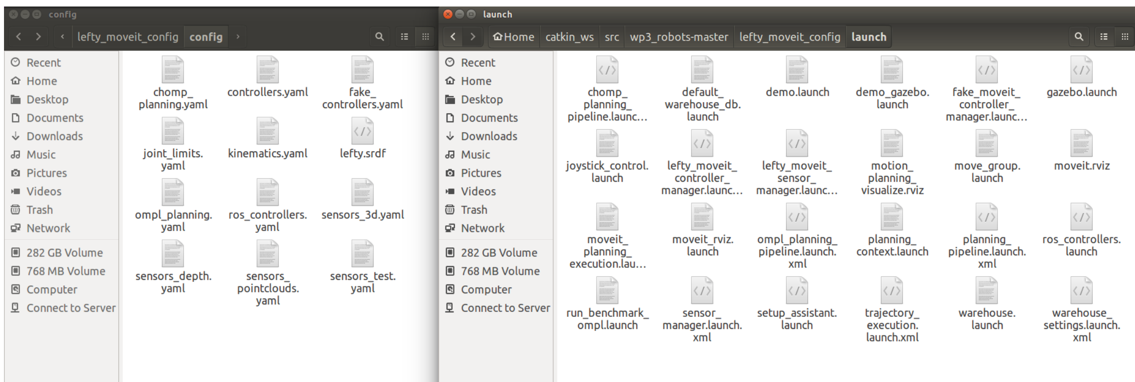


Figure 4.5: MoveIt! files for righty and lefty.

Launch the *demo.launch* file. This will show a 3D representation of the robot in Rviz. This demo file launch a simulation of the robot (working in fake execution mode)

```
~$ roslaunch righty_moveit_config demo.launch
```

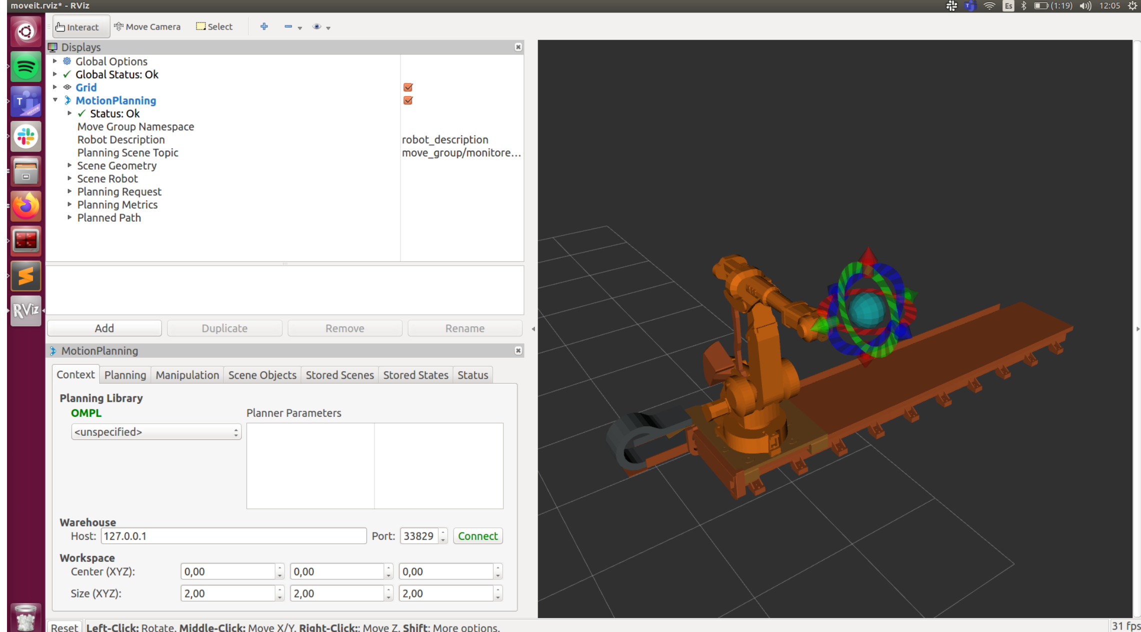


Figure 4.6: Righty simulation in Rviz.

- *Troubleshooting while launching the demo.launch file:*

```
eduard@eduard-GL65-9SFK:~$ roslaunch lefty_moveit_config demo.launch
... logging to /home/eduard/.ros/log/a4b4c7ca-637e-11ea-9ed4-00d861895b40/roslaunch-eduard-GL65-9SFK-5505.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

Invalid roslaunch XML syntax: not well-formed (invalid token): line 48, column 5
The traceback for the exception was written to the log file
eduard@eduard-GL65-9SFK:~$
```

Figure 4.7: Possible emerging error.

To fix that, go to demo.launch file and comment the following line:

<!−− arg name="pipeline" value="$(arg pipeline)"/ −−>

The next step is try to move the simulated robot using Python. To do so, it is going to be used the Python moveit_commander interface. This interface allows us to move the predefined move group.

Create a Python script (i.e moverobot.py): (See code in appendix E.1)

Note: It is very important to understand the lines of this code. Check (Link: MoveGroup-Commander functions) to know all the possibilities of working with Python interface

After running this Python script using *rosrun* the robot simulation should move (see in Figure 4.8). *REMEMBER: the Python script should be an executable script. To make it executable run* ***sudo chmod +x /usr/share/testfolder/aFile***

- *Troubleshooting selection of an unreachable position.* As it is logical, if you select an unreachable position the motion planner (RRT, RRTconnect, PRM, etc..) is not able find a path to reach the position. Thus, the terminal shows the next error:

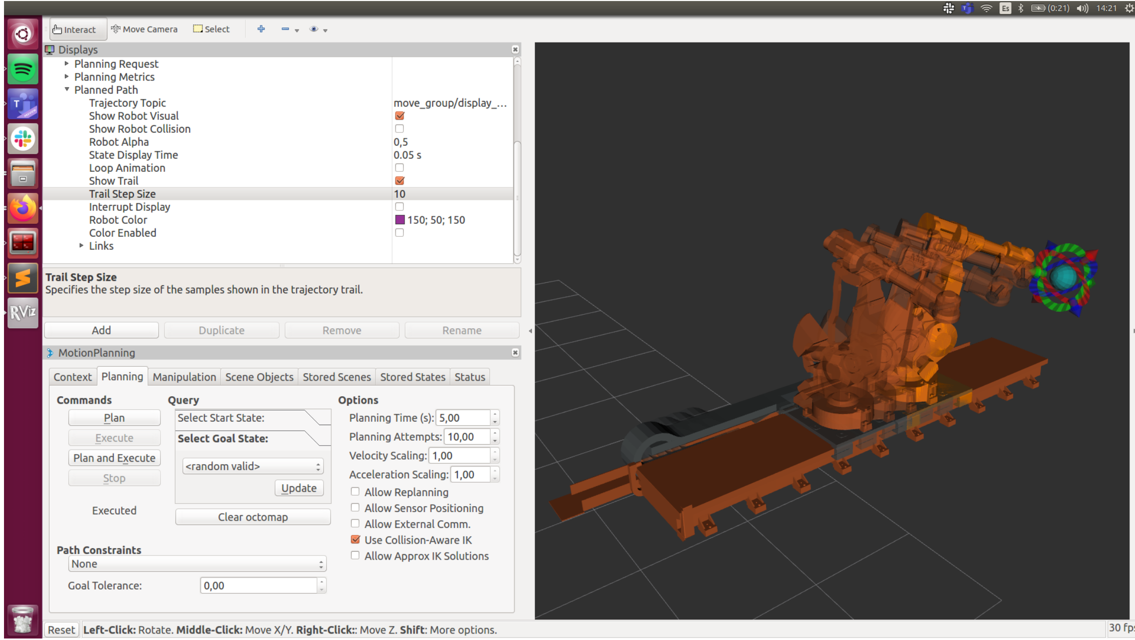


Figure 4.8: Rviz: Robot simulation moving to a defined position.

```
[ INFO] [1583926875.235648674]: Planner configuration 'lefty_tcp' will use planner 'geometric::RRTConnect'. Additio
nal configuration parameters will be set when the planner is constructed.
[ INFO] [1583926875.235869051]: RRTConnect: Starting planning with 1 states already in datastructure
[ERROR] [1583926876.546437002]: RRTConnect: Unable to sample any valid states for goal tree
[ INFO] [1583926876.546463951]: RRTConnect: Created 1 states (1 start + 0 goal)
[ INFO] [1583926876.546475122]: No solution found after 1.310741 seconds
[ INFO] [1583926876.546487790]: Unable to solve the planning problem
[ INFO] [1583926876.546998276]: Received event 'stop'
```

Figure 4.9: Error: Not reachable position.

4. **Establish connection with the real robot and move the real robot to a defined position.**

   Once the simulation is working, let's work with the real robot. First, test the robot visualization using rviz. Thus, launch the robot communication.

   ~$ roslaunch wp3_robots show_righty.launch

   This will show the simulation with the real-time values of the joints states of the real robot. The robot is publishing the current joints states.

   Close the robot state visualization tab (show_righty.launch), and launch the real robot.

   To launch the real robot:

   ~$ roslaunch wp3_robots load_robot_mod.launch

   To launch the load_robot_mod.launch file, is the equivalent action on the real robot of launching the demo.launch file on the simulated robot. Therefore, at this point, working with the real robot is almost the same as working with the simulated robot.

   Anyways, note that to move the real robot the ROS_main task should be runned in the robot controller. To run this task -> (*PROGRAM EDITOR/DEBUG/ROS_main*). When the program is loaded press the safety button + PLAY button.

   Thus, to move the real robot, now run again the previously created Python based script (moverobot.py). Code in appendix E.1.

Chapter 5

Object detection

In this project, object detection is crucial to find and to distinguish all the elements placed in the dismantling scene. From object detection, it is extracted necessary information used by pose estimation and consequently by the task planner. Therefore, good results in this part are also important for having good global resuts.

Only 2D images have been utilized for this part. Thus, the used algorithm aims to find the position of the components in the 2D image. This information is used afterwards in pose estimation to determine the world reference base frame positions of the LIB pack components (see Pose Estimation chapter 6). Another possibility is to use 3D object detection as presented in [42]. In this project, 2D object detection has been preferred over 3D object detection for computational reasons, 3D convolution requires a lot of computational power.

Why computer vision? "Experience in the field of robotic disassembly has indicated that robotic disassembly cannot simply be considered as the reversal of assembly, due to two main factors: the use of irreversible fasteners and the presence of a higher degree of uncertainties." [39] So, given these factors, a smarter system is needed.

Computer vision allows the reaction of the system in front of deviations and uncertainties in the object of disassembly. To react against these is not possible when the systems depend entirely on a static database. Moreover, the automatic detection of the components eases and reduces the preprocessing and the preparation of the entrance of the object of disassembly to the system. A system of this kind is also robust in front of production modifications.

This chapter reports the camera parameter adjustment, describes the used algorithm and explains the procedure done for training the algorithm.

5.1 Image capture: Zivid one parameter adjustment

To have positive results in image recognition, it is crucial to have good quality images. Good quality images help the algorithm to have better results. Then, if the training images are clear, better training results are obtained as well.

The adjustment of the camera parameters has great importance in taking good quality images. In this case, images are obtained using the Zivid camera (see section 6.1).

The main modifiable parameters for the Zivid camera are the exposure time, the iris aperture, the projector brightness and the gain. So, the adjustment of the camera parameters has been done

analyzing the obtained results using different values for these parameters. The obtained values and results are conditioned mainly by the specific light conditions (of the lab) and the capturing distance. In case of changing these conditions, the parameter adjustment should be redone.

Stops concept

Exposure stops, or just stops, is a common term in photography and is used to describe how much light hits the sensor, or the brightness level of the image, relative to a reference level (sometimes referred to as "0 stops"). The brightness of the image is doubled when you "move one stop up" and reduced by half if you "move one stop down". The total amount of stops a camera has at its disposal is therefore equivalent to the sum of how many times the light intensity can be doubled from lowest to highest possible. The amount of stops available in a sensor is very relatable to the dynamic range of the camera. The Zivid camera has about 23 stops available, which is a crucial component as to why Zivid is able to achieve good data on shiny objects. [44]

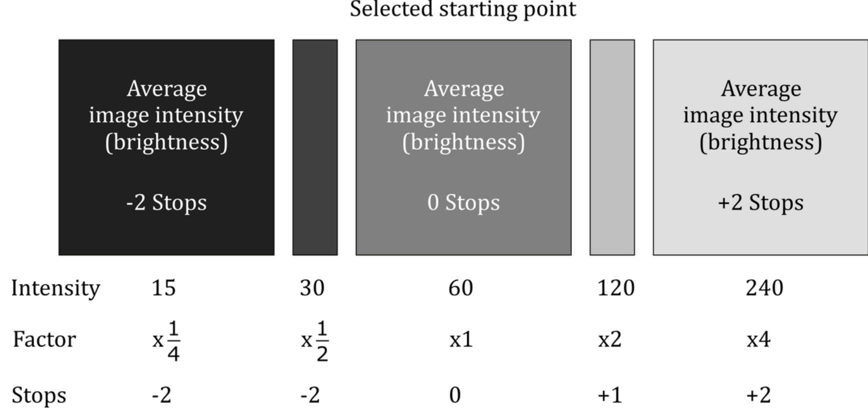


Figure 5.1: Stops concept

Exposure time

The exposure time - also known as shutter speed - is the amount of time that a single camera image is exposed to light, or how long the shutter remains open. A Zivid camera takes multiple images in order to calculate depth, and the exposure time of an individual image is defined in microseconds from 6 500 to 100.000. When exposure of 6.500 μs is chosen, the Zivid camera needs about 80 ms to capture a 3D image. Increasing the exposure time by some ratio increases the exposure of the image by the same amount.

Taking images means sampling light information in a certain time window. Many problems can arise from the ambient light. Specially for structured light cameras, the frequency of the AC power source where the ambient light sources are connected can have a negative influence to the results, specially in point clouds. Thus, the exposure time of the sampling has been selected taking into account the frequency of the AC source. By choosing an exposure time that satisfies the equation below (where n is a positive integer and f is the frequency of the light source), it is possible to filter out noise that varies with time from an ambient light source:

$$t_{exp} = \frac{n}{2f_s}, n = |Z| \tag{5.1}$$

Thus, it is recommended to use sampling rates in the multiples of 10.000 μs in countries that have a 50 Hz power line (Table 5.1), such as Europe, and 8.333 μs in countries with 60 Hz power line, such as the US.

Exposure time (Ms)	6500	10000	20000	40000	80000	100000
Capture time (ms)	90	130	250	490	970	1210
Stops	-0.62	0	+1	+2	+3	+3.32

Table 5.1

Iris aperture

The camera aperture is the opening of the lens window. Some cameras come with fixed aperture, while others are adjustable. The aperture is typically described by f-numbers N (see equation 5.1), where f is the focal length and D is the diameter of the entrance pupil.

$$N = \frac{f}{D} \tag{5.2}$$

Modern lenses use standarized f-number: f/1, f/1.4, f/2, f/2.8, f/4, f/5.6, f/8, f/11, f/16, f/22, f/32, etc.. Stepping up or down this series corresponds to adding or substracting one stop.

In this case, the zivid camera has an integrated mechanical iris. The iris parameter can be modified in a range from 0 to 72, where 0 is a completely closed iris and 72 completely open iris. The correspondece between the iris and the f-number is the following:

f -number	$f/1.4$	$f/2$	$f/2.8$	$f/4$	$f/5.6$	$f/8$	$f/11$	$f/16$	$f/22$	$f/32$
Iris	72	46	35	27	21	17	14	12	10	8
Stops	+4	+3	+2	+1	0	-1	-2	-3	-4	-5
Focus, near (mm)	925	900	850	800	750	650	600	500	425	350
Focus, far (mm)	1100	1150	1200	1300	1500	1900	2800	13000	Inf	Inf
Depth of Field (mm)	175	250	350	500	750	1250	2200	12500	Inf	Inf

As it is specified in the table above, the aperture of the iris depends on the distance of the object that is deemed to be focus. To explain that, it is important to clarify the concept circle of confusion (CoC). In optics, a circle of confusion is an optical spot caused by a cone of light rays from a lens not coming to a perfect focus when imaging a point source (see Figure 5.2). The modification of the aperture of the iris allows to focus objects in a limited boundary range (Focus, near and far in the table).

Projector brightness

The amount of light emitted by the Zivid camera projector is controlled by the parameter projector brightness. The range of the parameter is 0 when the projector is completely stopped and 1.8 when at the maximum output power mode. Maximizing projector brightness maximize the signal amplitude of the camera which in turn minimizes impact from noise as long as the reflected light from the projector does not over-saturate the pixel. So, maximize the projector brightness is a good way to maximize signal-to-noise ratio (SNR).

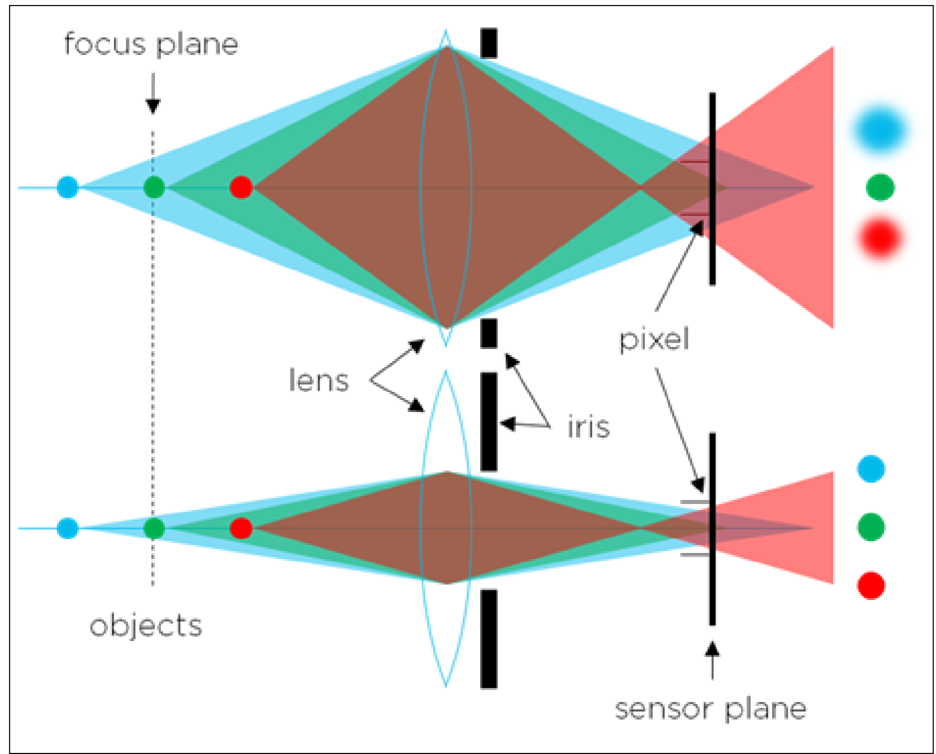


Figure 5.2: Circle of confusion (CoC) concept

The modification of the brightness parameter affects to stops as well . Besides, when the projector brightness is higher than 1.0, a thermal safety system limits the maximum duty cycle to ensure the thermal balance of the camera.

Projector Brightness	0.25	0.50	1.00	1.80
Stops	-2	-1	0	+0.85
Lumen	100	200	400	720

Gain

'Gain in a digital imaging device represents the relationship between the number of electrons acquired on an image sensor and the analog-to-digital units (ADUs) that are generated, representing the image signal. Increasing the gain amplifies the signal by increasing the ratio of ADUs to electrons acquired on the sensor. The result is that increasing gain increases the apparent brightness of an image at a given exposure.'

Higher gain values help to have better results in images with dark regions and dark objects. Anyways, while the gain is increased, the noise in the 3D point cloud is icreased as well.

Gain	1x	2x	4x	8x	16x
Stops	0	+1	+2	+3	+4
dB	-6	0	6	12	18

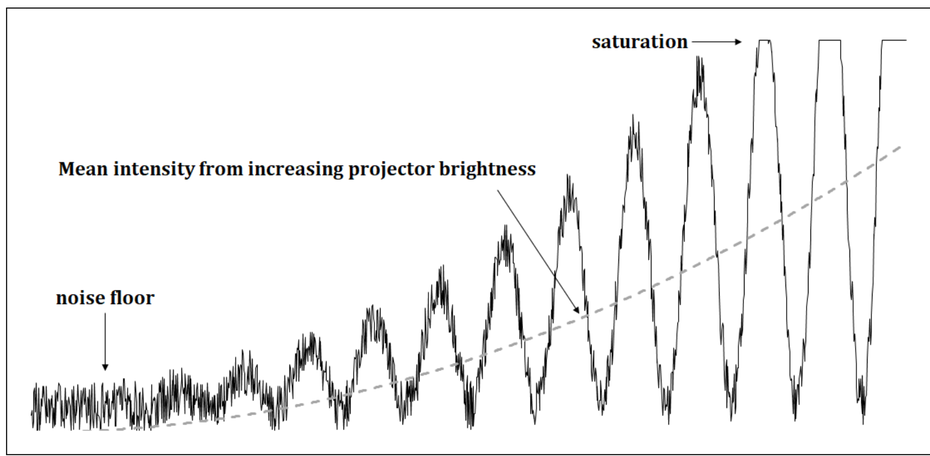


Figure 5.3: Noise modifying the projector brightness

Camera parameters set up

To realize the camera set up it has been taken into account the stops method. In this case, there are two unchangeable conditions: The capture distance with an approximated value of 1.7 m and the location of the lab (Grimstad, Norway). Thus, these conditions have a direct effect in two parameters:

- **Iris aperture**: The distance is between the boundary values [650 mm and 1900 mm], so the iris aperture should be 17 to have the image correctly focused.
- Due to the project location (Norway with a 50Hz power line), **the exposure time** should be a multiple of 10.000.

Thus, starting from these conditions, different pictures using different values for the exposure time, the projector brightness and the gain have been done and analysed in order to define the optimal values.

- Results:

First test

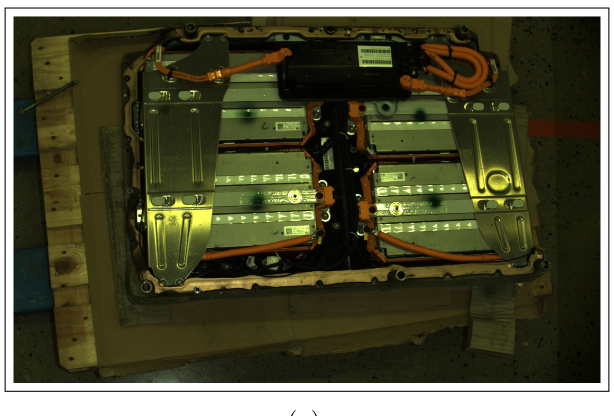
Parameters:	Iris Aperture	Exposure time	projector brightness	Gain	Laboratory light	Total Stops
Value	17	20.000 μs	1.0	1.0	Turned on	-
Stops	-1 Stops	+1 Stops	+0 Stops	+0 Stops	-	+0 Stops

As it can be observed in figure 5.4a and in the histogram 5.4b, the image is dark (colors placed in the left side of the histogram), thus in the next test the stops should be increased.

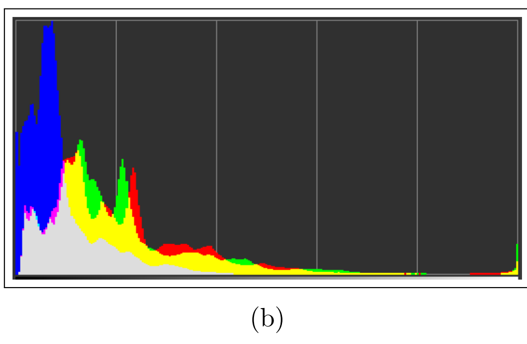
Apart from that, some reflections can also be observed. This reflections are caused by the material or object reflectivity combined with the ambient light and the projector light. Reflections can cause problems in the 2D image detections and problems in the definition of the 3D point cloud as well.

Second test

At this point, given the dark images obtained in the first test the Stops should be increased. There are three ways to increase the Stops: increasing the exposure time, the gain, or the projector brightness.



(a)



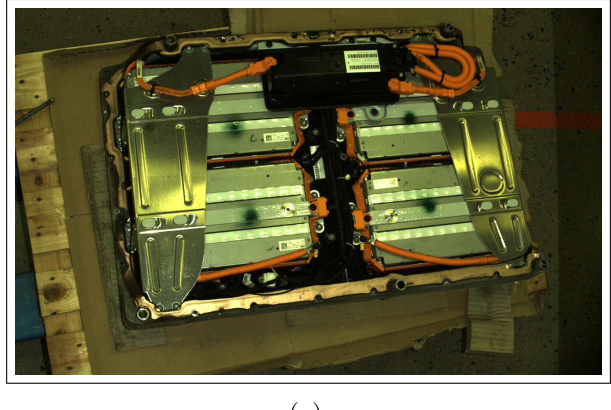
(b)

Figure 5.4: Image (a) and image histogram (b)

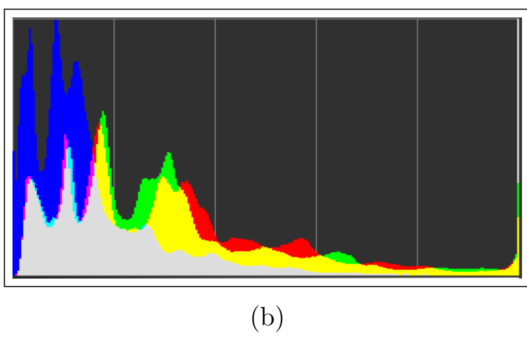
Thus, the projector brightness has been increased (from 1.0 to 1.8).

Analysing the brightness of the images obtained in this second test, the results are better than in the first test. The histogram (see Figure 5.6b) has moved a bit to the right. But the reflections have increased. Moreover, the definition and quality in the dark regions are very poor (see Figure 5.6a).

Parameters:	Iris Aperture	Exposure time	projector brightness	Gain	Laboratory light	Total Stops
Value	17	20.000 μs	1.8	1.0	Turned on	-
Stops	-1 Stops	+1 Stops	+0.85 Stops	+0 Stops	-	+0.85 Stops



(a)



(b)

Figure 5.6: Image (a) and image histogram (b)

Third test

Given the reflectivity problems observed in the second test, the laboratory light was turned off and the projector brightness was decreased from 1.8 to 1.0 again. So, with less light and still having the problem with dark regions, for the third test, the exposure time and the gain were

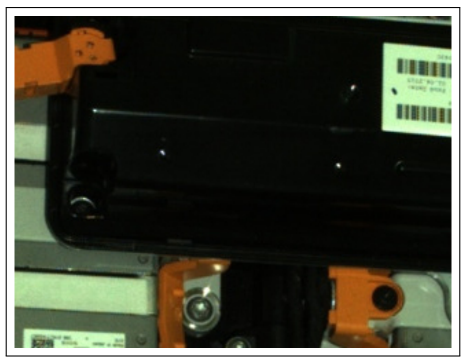
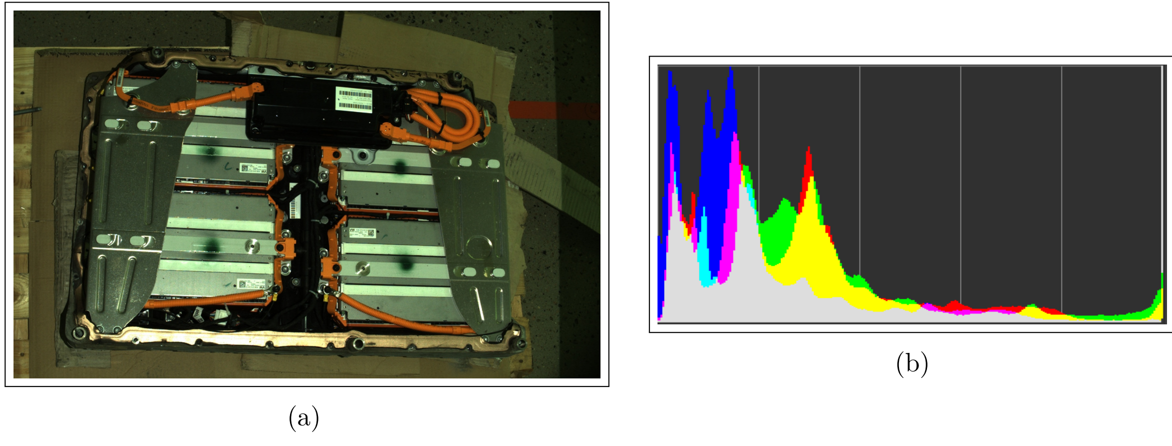


Figure 5.7: Dark region detail

increased. In this case the exposure from 20.000 μs to 50.000 μs and the gain from 1.0 to 4.0.

The results with this new configuration are better, the reflections are eliminated and the black regions have better results (see figure 5.8). But the image is still too dark.

Parameters:	Iris Aperture	Exposure time	projector brightness	Gain	Laboratory light	Total Stops
Value	17	50.000 μs	1.0	4.0	Turned off	-
Stops	-1 Stops	(aprox) +2.25 Stops	+2 Stops	+0 Stops	-	+3.25 Stops



(a)

(b)

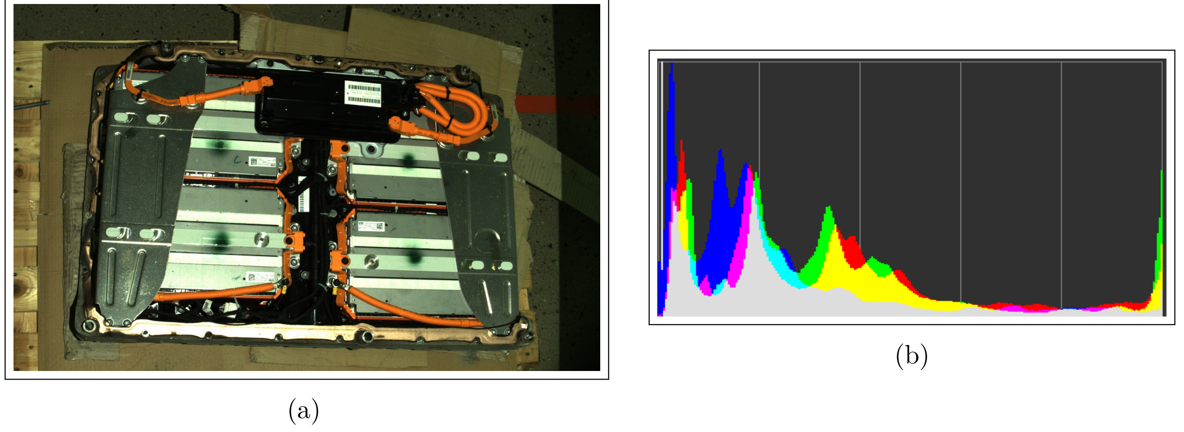
Figure 5.8: Image (a) and image histogram (b)

Fourth test

After the previous results, the last test aims to increase the brightness of the image. So, the exposure time was increased from 50.000 μs to 60.000 μs.

With these parameters, the image seems to be more balanced, the dark regions are well defined and there are no reflectivities.

Parameters:	Iris Aperture	Exposure time	projector brightness	Gain	Laboratory light	Total Stops
Value	17	60.000 μs	1.0	4.0	Turned off	-
Stops	-1 Stops	(aprox) +2.5 Stops	+2 Stops	+0 Stops	-	+3.25 Stops



(a)

(b)

Figure 5.9: Image (a) and image histogram (b)

5.2 Object detection: YOLOv3

When talking about object detection, it is important to clarify the concepts of image classification and object localisation. Image classification, refers to a computer vision process that is able to classify an image according to its visual content. Then, object localisation allows to detect the specific position of the object in the image (see Figure 5.10).

Object detection, basically provides the tools for finding all the objects (in this case the components) in one image, and drawing the bounding boxes around them. Finally, image segmentation is able to define the contours of the detected objects.

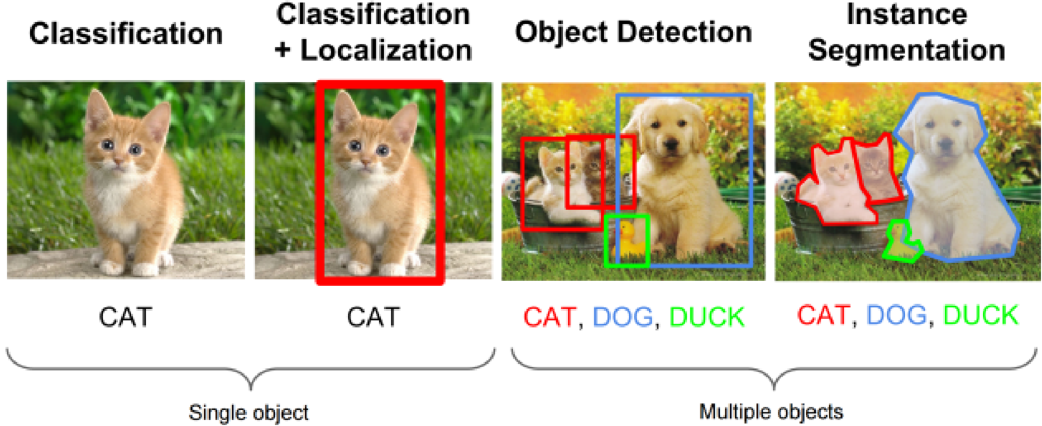


Figure 5.10: Object detection basic concepts overview

Given that the final purpose of the project is to implement the solution to the industry, algorithms based on segmentation have been discarded. This decision is based on that the creation of the training database for this type of algorithms is more complicated. This results on larger procedures to add new models into the known database.

When analysing object detection algorithms, they can be generally classified into two groups:

- **Algorithms based on classification:** Implemented in two stages, first, they select the regions of interest (ROI) in the image. Then, they classify this regions using convolutional neural networks (CNN). Since, they have to find the ROI of the image, the solution can be slow.

  One of the most representative algorithms of this type are the Region-based Convolutional Neural Network algorithm (RCNN, Fast-RCNN, Faster-RCNN and Mask-RCNN) and the RetinaNet algorithm.

- **Algorithms based on regression:** Instead of predicting the Regions of Interest of the image, this type of algorithms predict classes and bounding boxes for the images in one run.

  Within the well-known algorithms of this group, there are the You Only Look Once (YOLO) family algorithms and SSD (single Shot Multibox Detector).

YOLOv3.

In this project, the You Only Look Once YOLOv3 has been selected as the main algorithm for the object detection part.

YOLO is a real-time object detection algorithm. It is based on a convolutional neural network, and as explained before, it is an algorithm based on regression. Thus, the algorithm applies a single neural network to the full image.

The image is divided into smaller regions on which the bounding boxes (and the probabilities) are predicted for each region. These predicted probabilities weight the bounding boxes [41].

YOLOv1 was the first version of the algorithm. This version had 26 layers in total, with 24 Convolution Layers followed by 2 Fully Connected layers, but the main problem with YOLOv1 was its incapacity to detect tiny objects. Then, in December 2016, the paper 'YOLO 9000: Better, Faster, Stronger' by Redmon and Farhadi was released. A lot of improvements were included in the second version of the algorithm [22].

Finally, in April 2018 the same researchers published **'YOLOv3: An Incremental Improvement'**, with code available on a GitHub repository. This final version no longer copes with small objects and runs significantly faster than other detection methods [23] (see Figure 5.11).

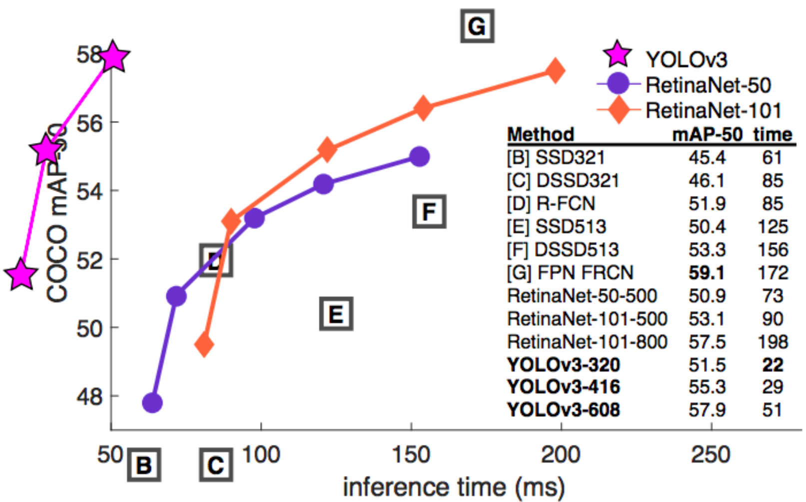


Figure 5.11: YOLO Performances [23]

YOLO functioning

The algorithm only requires one forward propagation pass through the neural network to obtain the predictions. Then, to detect only one time each object, a non-max suppression is done. In that way, the algorithm returns the detected objects with the bounding boxes and the probabilities.

Using YOLO algorithm, an individual CNN is able to predict multiple bounding boxes and the corresponding probabilities for them. YOLO trains on full images and directly optimizes detection performance (see Figure 5.12).

The system uses dimension clusters as anchor boxes [1] to predict the bounding boxes. Four coordinates (tx, ty, tw and th) are predicted for each bounding box. Therefore, if the previous bounding box has width and height (pw and ph) and the cell is displaced by (cx and cy) from the top-left

[1] '**Anchor boxes** are a set of predefined bounding boxes of a certain height and width. These boxes are defined to capture the scale and aspect ratio of specific object classes you want to detect and are typically chosen based on object sizes in your training datasets' [4]

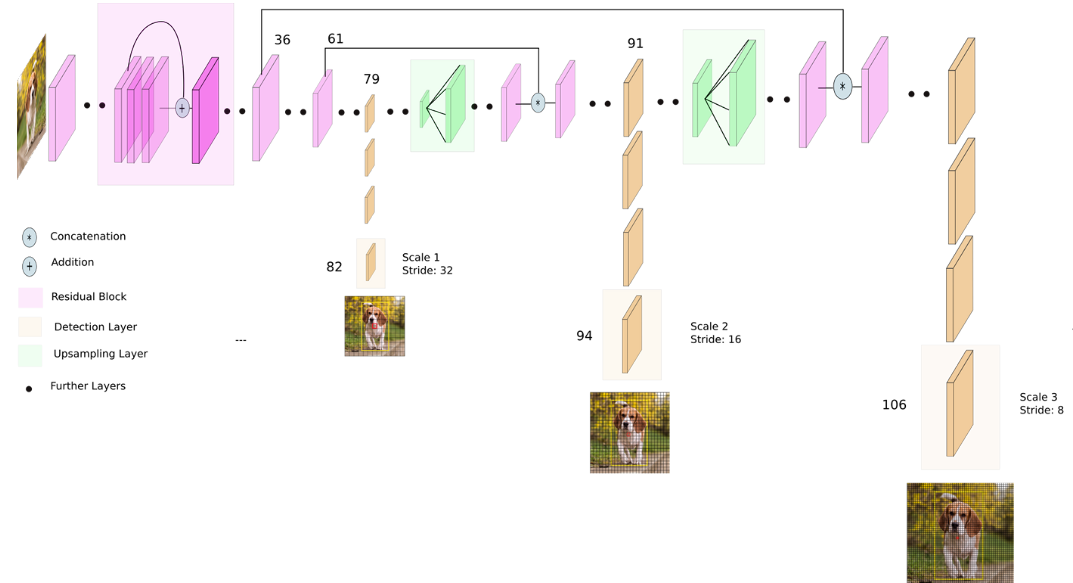


Figure 5.12: YOLO Architecture scheme

image corner, see Figure 5.13 [23]. The predictions are:

$$
\begin{aligned}
b_x &= \sigma\left(t_x\right) + c_x \\
b_y &= \sigma\left(t_y\right) + c_y \\
b_w &= p_w e^{t_w} \\
b_h &= p_h e^{t_h}
\end{aligned}
\tag{5.3}
$$

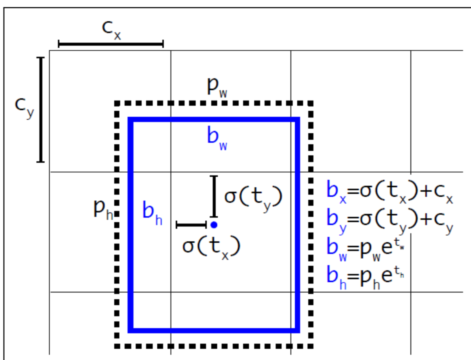


Figure 5.13: Predicted bounding boxes

Then, the algorithm predicts, using logistic regression, an objectiveness score for each bounding box. If the bounding box is overlapping the ground truth object [2], the score has a value of 1. If a bounding box is overlapping a ground truth object, but the score is not the best, it is ignored.

Later, to realize the class definition, the algorithm predicts the classes that a bounding box can contain using multi-label classification. A multi-label approach gives better results than other classifiers (i.e. softmax) for applications where there are many overlapping labels.

[2] **Ground truth** represents the desired output of an algorithm on an input

5.3 YOLOv3 Algorithm training

YOLOv3 is trained on full images with no hard negative mining. To train YOLOv3, it is used: multi-scale training, data augmentation and batch normalisation (within other standard procedures). The open-source neural network Darknet is used for training. Darknet is written in C and CUDA and supports CPU and GPU computation.

This section aims to explain the algorithm training procedure. Thus, the used terminology, the labelling, the images acquisition and its training stage have been explained. Moreover, the designed function *training_data_autogenerator()* has also been described here.

5.3.1 System's nomenclature

The detected components should have a concrete nomenclature in order to ease the access of the task planner to the database. Moreover, it is very important that the detection distinguish the different components, brands and versions.

1. **For the main components:**

   COMPONENT_BRAND_VERSION

2. **For the screws:**

   screw

3. **For the connection components:**

   connection_TYPE

5.3.2 Labeling

The main objective of labeling the pictures is to let the algorithm know where the the different components are located in the image. The algorithm takes the labeled images and extracts the features of the labeled areas in order to learn how the different objects are and how to detect them. To create the database for this project, it has been used the open-source program LabelImg (see Figure 5.14. It returns a text file in the defined YOLO format. In this format, the lines of the text file contain the information regarding each label. So, every line has the position of two corners of the label and then the class that it belongs to. [23]

In case that in future stages of the project an update of the database would be needed, for example to add new components or new LIB packs models, this labeling process should be repeated.

5.3.3 Training Images acquisition

In order to obtain the training images the script **training_data.py** (see the full code appendix B.13) has been created. This function aims to capture automatically images from different angles in order to obtain the training data easily.

Basically, the script runs a *for* loop iterating over a list that contains 4 different orientations for the TCP. For each orientation, another *for* loop takes twenty pictures with different position values (changing the y position within each picture).

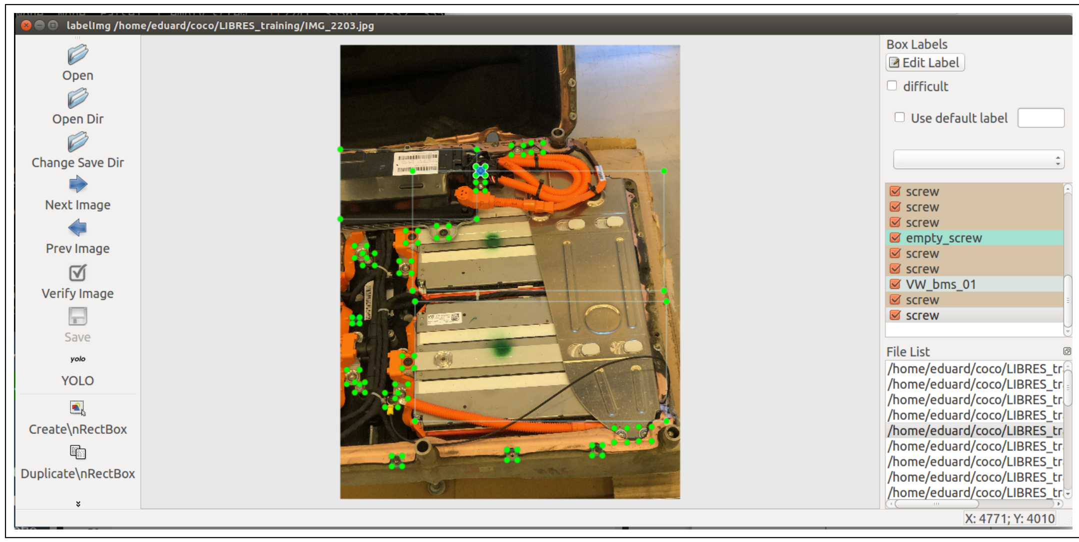


Figure 5.14: Labeling example (LabelImg window)

To have better training results, it is important to run the script several times taking pictures of the battery pack placed in different orientations.

```
for e in range(0,4):
        pose_goal.orientation.x = 0.0
        pose_goal.orientation.y = y[e]
        pose_goal.orientation.z = 0.0
        pose_goal.orientation.w = w[e]

        for i in range(0,20):
            pose_goal.position.y = pose_goal.position.y-0.02
            move_group.set_pose_target(pose_goal)
            plan = move_group.go(wait=True)
            # Calling `stop()` ensures that there is no residual movement
            move_group.stop()
            s.capture()
            s.im_num=s.im_num+1

        pose_goal.position.y = 7.20
        pose_goal.position.x = pose_goal.position.x+0.15
```

5.3.4 Training stage

The training of the algorithm has been done with the code train.py. The code is included in the official yolov3 open source code repositories.

The main arguments to run the training script are: A folder with all the images and text files with the label information and the LIBRES.data file.

- **Images and label information:**

  It has been created a folder containing all the images and the labels information. So, the images (.jpg) and the text (.txt) files are located in this folder. The name for the image and the text files should be the same.

- **LIBRES.data, LIBRES.names and LIBRES.txt files:**

  The main objective of the LIBRES.data file is to provide the path to access the necessary information to train the algorithm (the LIBRES_classes.names and the LIBRES.txt files).

  ———— LIBRES.data ————

  ```
  classes=5
  train=data/LIBRES.txt
  valid=data/LIBRES.txt
  names=data/LIBRES_classes.names
  ```

  On the one side, the LIBRES.names file contains all the names of the components. The names are saved following a specific order. That order corresponds to the order given in the labeling process.

  ———— LIBRES_classes.names ————

  ```
  VW_modules_01
  VW_bms_01
  Mitsubishi_modules_01
  screw
  empty_screw
  ...
  ```

  On the other side, the LIBRES.txt file contains the path to all the training images.

  ———— LIBRES.txt ————

  ```
  ...
  /.../ path_to_file/IMG_2413.jpg
  /.../ path_to_file/IMG_2261.jpg
  /.../ path_to_file/IMG_2259.jpg
  /.../ path_to_file/IMG_2211.jpg
  /.../ path_to_file/IMG_2200.jpg
  ...
  ```

- **Training process:**

  A total number of 89 images have been used to train the algorithm. The training process (see Figure 5.15 has lasted 1.431 hours.

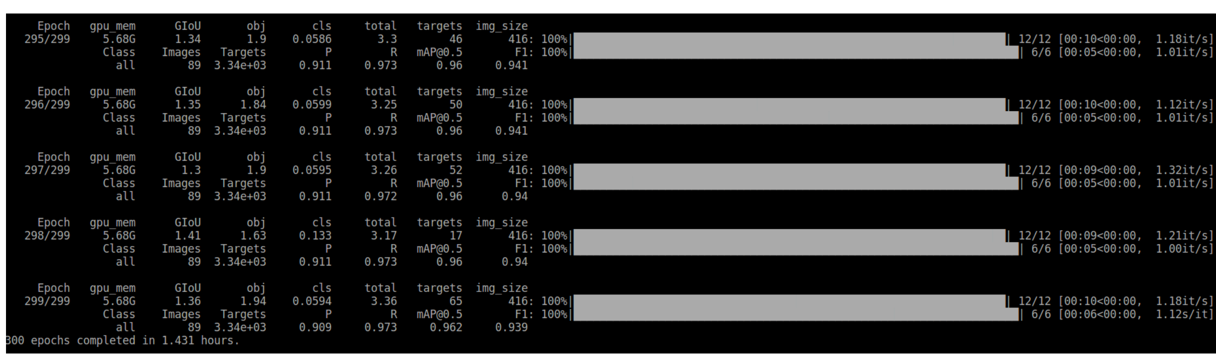


Figure 5.15: YOLOv3 training process

5.3.5 YOLOv3 training results

Every time that a training of the algorithm is realized, the algorithm returns the values different parameters along the training epochs (see Figure 5.16). "One Epoch is when an entire dataset is passed forward and backward through the neural network only once." [9]. Apart from detecting training problems, these results are mainly used to detect overtrainings.

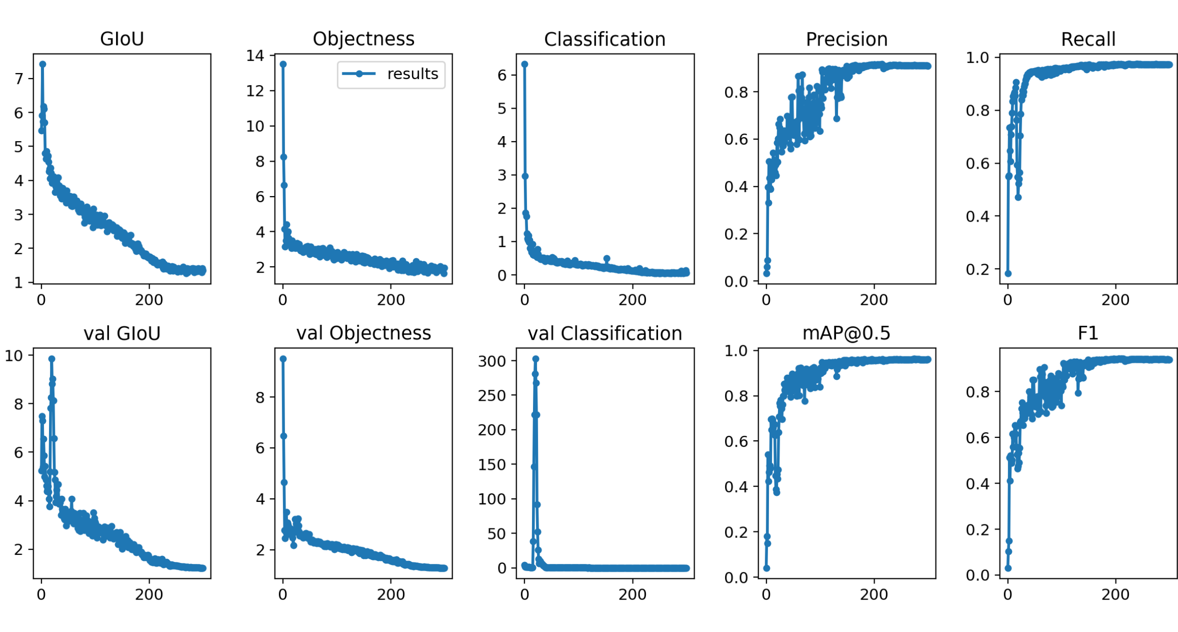


Figure 5.16: YOLOv3 obtained training results.

5.3.6 Training data creator (function)

The creation of a database is basic to have a good training of the algorithm. The more images used for training the algorithm the better detection results are obtained.

The aim of the function *training_data_autogenerator()* (see full code in appendix B.7), is to use the components localisation images taken (while the system is working) to create YOLO training data. Generally, the function converts the information obtained from the detection into valid training data for YOLO (image, .txt file and path).

To do this, first of all, the function reads and opens an image, and the corresponding .txt file with the results of the detection. Using a for loop, converts each line of the .txt file written in the default output format to the valid training data format. How the labels are defined is the main difference between both formats. From one side, in the training data format, the labels are defined by the central (x,y) position of the label and the label dimensions (height and width) expressed in parts per unit. From the other side, in the detection output format, the (x,y) positions of two opposite corners of the label are defined. Thus, for each component, the function finds dimensions and the center of the label, and using the image size, converts these values into parts per unit.

Finally, when the conversion has been done, these values are written in a .txt file, and the path of each file is added to the *autogenerated_data.txt* file to complete the training data generation.

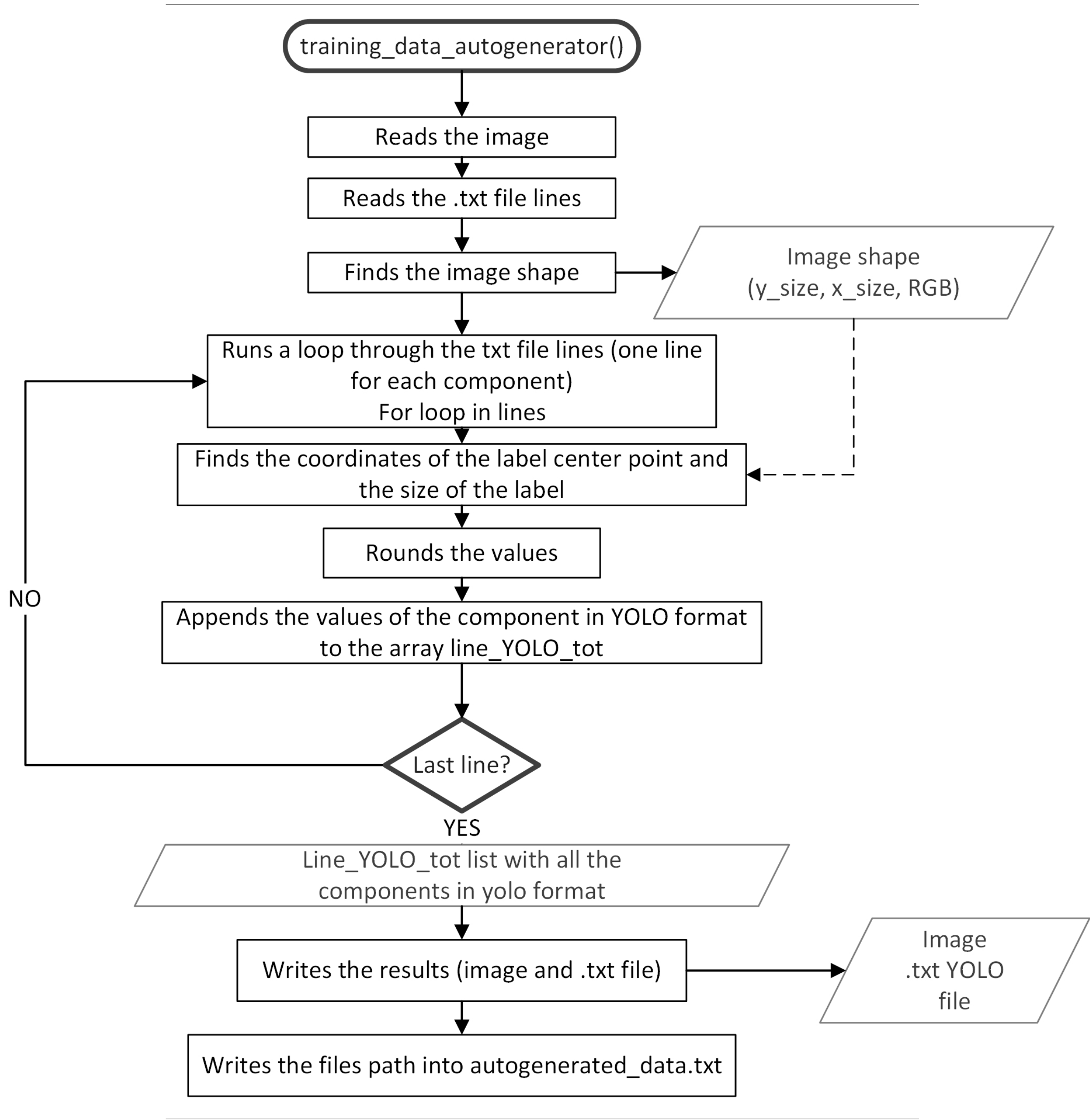


Figure 5.17: Task planner function training_data_autogenerator() flow chart

Chapter 6

Pose estimation

The pose estimation problem is to determine the pose in world reference base frame of the different components previously detected in the object detection stage. In general, pose estimation includes the determination of three translations and three rotations. However, in this study, pose estimation is limited to three translations and a fixed rotation.

Thus, after object detection, presented in chapter 5, the next step is to relate the image coordinates of the components with the 3D point cloud information. Which results in the component positions in the world reference base frame. This is necessary information to move the robot to the desired positions and realise the dismantling.

This chapter presents the project's camera and 3D datasets, it explains how this 3D information is used and matched with the information extracted from object detection to determine the positions of the different components.

6.1 Structural light camera: Zivid one 3D camera

Zivid One 3D camera has been assigned for this project. Zivid is a market-leading provider of 3D machine vision cameras and software established in Norway. Its Zivid One and Zivid One Plus products are regarded as the world's most accurate real-time 3D color cameras and bring a human-like vision to the smart factories and warehouses of Industry 4.0.

Figure 6.1: Zivid One

Some of the most relevant Zivid One M specifications are showed in the table below (see more

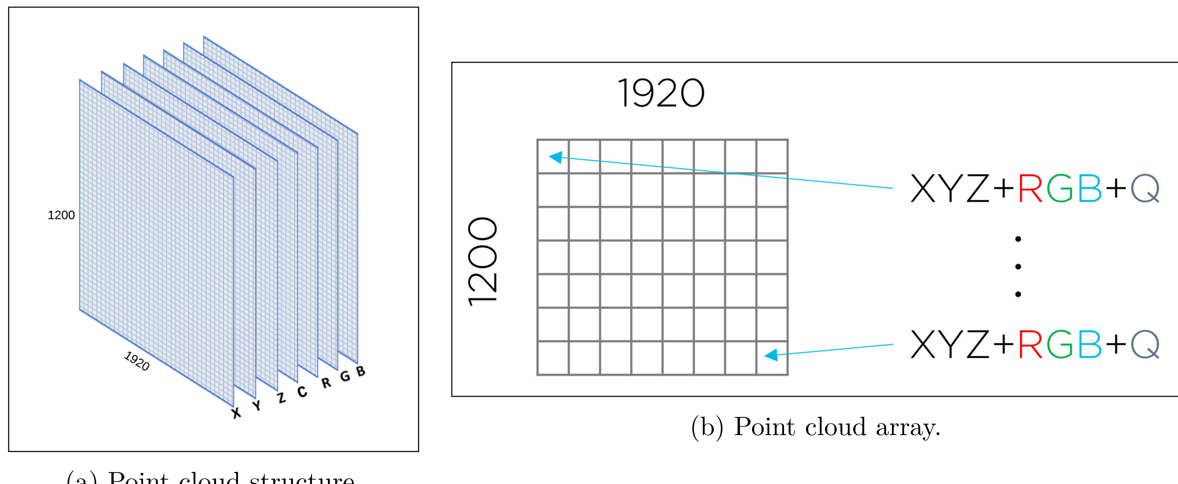
specifications in appendix A):

SPECIFICATIONS	
Precision	0.1 mm
Acquisition rate	10 Hz
Output	3D+RGB
Imaging	3D HDR

6.1.1 Pointcloud concept

Point clouds represent the X, Y, and Z geometric coordinates of a sampled scene. Point clouds are a means of assembling a massive quantity of single spatial measurements into a dataset.

In this project, the PointClouds have been obtained using the camera Zivid One. Zivid one is an structural light camera and its output is an organized point cloud. This means that the point cloud is organized as a 2D array of points that features an image like structure. The correlation between pixels in 2D images and the 3D points in ordered point clouds is of 1:1. Thus, the results of the 2D operations and algorithms can be directly obtained in the 3D point cloud. Given that the camera has 2.3 MP (1920 x1200), and because of the 1:1 correlation, the created point cloud consists of 2.3 milion points.



(a) Point cloud structure.

(b) Point cloud array.

Figure 6.2: Point cloud

6.1.2 Structured Light Imaging

In this type of 3D cameras, the point cloud is obtained modulating patterns of light on the scene and capturing their reflection with a 2D imaging camera.

Thus, a known pattern is projected onto the surface, in this case the battery pack. When the camera views the pattern from a different prespective, the surface features of the battery pack distort the pattern. Analysing these distortions, the surface topography can be reconstructed.

In this case, the Zivid camera contains a projector and a camera. The camera and the projector are placed in different distances forming an specific angle. The depth information is obtained dividing the stripe displacement in each point of the image (see Figure 6.3). In the specific case of the Zivid

one, for every frame, several images are captured at an acquisition rate of 10 Hz in order to increase the accuracy of the captured point cloud.

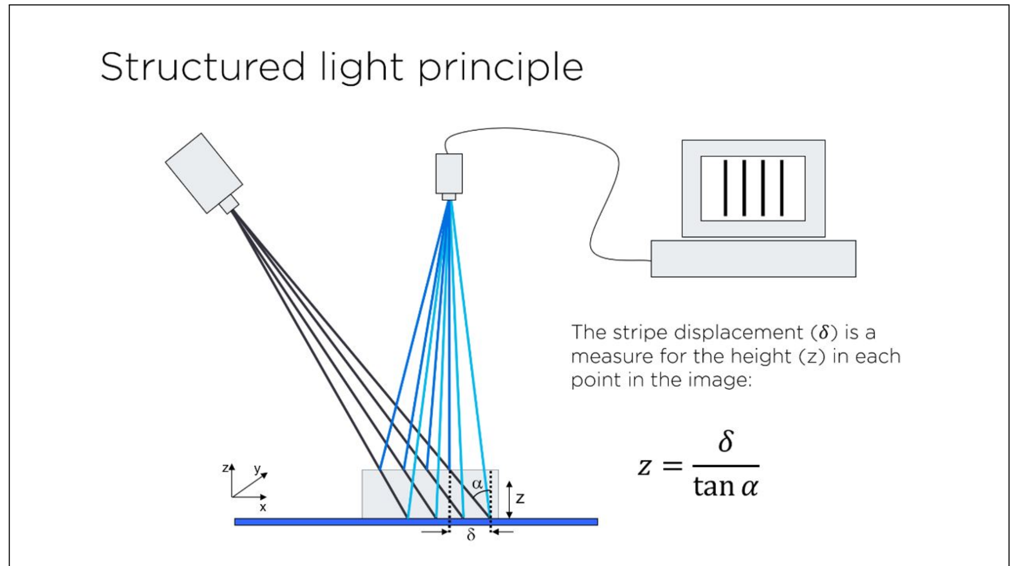


Figure 6.3: Structured light imaging

6.2 Point cloud and depth images storage

The creation of the point cloud files is one of the basics of this project. It has been used for:

- In the task planner, to save the point clouds used for the detection and the pose estimation.
- For the extrinsic calibration. To save the point clouds used by the Zivid CLI calibration tool.
- In initial stages of the project, in order to understand the pointclouds.

Capture service

As explained before, the integration of the different system elements (camera, computer and robot) has been done through ROS. Thus, first of all when the computer wants to indicate to the camera that a capture is needed, the ROS capture.srv service has to be invoked.

This service is invoked to trigger a 3D capture. The resulting point cloud is published into the topic */zivid_camera/points* and the depth image is published into the topic */zivid_camera/depth/image_raw*. Moreover, the 2D color image is published into the topic *color/image_color* .

From the one side, to save the Pointcloud2 message, it should be converted into any of the valid point cloud formats (.zdf, .pcl, .ply, .pcd, etc..). Given its ability to store and process point cloud dataset and its capacity of storing different data types, in this project, the .pcd format has been selected [36].

From the other side, the chosen format for saving the depth image dataset has been the Numpy file format (.npy). This format is the fastest to store points (thus, reduces the system timings) and it eases the access of the task planner to the depth image [40].

To save the Pointcloud2 message and the depth image, a ROS subscriber has been created. This subscriber, receive the messages and launches the function callback2 (to save and transform the Pointcloud2 message to .pcl data) and callback3 (to save and transform the depth message to .npy data).

```
self.Pointcloud_sub = rospy.Subscriber('/zivid_camera/points', PointCloud2, self.callback2)
```

To do so, the pcl library should be installed:

```
~$ pip3 install python-pcl
```

```
def callback2(self,ros_cloud)
    points_list = [ ]
    for data in point_cloud2.read_points(ros_cloud, skip_nans=True):
        points_list.append([data[0], data[1], data[2], data[3]])
    pcl_data = pcl.PointCloud_PointXYZRGB()
    pcl_data.from_list(points_list)
    pc_num= str(self.im_num)+'.pcd'
    rospy.loginfo(pc_num)
    pcl_data.to_file('/home/eduard/LIBRES_SYS/yolov3/data/samples/'+pc_num)
```

```
def callback3(self,data):
    try:
        NewImg = self.bridge.imgmsg_to_cv2(data, "passthrough")
        depth_array = np.array(NewImg, dtype=np.float32)
        directory_im2 = r'/home/eduard/LIBRES_SYS/yolov3/data/depth/'
        im_name= 'depth'+str(self.im_num)+'.npy'
        np.save(os.path.join(directory_im2, im_name), depth_array )
        cv2.waitKey(100)
        print('npy saved')
    except CvBridgeError as e:
        print(e)

    cv2.destroyAllWindows()
    rospy.loginfo( 'depth image saved')
```

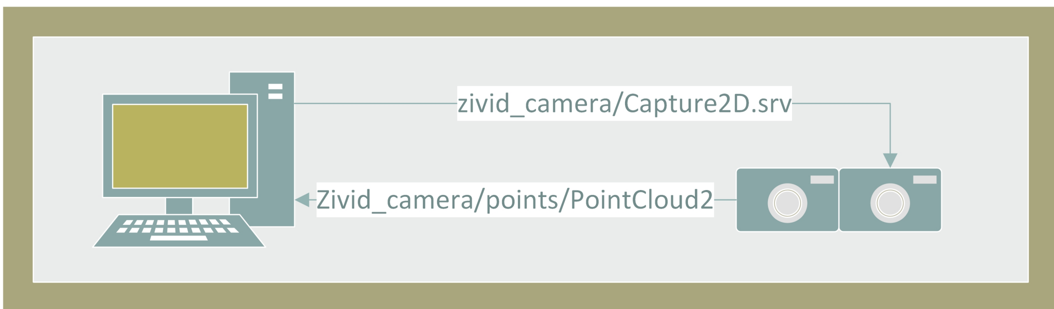


Figure 6.4: Point cloud saving flow chart

Point cloud data visualization

There are different ways to visualise these saved point clouds. One is using the pcl_viewer. This viewer can easily be installed in Ubuntu running:

```
~$ sudo apt-get install -y pcl-tools
```

Thus, to visualize the point cloud run (substituting the point cloud path):

```
~$ pcl_viewer -multiview 1 "point_cloud_path".pcd
```

The obtained output is showed in Figure 6.5.

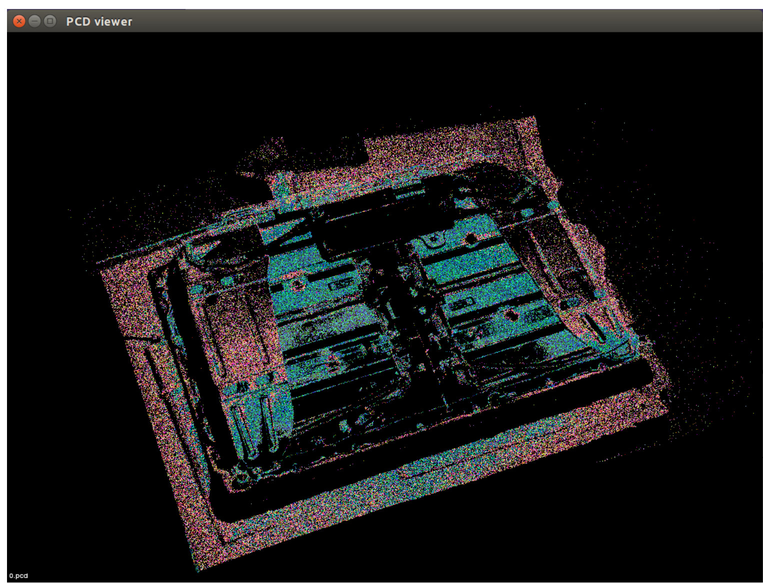


Figure 6.5: pcl_viewer

6.3 Intrinsic and Extrinsic Parameters

Geometric camera calibtation refers to the procedure of estimating the parameters and image sensor of a camera. These parameters can be used to correct lens distortion, find objects in world units, determine the position of the camera, etc..

Thus, to find the camera parameters means to find the intrinsics, extrinsics and distortion coefficients.

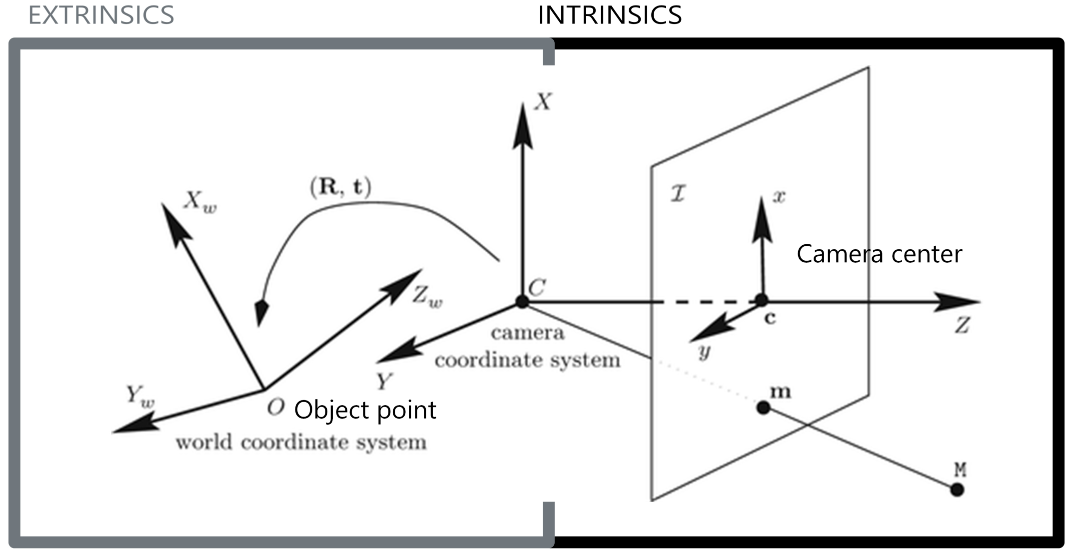


Figure 6.6: Extrinsic and intrinsic parameters concept

6.3.1 Pinhole Camera Model

The pinhole camera model determines the relationship between the coordinates of a point placed in the world space (P(X,Y,Z) in Figure 6.7) and its projection onto the image plane [14].

For ideal pinhole cameras, the camera aperture is described as a point and no lenses are used to focus the light. This cameras (called perspective cameras), can be represented by a direct perspective transformation. This transformation does not consider many effects of a real camera such as distortions, blurring, unfocused objects, etc...

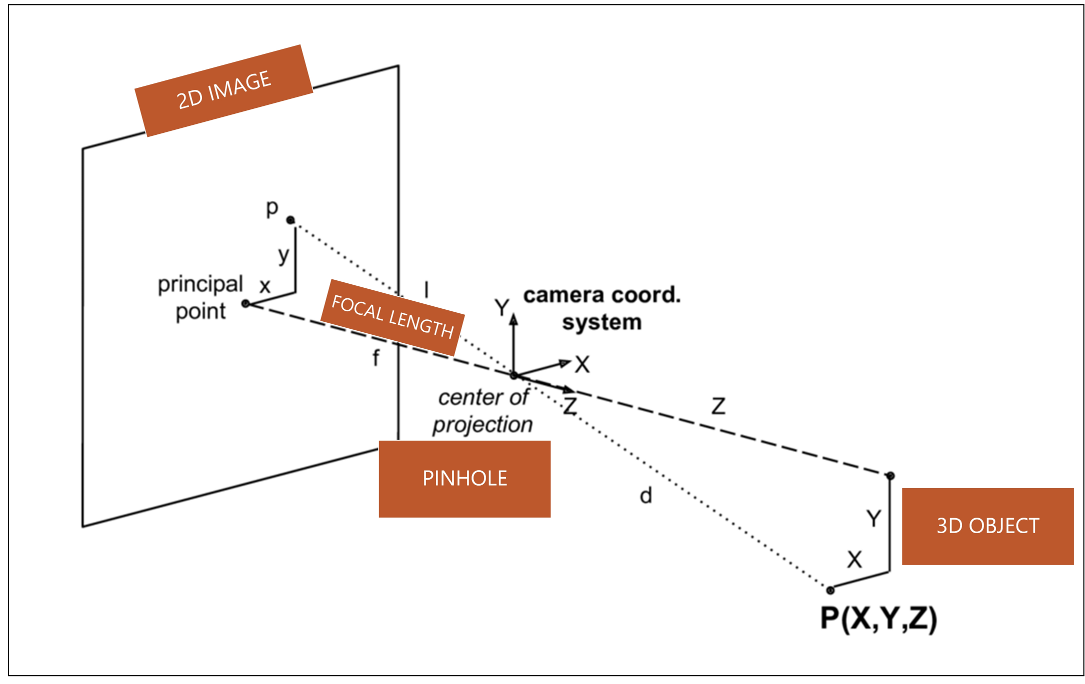


Figure 6.7: Pinhole Camera Model

Geometry and mathematics of the pinhole projection

The aim of the pinhole equations is to compute the intersection of a light ray coming from a point placed in the world (P=(x,y,z)) with the image plane. [33]

Basic perspective projection: Deriving using similar triangles, the basic equations (6.1 and 6.2) can be obtained.

$$x = f\frac{X}{Z} \tag{6.1}$$

$$y = f\frac{Y}{Z} \tag{6.2}$$

So, from the equations (6.1 and 6.2), the homogeneous transformations can be found. Fundamentally, the homogeneous coordinates represent a 2D point (x,y) by a 3D point (x',y',z').

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{6.3}$$

6.3.2 Intrinsic Parameters

As explained before in the subsection 6.3.1, the intrinsic matrix transforms the 3D camera coordinates to 2D homogeneous image coordinates. The intrinsic parameters matrix is (where f is the focal length, s is skew coefficient, and c is the optical center):

$$K = \begin{pmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{6.4}$$

- Focal length, f_x and f_y: Is the distance between the pinhole and the image plane. The focal length is usually measured in pixels.
- Otpical center, c_x and c_y: it is the principal point in the Figure 6.7, in pixels.
- Axis skew, s: Is non-zero if the image axes are not perpendicular.

In this case, the values for the intrinsic parameters matrix (K) for the Zivid are:

$$K = \begin{pmatrix} 2765.132 & 0 & 946.901 \\ 0 & 2764.617 & 579.477 \\ 0 & 0 & 1 \end{pmatrix} \tag{6.5}$$

These parameters have been found in the *sensor_msgs/CameraInfo.msg* ROS message published in the topic *depth/Camera_info* (see ROS messages and ROS topics in section 3.1.2). The message contains the meta data information for the camera.

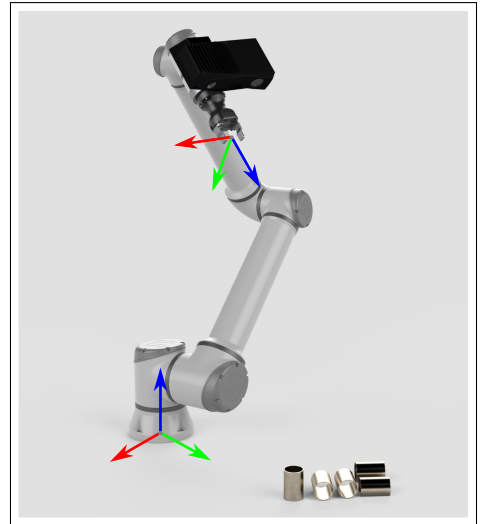
6.3.3 Extrinsic Parameters Calibration

The main objective of the extrinsic calibration of the camera is to calculate the necessary transform matrices in order to obtain the position of the object in the robot base coordinate system (extrinsic parameters).

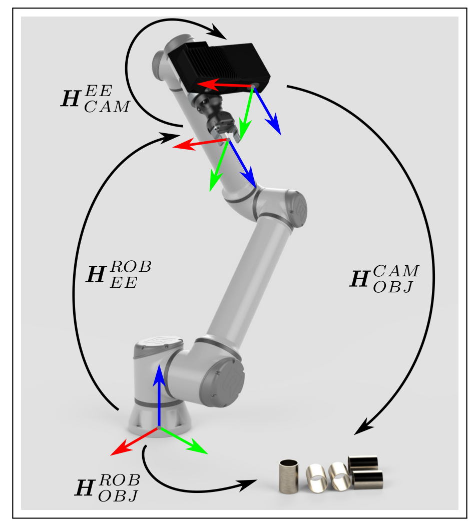
The relation

$$H_{OBJ}^{ROB} = H_{EE}^{ROB} * H_{CAM}^{EE} * H_{OBJ}^{CAM} \tag{6.6}$$

In this case, it has been used an eye in hand configuration, so the end effector of the robot is equipped with a camera. For that configuration, realize the camera calibration means calculating the transform matrix H_{CAM}^{EE} (see equation 6.6). Note that, in equation 6.6, H_{OBJ}^{ROB} is the transform matrix of the object from the robot base frame, H_{EE}^{ROB} is the transform of the end-effector from the robot base frame, H_{CAM}^{EE} is the transform of the camera in frame end-effector, and H_{OBJ}^{CAM} is the transform of the object from the camera base frame.



(a) Robot's and end-effector's coordinates systems.



(b) Plot of the transform matrices concept and robot's, end-effector's and camera's coordinates systems.

Figure 6.8: Eye-in-hand configuration

Thus, it has been taken different images of the checkerboard [1] (see Figure 6.11) from different known robot configurations. Then, to find the transform matrix (H_{CAM}^{EE}) it has been used the Zivid CLI tool. The arguments of the Zivid CLI tool are the image and the matching robot pose in a .yaml file. So, it has been created a python script in order to convert the different robot positions from quaternions to the correspondent opencv format.

Zivid 3D camera uses a special 9x6 gray/white checkerboard. For that reason, the calibration object used has been provided by the camera manufacturer, see Figure 6.11.

[1]The checker board is the object used for calibration.

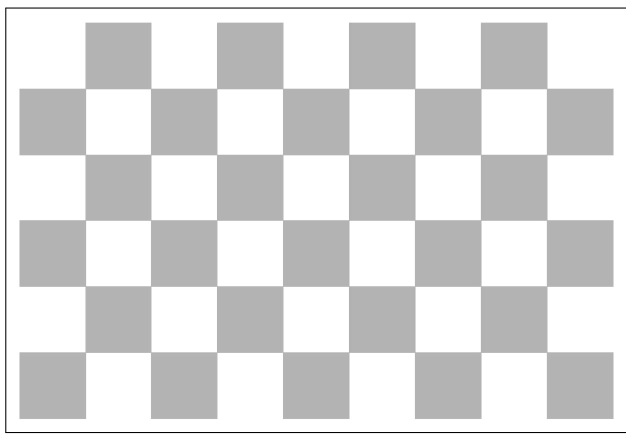


Figure 6.9: 9x6 gray/white checkerboard

Extrinsic calibration process

In this case, the Zivid camera extrinsic calibration has been done using the Zivid Command Line Interface (CLI) tool for Hand-Eye calibration. The main function of this tool is to obtain the transform matrix of the camera from the end-effector (H_{CAM}^{EE}).

To calculate this transform, the tool needs a dataset containing:

- The point cloud captures of the calibration checkerboard (Figure 6.11) from different positions and orientations (in .zdf format).
- The corresponding positions of the end-effector (in .yalm format, see appendix C.3).

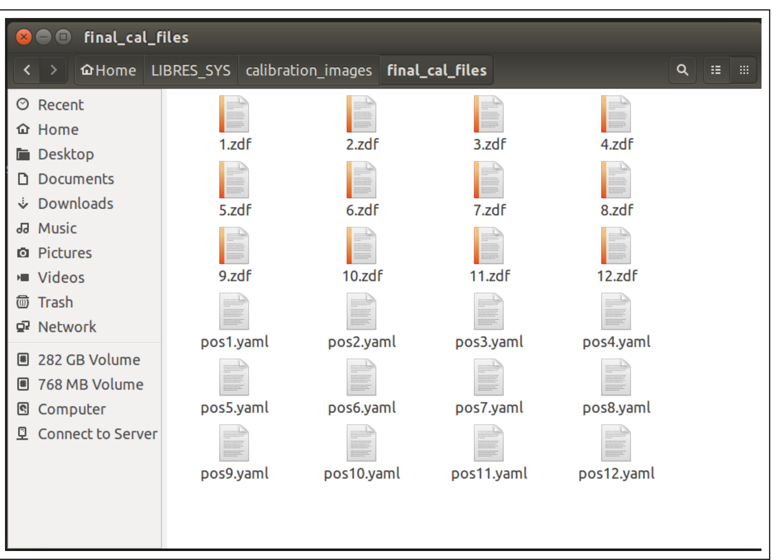


Figure 6.10: Calibration dataset (.zdf and .yalm files)

From the one side, n order to move the robot to the different positions the script move_to_cal.py has been created, see the full code in appendix B.11. It is essential to underline that the end-effector used to take the calibration point clouds is defined in a specific position, in case of using a different end-effector position the quaternions defined in the script might be invalid needing to redefine new valid poses. In the case of redefining the poses, the .yalm files should also be updated.

From the other side, to obtain the .yalm files quickly from the positions the obtain_cal_files.py has been created. The arguments for this script are two lists. The first one contains the x, y and z positions, and the second one contains the rotation in quaternions (q_x, q_y, q_z and q_w). With this information calculates the transform matrix and automatically creates all the .yalm files, see the full script in appendix B.14.

After creating all the necessary files, the next step has been opening the Zivid CLI tool for calibration.

```
C:\Users\>ZividExperimentalHandEyeCalibration.exe --h
SYNOPSIS
        ZividExperimentalHandEyeCalibration.exe [-h] [--eih] [--eth] [-n <max-
           images>] [-d <input-dir>] [--tf <transform-file>] [--rf <residuals-
           file>]

OPTIONS
        -h, --help help
        --eih, --eye-in-hand perform eye-in-hand calibration
        --eth, --eye-to-hand perform eye-to-hand calibration
        -n, --max-images <max-images> max number of input images (default: 100)
         -d, --input-dir <input-dir> input directory for point clouds and robot
            poses
        --tf, --transform-file <transform-file> save resulting hand eye transform
           to file
        --rf, --residuals-file <residuals-file> save resulting hand eye residuals
           to file

Please specify either --eye-in-hand or --eye-to-hand
```

Then, the path where .yalm and .zdf files are located has been provided. Moreover, the path where the calibrated transform and the residuals should be stored has also been provided.

```
C:\Users\>ZividExperimentalHandEyeCalibration.exe --eih -d "C:\Users\\Documents\
   final_cal_files" --tf "C:\Users\\Documents\final_cal_files\calibration
   transform.yaml" --rf "C:\Users\\Documents\final_cal_files\residual.yaml"
```

Finally, after running the previous command the finals results have been obtained. Appendix C.1. Thus, the final transform matrix is:

$$H_{CAM}^{EE} = \begin{bmatrix} -1.948419e-01 & 1.6311910e-02 & 9.806989e-01 & -1.040664e+03 \\ -9.808231e-01 & 1.594048e-03 & -1.948931e-01 & 2.8582285e+02, \\ -4.742362e-03 & -9.998656e-01 & 1.568851e-02 & 4.379912e+01 \\ 0. & 0. & 0. & 1. \end{bmatrix}$$

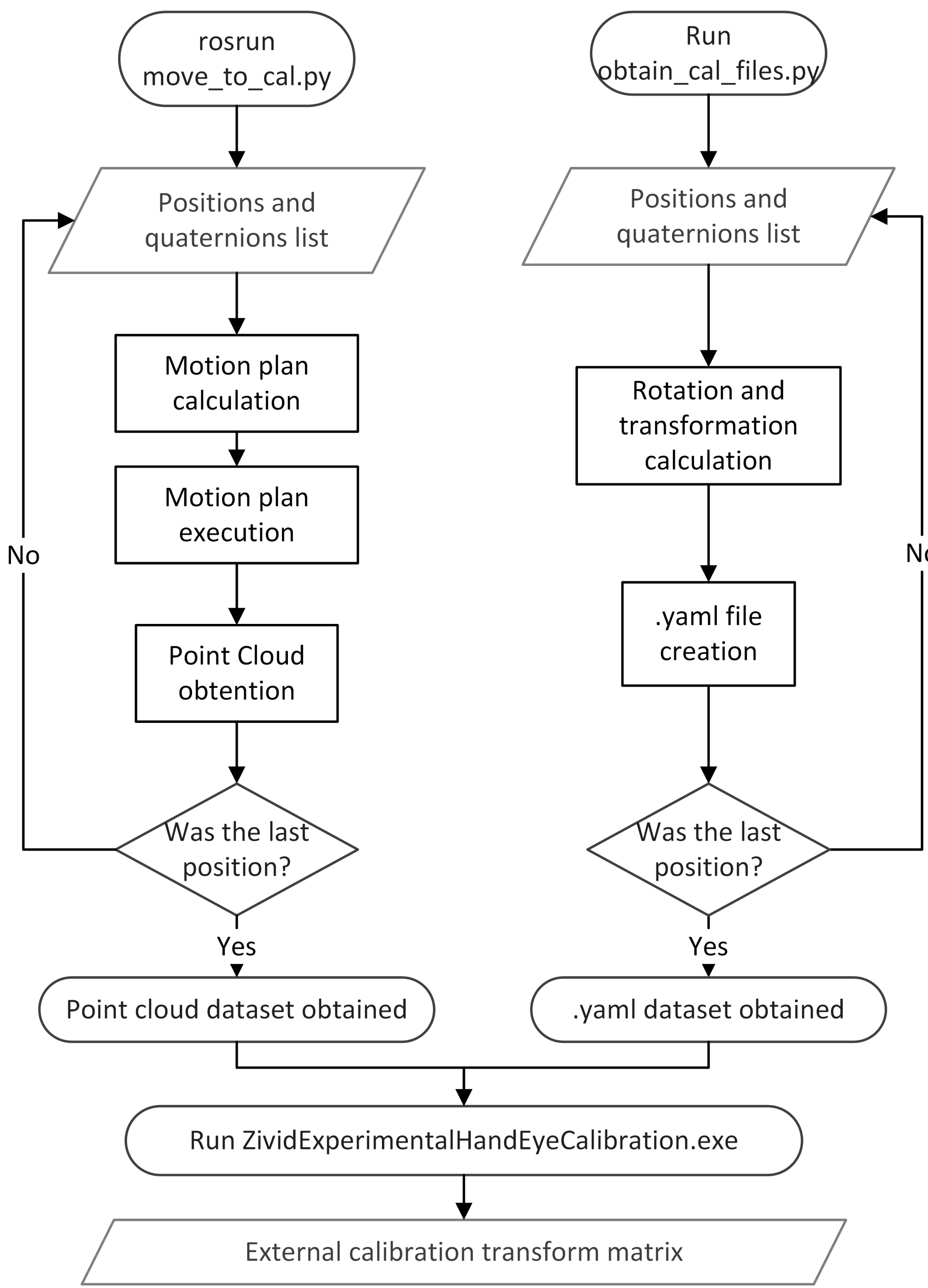


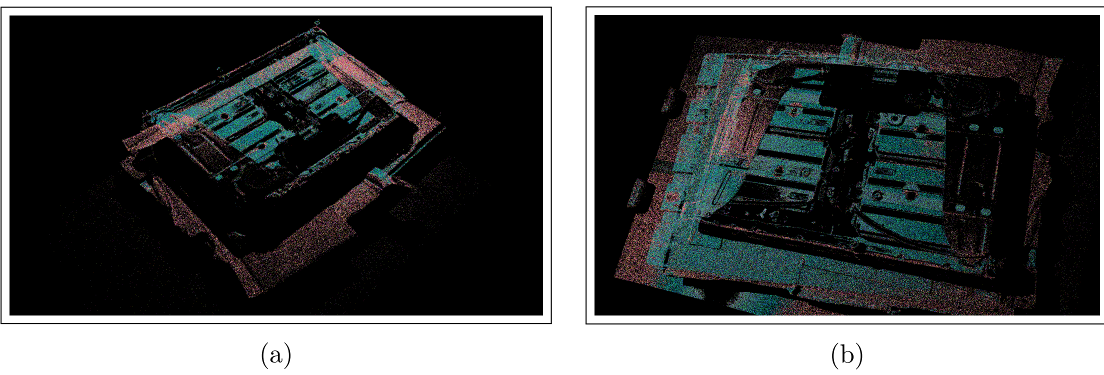
Figure 6.11: Extrinsic calibration flow chart

6.4 World coordinates of the detected components

In order to be able to move the robot to the positions where all the components are located, the world reference frame position of them should be obtained. To do so, the 2D image detection is used. As it is explained in chapter 5, the different components are detected and the positions of them in the coordinates of the image are obtained. When the detection is done, a .txt file containing all the positions is created. Each line of the .txt file contains the position of two corners of the

bounding box, inside which the component is located, the class [2] and the detection accuracy (see the example below).

x_1	y_1	x_2	y_2	class	accuracy
862	473	1314	669	1	0.796174



(a) (b)

Figure 6.12: Point cloud data example

A previous step before obtaining the world reference frame position of the components is to obtain the camera coordinates of the component. This means the x,y and z coordinates of the desired components in camera reference frame (see Figure 6.13). It is important to underline that the point cloud information is always in camera reference frame (see point cloud examples in figure 6.12). This is the main purpose of the function CamR_pos (full code in appendix B.2).

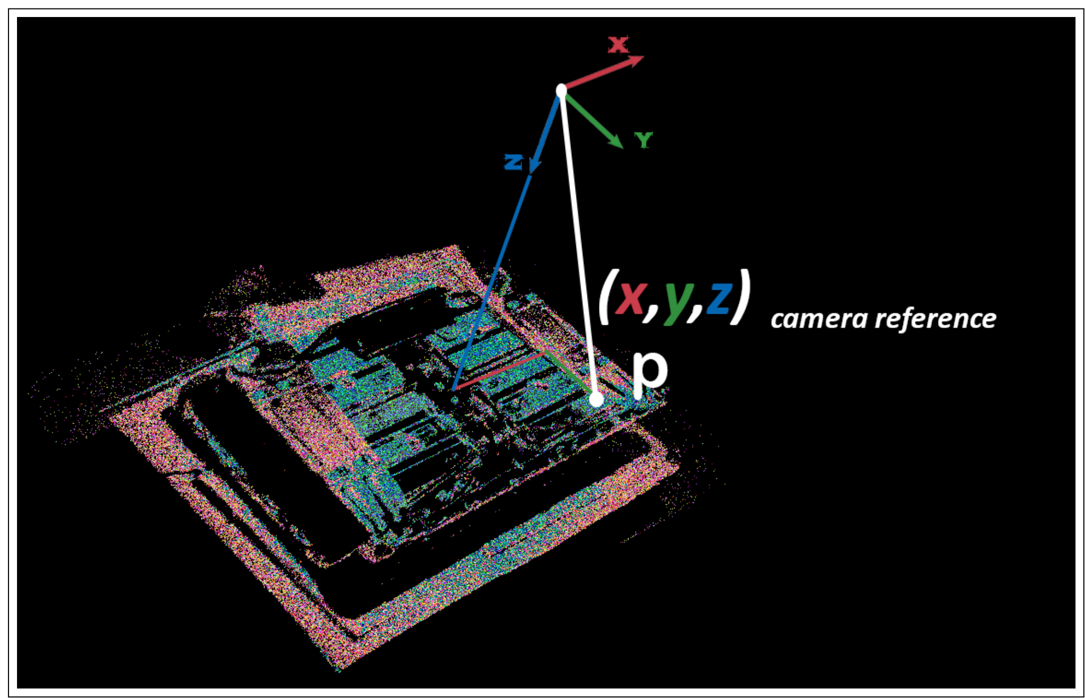


Figure 6.13: Camera reference coordinates concept

[2]The class of the detection is an integer that represents a component, i.e. screws, battery module, etc...

CamR_pos and WR_pos

For this project, two versions of the CamR_pos function have been written. The first one, finds the desired coordinates using and analysing the pointcloud, and the second one (final version) analyses directly the depth image.

- First version

In this first version, the function matches the 2D image (in this case 1200 pixels x 1900 pixels) with the 3D point cloud. To do this matching, first the function finds the maximum and the minimum x and y values of the point cloud. Then, an interpolation is done using: this point cloud values, the maximum and the minimum values of the 2D image (in this case $x_{min} = 0$, $x_{max} = 1200$, $y_{min} = 0$, $y_{max} = 1900$), and the (x, y) positions in the 2D image of the both corners of the label. The result of this interpolation is the x and y values of the label corners in camera reference frame.

Once the x and y position of the the detected label have been found, the next is to find the z position (still working in camera reference frame). Using a for, the function goes over the point cloud looking for the z value corresponding to the x and y positions bounded inside the label and calculating the mean of the z value inside the label.

The main reason for discarding this version were the imprecise results and the high computational cost of the solution. Since, it is not an organized point cloud, the point cloud cannot be indexed. For that reason the function has to run over the whole point cloud to find the positions.

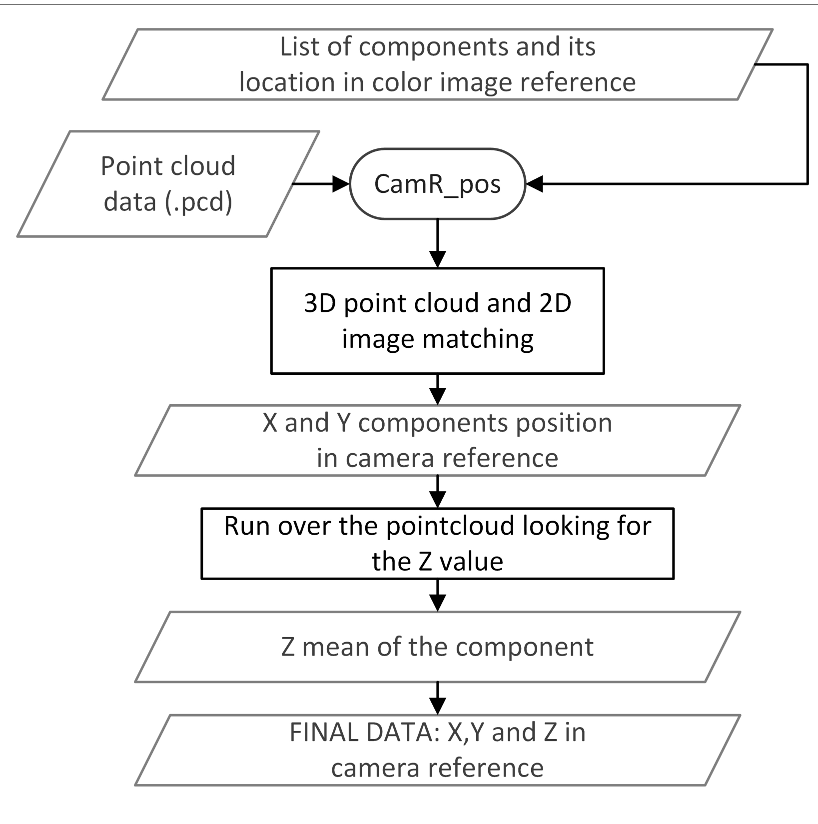


Figure 6.14: CamR_pos flow chart (first version)

- Final version

In the second (and final) version, it has been suggested a different approach for the solution. Instead of looking to the point cloud to find the positions, the depth image has been analysed.

Thus, the idea in this second approach is to find the z coordinates of the desired area using the depth image. To do so, the function runs over the pixels where the components have been detected and looks for the z value in these pixels. With the z value, the pixel position and the intrinsic parameters (explained in section 6.3.2), the x and y values can also be found. Basically, the x and y positions are found solving the equations 6.7 and 6.8 (where x, y are the positions in camera reference, c_x and c_y are the position of the optical center, and f_x and f_y are the focal length):

$$x = (x_{pixel} - c_x) * z * f_x^{-1} \tag{6.7}$$

$$y = (y_{pixel} - c_y) * z * f_y^{-1} \tag{6.8}$$

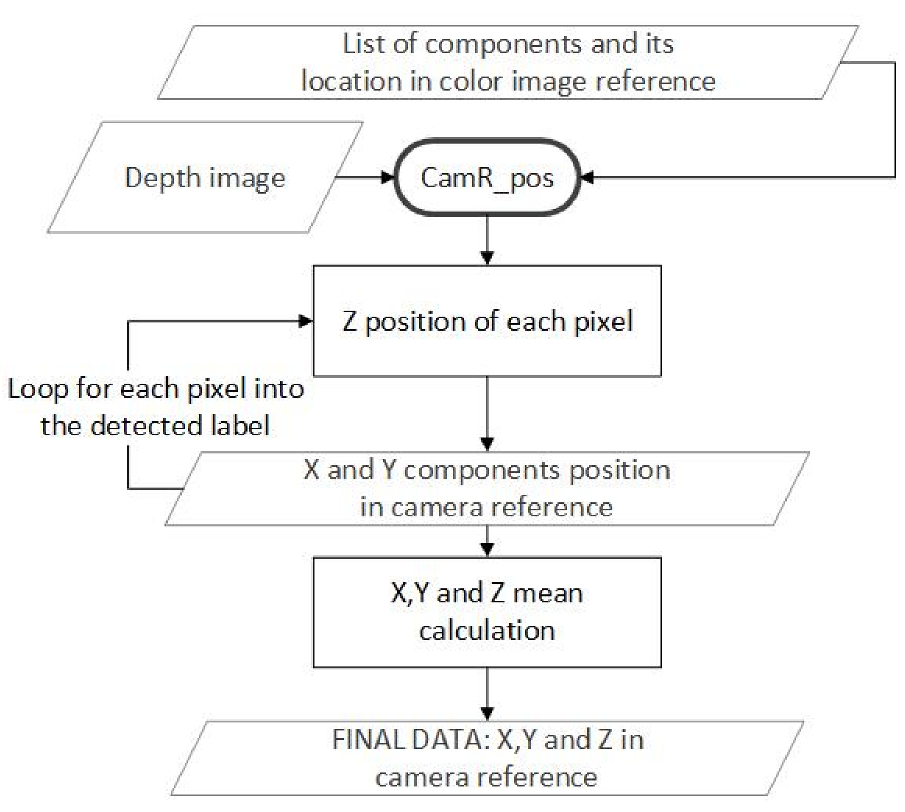


Figure 6.15: CamR_pos flow chart (final version)

After calculating the positions in camera reference frame, the world reference coordinates can be calculated and this is the aim of the WR_pos function B.3. To do so, the transform matrix [3] of the camera in world reference has been calculated. The transformation matrix is obtained with the rotation and the translation of the camera at the capture instant. Thus, the world reference positions are finally calculated using the equation 6.9, where POS_{OBJ}^{WR} is the position of the object

[3]Matrices allow arbitrary linear transformations to be displayed in a consistent format, suitable for computation.This also allows transformations to be concatenated easily (by multiplying their matrices). [11]

in world reference, H_{CAM}^{WR} is the transformation matrix of the camera frames to the world reference frames and POS_{OBJ}^{CAM} is the position of the object in camera reference.

$$POS_{OBJ}^{WR} = H_{CAM}^{WR} * POS_{OBJ}^{CAM} \tag{6.9}$$

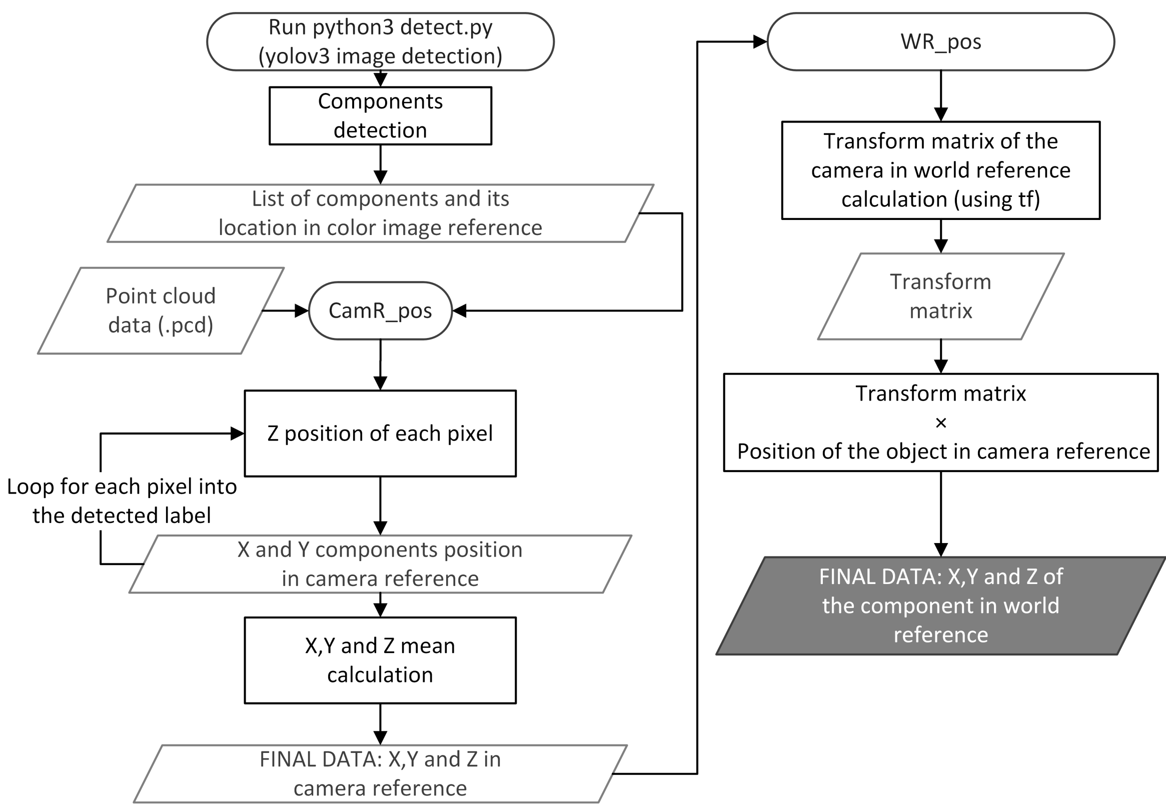


Figure 6.16: WR_pos flow chart

6 Degrees of Freedom of the screws

Given that in this project the specific tools needed for the removal operation have not been designed, and how it is explained in section 6.4 only the (x,y,z) coordinates in world reference has been obtained.

For the specific case of the screw removal operation, depending on the tool, an estimation of the 6 Degrees of Freedom (DOF) of the screw might be needed. Although it has not been implemented, a theoretical analysis of this solution has been done.

To find the 6 DOF of the screws, the designed solution should follow the next steps:

- Extract an small round of interest point cloud near the screw. This action should be repeated x times to have a high density point cloud in the area.
- A 3D model of the screw should be created (in .stl format)
- Mesh the model to create a model pointcloud. This operation can be done i.e. with tools like Meshlab or Blender.

- Align the meshed model with the recorded point cloud. The ICP tool could be useful for this task.
- Obtain the 6 DOF of the screw.

Chapter 7

Methodology, task planning

Robots need task planning algorithms to sequence actions toward accomplishing goals that are impossible through individual actions. In the context of the work presented in this thesis, each battery pack is different depending on the model. Thus, the order of the operations is going to be different depending on the position of these components, i.e., a specific battery pack model can have the battery management system screwed on the modules, but in another one maybe this component is screwed next to the battery modules, so the order of operation is different. For that reason, task planning has been one of the main challenges of this project.

This chapter presents the task planning methodology followed to make the task planner design, then explains the task planner design and how it integrates all the system elements. A detailed explanation and the flow charts of all the involved Python-based functions and scripts are also included.

7.1 Task planning methodology analysis

In this study, there have been considered two ways of handling the design of a computer vision based dismantling system capable of dismantling a LIB pack to module:

- **Model detection combined with a database position**

  One possibility to solve the task planning issue is to use a LIB pack model detection combined with a database containing all the positions and order of operations.

  First of all, this model requires the creation of a database containing all required operations and positions. As each battery pack needs its own predefined data, if the OEM (Original Battery Manufacturers) change or make some variations to the model, the database must be updated.

  In this method, computer vision techniques detect the specific model of the battery pack and once the battery model is known, the next step is to define its orientation and position in order to relate and use the information located in the database. This database contains the positions and the operations that the robot must follow to perform the disassembly. [35]

Pros	Cons
- Optimal path	- Huge database
- Robust to errors	- Sensitive to changes
- Time effective	- Unnecessary moves if some parts are missing
	- Possible crash with not perfectly screwed screws

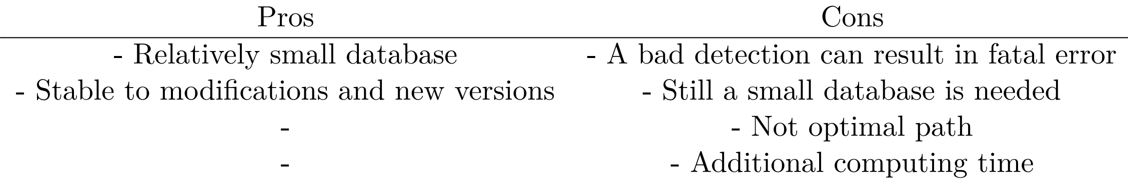
- **Fully automated dismantling system**

  Another handling for the design of the task planning problem is a completely automated dismantling system. For this handling, the main challenge is to find a common dismantling task plan for all the different battery models and reduce the consequences due to bad detections.

  First of all, the system detects, cuts and removes all the wires and cables. Afterwards, it detects and removes all the screws. Once the screws are removed, some components are free to be taken out. When these components are out of the battery pack, the system checks if there are new accessible screws, and repeats all the operations.

  Given that some components have more complex dismantling operations, not just unscrew and remove, this system should also have a small database containing the operations for this group of components. Thus, when the system detects a component that needs a special operation or moves, i.e. a slide, it will access the database of moves.

  In a system of this kind the detection errors can be fatal, so a safety database should also be included. The safety database contains the necessary information to ensure that the operations are safe. For instance, the number of screws that have to be removed for removing a certain component. [35]



Pros	Cons
- Relatively small database	- A bad detection can result in fatal error
- Stable to modifications and new versions	- Still a small database is needed
-	- Not optimal path
-	- Additional computing time

Given that one of the requirements of the system is to have high flexibility, the fully automated dismantling system handling presented above has been selected as the methodology to follow by the task planner.

7.2 Fully automated dismantling system

Due to the elevated degree of autonomy of the fully automated dismantling systems, this kind of systems requires a high-level task planner. Theoretically, the system should be capable of resolving and carrying out all the uncertainties appeared during the dismantling process.

Nonetheless, sometimes the achievement of the robustness and the flexibility required in the disassembly process is a huge challenge. It is impossible to guarantee that all the uncertainties can be solved using automation exclusively.

It also has to be taken into account that there are variations and uncertainties caused by the use of the product. Therefore, LIB that has to be treated are in their End-of-Life, and this has its consequences.

In order to solve or minimize to the maximum these uncertainties, the disassembly has been implemented following the principle of cognitive robotics. 'Cognitive robotics allow a robot to reason, revise and perceive change in unpredictable environments' [35] https://link.springer.com/content/pdf/10.1007

7.2.1 Agent emulating human behaviour

Human operators are more capable of dealing with uncertainties in comparison to automated systems. Thus, the intention of cognitive robotics is to emulate the behaviour of a human operator

during the disassembly process.

The agent emulating the human behaviour needs to be adapted with the closed perception action loop architecture and competences of the automation. Due to the limitations of automation, human assistance must be included in the behaviour control. As a consequence, in the hypothetical case of unresolved situations an operator will help the robot.

The first step is to recognise the model of the sample in order to see if the model is known. If the model is known, the robot follows the instructions in the database. In case that the system does not realise a successful dismantling, the operator helps to learn and revise the system. If the model is being seen for the first time, the robot (cognitive robotic agent) executes different operations according to the components detected in the current state. The removal process is carried out with a certain number of attempts and different techniques. After exhausting the automated alternatives the operator assistance is requested.

In relation to the process flow (see Figure 7.1), a sequence of actions is generated using the knowledge in the database and the disassembly system goes through the states of disassembly. When one main component has been successfully removed, the agent passes to the next stage of the disassembly. This continues until the goal state is reached.

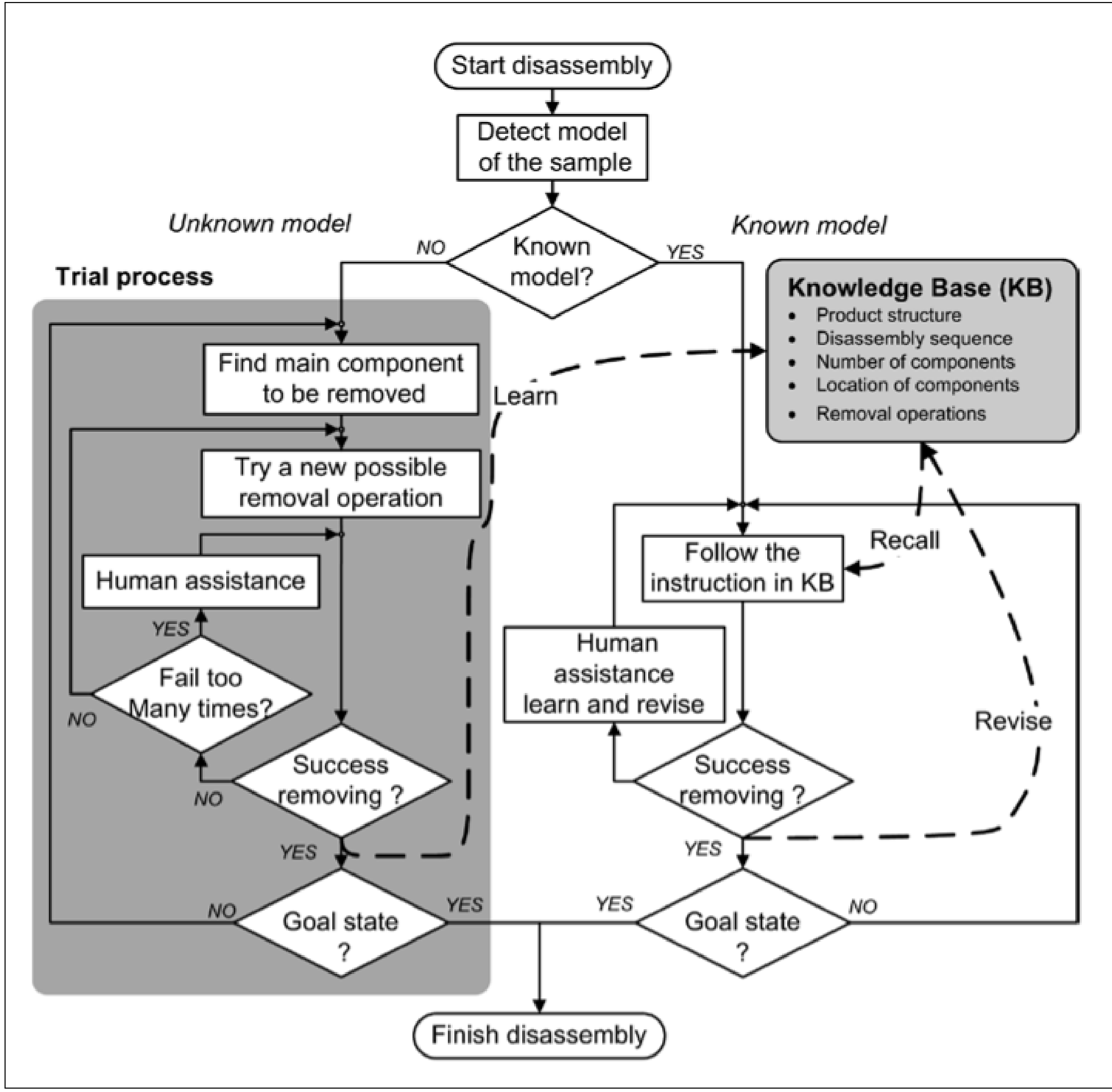


Figure 7.1: Concept overview: Agent emulating human behaviour [12]

7.2.2 Semi-destructive disassembly

'The semi-destructive approach aims to destroy only connective components, e.g. via breaking, folding or cutting, leaving main components with little or no damage. This increases the efficiency of the operation and has been found in many cases to be economically feasible. Many research works relating to automatic disassembly use semi-destructive techniques to overcome the uncertainties in the product condition and geometry. ' [35]

7.3 Task Planner Design

The task planner is responsible for merging the different functions and elements of the system. To do this, the task_planner.py script based on Python has been created (see the full code in appendix B).

Fundamentally, the task planner script integrates the robot, the computer and the Zivid camera, using tools like Python, ROS, OpenCV, etc... . Using the information provided by these elements, the task planner is in charge of decision making. After analysing the situation and the different components, it communicates with the robot in order to send the dismantling actions.

The designed task planner integrates different Python scripts and functions. The aim of each function or script is different. To explain them, in this section, they have been categorised depending on their functionality inside the system.

- **Image Capturing**
- **Object detection (2D Image analysis).**
- **Pose estimation (3D Point cloud and depth image).**
- **Decision making.**
- **Robot communication and path plannification (using ROS and MoveIt!).**

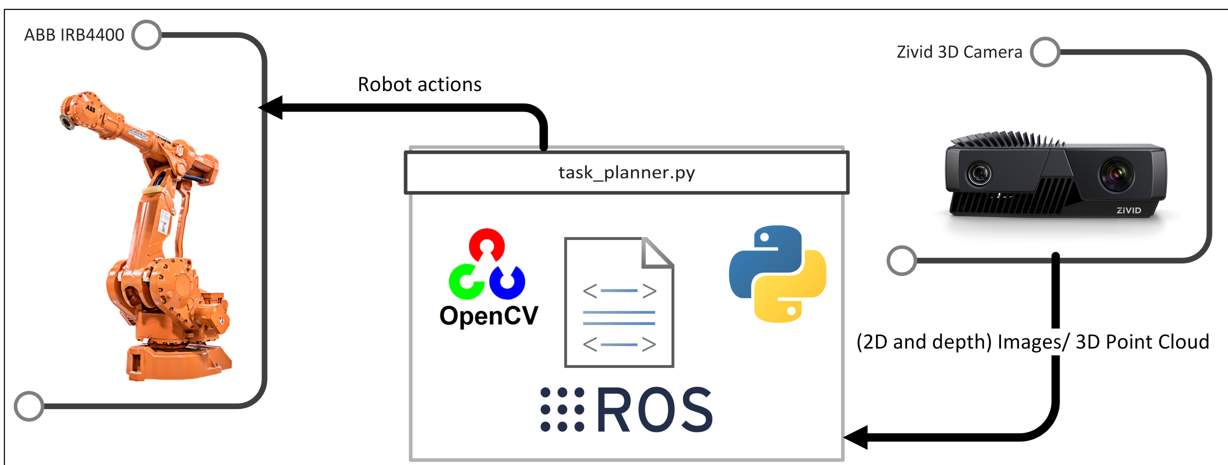


Figure 7.2: Task planner concept

7.3.1 Task Planner overview

This section aims to give a general overview of the task planner process and mention the different functions and scripts. After a short explanation, a reference to the detailed description of these function and scripts has been provided.

First of all, the main loop of the task planner (or the dismantling system) is the next: Takes 2D and 3D Images, detects components, finds the world reference positions, decides the order of the operations, removes the components and repeats this actions until the goal state is reached (see step A-E respectively in Figure 7.3).

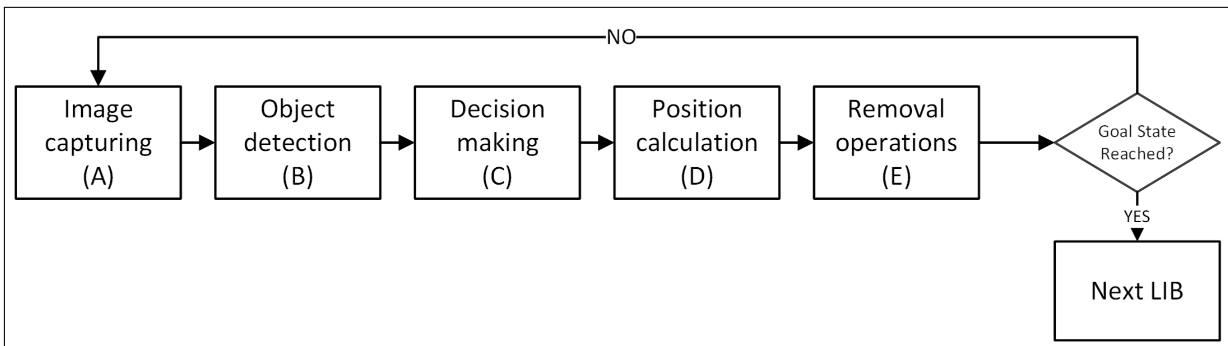


Figure 7.3: Task planner main loop

Image Capturing (A)

Firstly, the task planner moves the robot into three predefined positions (certain translation and orientation) in order to take the images (runs 'rosrun lefty_move move_to_pict.py').

The aim of taking three images is to ensure that the system is able to indentify the different parts of the LIB pack. For that reason, the images are taken from different orientations. In case that for other LIB packs were necessary to take four pictures instead of three, the code can be easily modified.

The information provided from the images does not contribute precisely in the same way (in the task planner point of view). In this case, the first and third images are taken with more inclination. These images aim to detect the screws placed in the lateral sides of the battery. Sometimes, given the geometry of the battery packs, the screws placed in this location can be occluded. Then, the second image is analysed to detect all the components and the decision making is based on it.

To give a better accuracy to the system, the positions of the detected screws (in different images) are merged. Thus, if, i.e. a screw is detected into 2 or more images a mean of the positions is calculated.

Used functions and scripts:

- **Script (ROS): move_to_pict.py** -> See the full explanation in section 7.3.2, flow chart in Figure 7.8 and code in appendix B.9.

Object detection (2D Image) (B)

Once the images have been taken, the next step is to apply the object detection. The object detection part is responsible for detecting the different components in the image. Such as screws, modules and BMS. For more information regarding the detection of different battery components, see Chapter 5.

the object detection script used by the task planner is *detect.py* [1]. This script uses the YOLO (You Only Look Once) object detection system to obtain the image position, of the components. For more information about the YOLO alorithm, see section 5.2. The output of the object detection system is a .txt file containing all the image coordinates of the labels containing the objects.

Below, in Figure 7.4, there is an overview of the inputs and outputs of the script *detect.py* during an object detection.

- **Script (Python): detect.py**

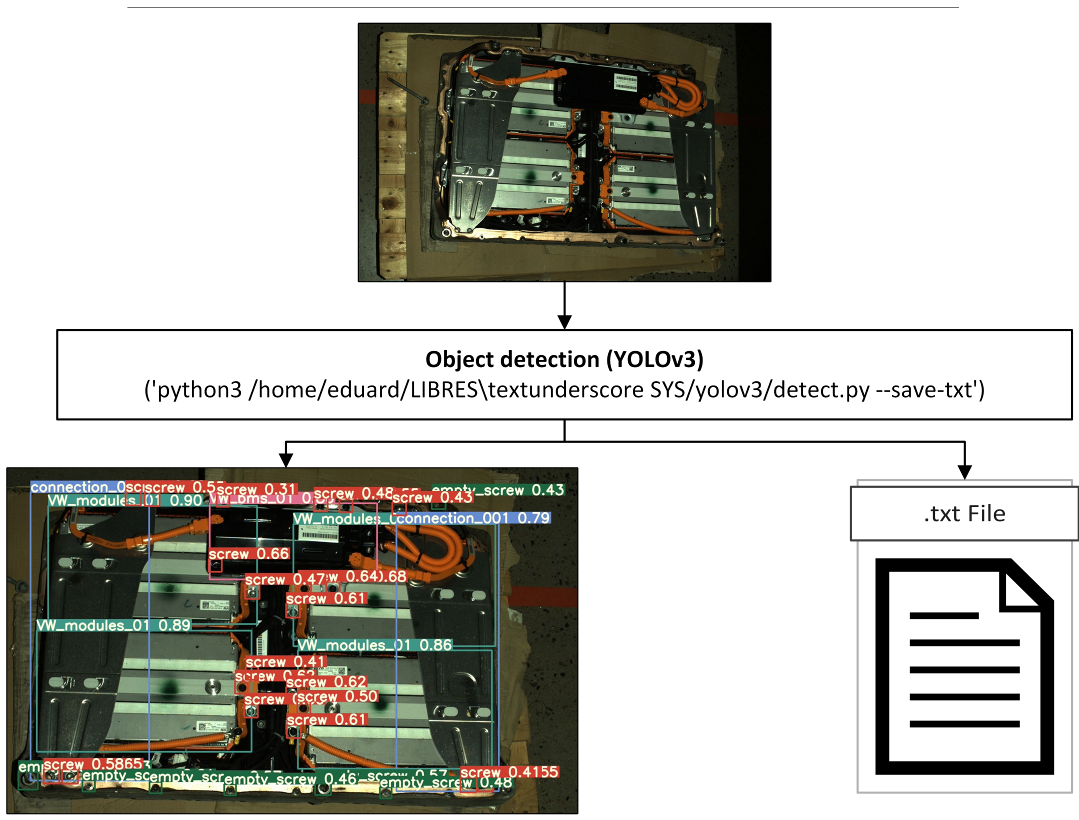


Figure 7.4: Object Detection Overview

Decision making (C)

Once the different objects are detected, the *num_components()* function converts the data from the lines of the .txt files into arrays in order to ease the programming. For more information, see section 7.3.2.

[1]Calls: ('python3 /LIBRES_SYS/yolov3/detect.py –save-txt')

The data returned by the function *num_components()* is used by the function what_component() in order to decide the order of the removal operations. To realize this task, the function establish priorities to the components using the computer vision based subfunction has_comp_over().

To decide the order of the removal operations, only one image is used, the one taken with less camera angle with respect to horizontal. The images with a higher inclination are only used for detecting screws.

- **Function (Python): num_components()** -> See the full explanation in section 7.3.2, flow chart Figure 7.9 and code in appendix B.5.
- **Function (Python): what_component()** -> See the full explanation in section 7.3.2, flow chart Figure 7.10 and code in appendix B.6
- **Function (Python): has_comp_over()** -> See the full explanation in section 7.3.2, flow chart Figure 7.11 and code in appendix B.6

Position calculation (D)

At this point the system has the positions of the objects in 2D camera coordinates (pixels). These positions are not useful for the system thus, in order to move the robot to these positions, they have to be transformed into 3D points.

Therefore, to find the world reference coordinates of the different components, first the camera reference coordinates [2] are found (CamR_pos()). Then, given that the capture positions are known, the world reference position of the objects can be obtained (WR_pos).

This action is done for all the images taken. Once all the positions from the different points of view are found the nearby points (representing the same object or component) are merged.

Used functions:

- **Function (Python): CamR_pos()** -> See the full explanation in section 6.4, flow chart in Figure 6.15 and code in appendix B.2.
- **Function (Python): WR_pos()** -> See full explanation in section 6.4, flow chart in Figure 6.16 and code in appendix B.3.

Robot communication and removal operation (E)

Finally, when the order of the operations and all the positions are known, the last step is to proceed with the dismantling. To do so, the task planner calls the removal operation (remove()) for every single component.

For this project, the removal operation has not been specifically designed. Now the system is moving the robot onto the calculated positions (move_pos().py) and waits for 1 second, simulating a supposed removal operation.

Used functions and scripts:

[2] The camera reference coordinates of an object means the position of the object in relative to the camera.

– **Function (Python): remove()** -> See the full explanation in section 7.3.2, flow chart in Figure 7.13 and code in appendix B.4.

– **Function (ROS-Python): move_pos().py** -> See the full code in appendix B.10.

7.3.2 Integrated Functions and scripts

The aim of this subsection is to mention and give a detailed explanation of the different functions and scripts used by the task planner.

Function: main()

The main function basically runs the core of the task planner. See the *main* function flow chart in subsection 7.3.3 (Figure 7.7).

Fist of all, the function launches the variables (*screw_pos, connect_pos, compon_pos, empty_screw_pos, co_tot*). The intention of these variables is basically to save the resulting output of the *num_components* function. Then, it reads the file containing the classes names (LIBRES_classes.names), and saves them into a dictionary. The creation of this dictionary aims to allow the system to have access to the nomenclature (see section 5.3.1). The nomenclature is useful for the system because it can relate the detected classes with their name and in that way know what kind of component it is analysing.

After the object detection is carried out, the function begins to run its principal loop. Note that this loop keeps running until the dismantling is completed. The first action of the loop is to take the 2D and the 3D images of the battery pack (*move_to_pict.py*). Next, runs the object detection to detect the components placed in these images, using the YOLOv3 algorithm (*detect.py*).

When the detection is done, the function analyses each taken image running a "for in range" loop [3]. For each image, it finds the different components and converts the positions of the objects from image base frame to camera reference frame. Note that the decision-making is just done with the second image (see function *what_component* and its flow chart 7.10). Thus, when the detected positions have been converted into camera frame, they are saved into three arrays (*rem_component_cam_0*, *rem_component_cam_1*, *rem_component_cam_2*). Precisely these are the arguments of the function *merge_detection* which is the next to be run. The output of this function are the positions of the different components in world reference frame (*rem_component_final*). The components are placed in order of removal preference. Finally, the function runs a for through *rem_component_final*, calling the *remove* function.

The system considers that the loop is ended when the *rem_component_final* is empty. Therefore, the goal state is reached.

Script: move_to_pict.py

This script aims to move the robot to the predefined capture positions and take and save the 2D and 3D images. Flow chart of the script in subsection 7.3.3 (Figure 7.8)

[3]It is important to consider that the design of these functions is based on the Volkswagen battery pack (section 2.3.2. To have a good view of all the necessary components, in this case, is enough to take three images from different angles.

To do so, the script combines the essential coding to move the robot, presented in subsection 3.2.3, and the ROS communication the Zivid camera.

The structure followed by the script is the following:

– Init the sample class. Configure the camera parameters, create the image, point cloud and depth image subscribers and wait until the camera service is ready.

– Move the robot to the first capture position.

– Invoke the capture service.

– Save the 2D image, the point cloud and the depth info

– Repeat the previous steps for the second and the third capture positions.

Therefore, after completing each robot trajectory, the capture service (of the camera) is invoked. Every time that this service is invoked, the 2D and the 3D information is published into the topics ("*/zivid_camera/color/image_color*"," */zivid_camera/depth/image_raw*" and "*/zivid_camera/points*").

Every time that a color image message is published to the topic "*/zivid_camera/color/image_color*", the rospy subscriber (*image_sub*) launches the *callback()* function. This function converts the image ROS message into an open-cv2 file format. Then it saves the image into the desired directory.

Then, if the point cloud dataset message is published to the topic "*/zivid_camera/points*", the rospy subscriber calls the function *callback3()* which converts the point cloud ROS message data into a .pcd file using the pcl library. More detailed explanation on how to save the point cloud data in section 6.2 (Point cloud and depth images storage).

Finally, when a depth image message is published to the topic " */zivid_camera/depth/image_raw*", the third rospy subscriber launches the function *callback3()*. This function aims to convert the ROS message into a Numpy array and saves it under .npy format. More detailed explanation on how to save the point cloud data in section 6.2 (Point cloud and depth images storage).

The names of the .jpg, .npy and .pcd files, follow the numbers sequence (0,1,2,..).

Function: num_components

The main objective of the function num_components is to read the results of the object detections and to translate these results in Python variables in order to ease the programming.

The inputs of the function are:

- Dict_comp (Dictionary): This dictionary contains the class names corresponding to the class numbers.
- num_im (Integer): This variable gives the information of what image is being analysed. For instance, when the first picture is analysed, the value is 0, in case of analysing the second image the value is 1, and so forth.

The results of the object detections, are saved in the*/output* folder in a .txt file. As it is explained in section 6.4, the .txt file contains the position of two corners of the bounding box (within each component located), the class, and the detection accuracy.

Using a for loop, the function runs over the lines of the .txt file converting them into arrays. Each line is classified and saved into different variables according to the kind of detected components (screws, connective, connection, empty screws or components).

x_1	y_1	x_2	y_2	class	accuracy
862	473	1314	669	1	0.796174

So, the outputs of the function are:

- screw_pos (array): An array containing the coordinates of the detected screws. Referenced in the image base frame.
- connect_pos (array): An array containing the coordinates of the detected connective components. Referenced in the image base frame.
- compon_pos (array): An array containing the coordinates of the detected general components. Referenced in the image base frame.
- empty_screw_pos (array):
- co_tot (array): An array containing the coordinates of all the detected components. Referenced in the image base frame.

Functions: what_component and has_comp_over

The function what_component is the responsible of the decision making. As explained in Section 7.3.1, the function analyses the detected objects and using computer vision decides the order of the removal operations. The function has_comp_over could be considered as a sub-function of the function what_component, it establishes the different probabilities of a specific component of having a component over.

From one side, the inputs for the function what_component are:

- screw_pos (array): An array containing the coordinates of the detected screws. Referenced in the image base frame.
- connect_pos (array): An array containing the coordinates of the detected connective components. Referenced in the image base frame.
- compon_pos (array): An array containing the coordinates of the detected general components. Referenced in the image base frame.
- empty_screw_pos (array):
- co_tot (array): An array containing the coordinates of all the detected components. Referenced in the image base frame.

First of all, the function creates the variable *compon_rem* (array), the aim of this array is to contain the list of the components positions in the removal order. Given the nature of the disassembly, the first components to be removed are the screws. Thus, the screws positions are the first to be added to the *compon_rem* array.

To add the rest of the components and decide the order of the operations, the function what_component runs over the list co_tot analysing each component using the function has_comp_over.

Essentially, has_comp_over realize a computer vision analysis of a specific component, and returns the probability of being over the other components. Thus, the components are ordered from more probability to be over to less probability.

From the other side, the inputs for the sub-function has_comp_over are:

- co_tot (array): An array containing the coordinates of all the detected components. Referenced in the image base frame.
- co_an (array): Array containing the image coordinates of a specific component (the component being analysed).

The analysis of the probabilities is done using computer vision. The function has_comp_over, runs over the co_tot using a for loop and compares each component with the specific component being analysed (co_an).

To realise this comparison, the full image is converted into grey-scale and equalised. Image equalisation is a good method to have better quality images without losing information. Moreover, the equalised image has a better distribution of the intensities of the pixels.

After obtaining the equalised image, the function crops the image into the area of interest, in this case, the area of the analysed component. This window is binarised using an Otsu threshold. Thresholding is an effective tool to separate an object from the background, thus in this case, segmentation is done to try to separate both components. In the Otsu method, the threshold is determined by minimising intra-class intensity variance [19].

When the window has been binarised, the means of the intensities in the intersection area and in the rest of the field are calculated and compared with the maximum value (255). The separation between the values is calculated. This difference is what determines the probability of the analysed component of being over the other component.

This procedure is repeated for all the components. The maximum value is taken because it represents the maximum difference between the analysed components, so the worse case.

The output of the function is:

- max(p_list) (float).

Function: remove()

As explained before, the detailed design of the removal operation is not included in the scope of the project. To program a detailed and working function for the removal operation, the tools should be completely defined. Anyways, the designed function simulates the removal operation and it is prepared to be completed in future stages of the project with the specific removal operation.

The inputs of the function are:

- pos_comp (Array): Array containing the position and the class of a specific component.
- Dict_comp (Dictionary): Python dictionary containing the related names of the detected classes.

First of all, the function remove() checks what kind of component is the selected to be removed. Since the function is structured depending on the type of the component (screw, connection or

general component), in future stages of the project, it is going to be easier to substitute the simulated script for the definitive removal script or function. In the current version, the function calls the same script for all the components (move_pos.py).

Given a specific position (in world coordinates) the script move_pos.py, moves the tip link of the robot to this position. Specifically, the robot does the next sequence of actions (see 1-3 in Figure 7.5):

1. Moves the robot to a safety position. Displaced 30cm safety margin in the z-axis added to the first target.

2. Moves the robot to the desired position.

3. Moves the robot back to the safety position and quits the script.

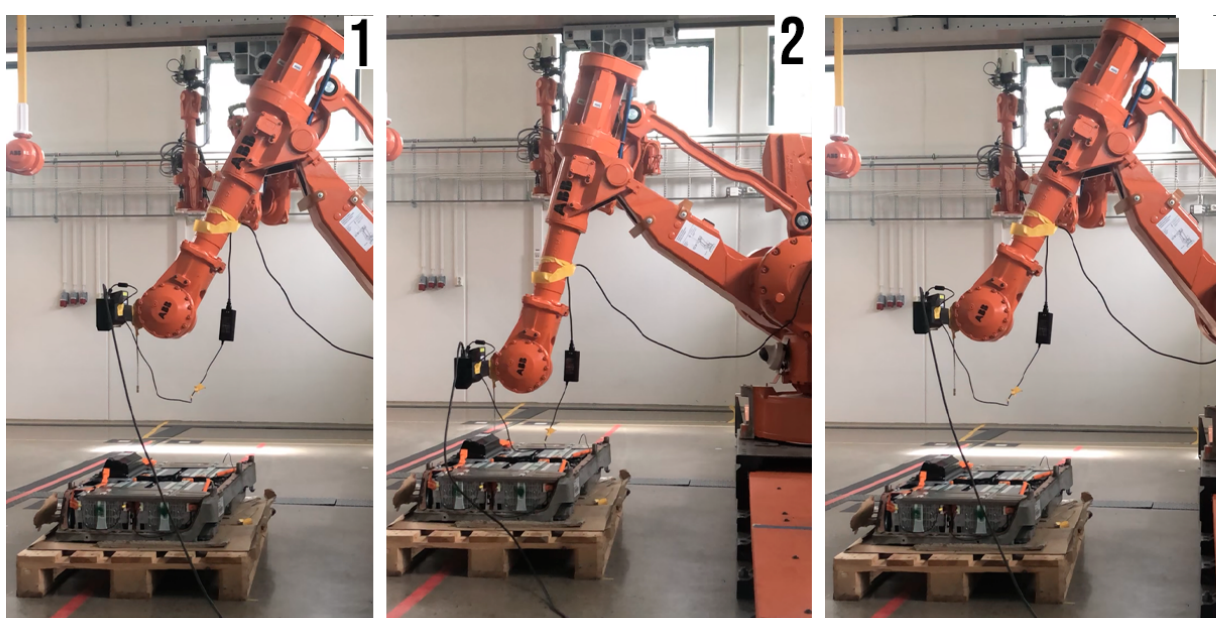


Figure 7.5: Safety positions

Function: merge_detection

The function *merge_detection* aims to merge the positions of the components (for this version just the screws) commonly detected in one or more images.

The inputs of the function *merge_detection* are:

- rem_component_cam_0 (Array): Array containing the positions of all the screws detected in the first image in camera reference frame.

- rem_component_cam_1 (Array): Array containing the positions of all the components detected in the first image in camera reference frame. The components are placed in the list in order of removal. This is because the decision-making has been previoulsy done in this image (given its inclinations and consequently its light conditions).

- rem_component_cam_2 (Array): Array containing the positions of all the screws detected in the last image in camera reference frame.

First of all, the function runs a "for in range" loop finding the world reference frame positions of the input lists of components. To do so, it uses the *WR_pos* function. Then, these positions are all stored in the variable rem_component_WR, also the repeated positions.

The next step of the function is to filter the repeated positions to find the output list (*rem_component_WR_filt*). To filter the points, the function runs over the rem_component_WR array adding the not-repeated components to the filtered list. The function considers the two components as the same one, when the x and y distances are lower than 1cm, and defines the final position as the mean of both points.

Functions explained in other sections: camR_pos and WR_pos

The aim of the functions camR_pos and WR_pos is to find the position of the detected objects in world reference in order to use this information to move the robot to the desired positions (decided by the previously explained functions). They are also used by the task planner, but the full description of the functions is in the Pose Estimation chapter (see section 6.4).

7.3.3 Integrated Functions and scripts flow charts

Task planner Flow Chart

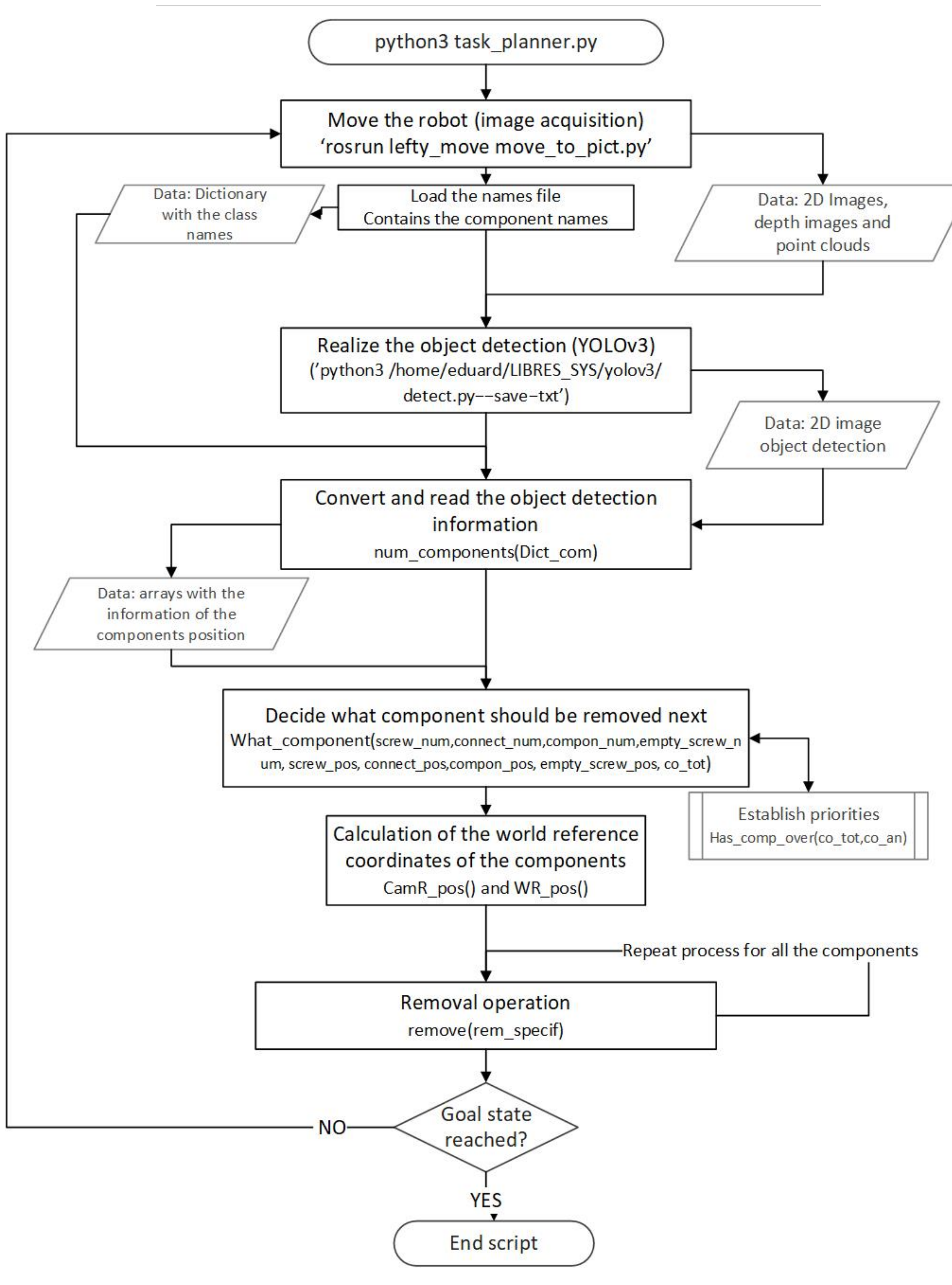


Figure 7.6: Task planner flow chart

Function: main()

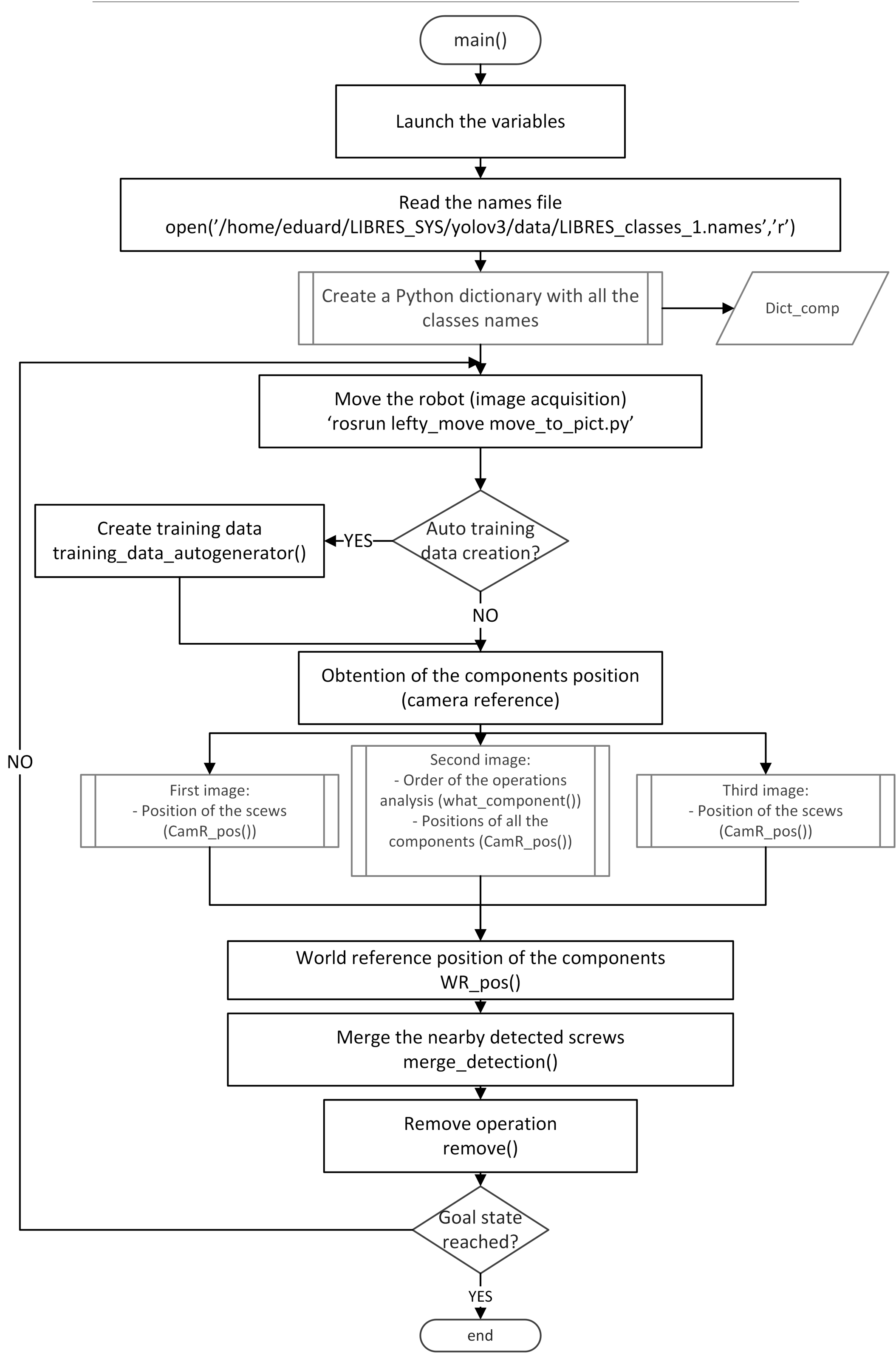


Figure 7.7: Task planner main function flow chart

Script: move_to_pict.py

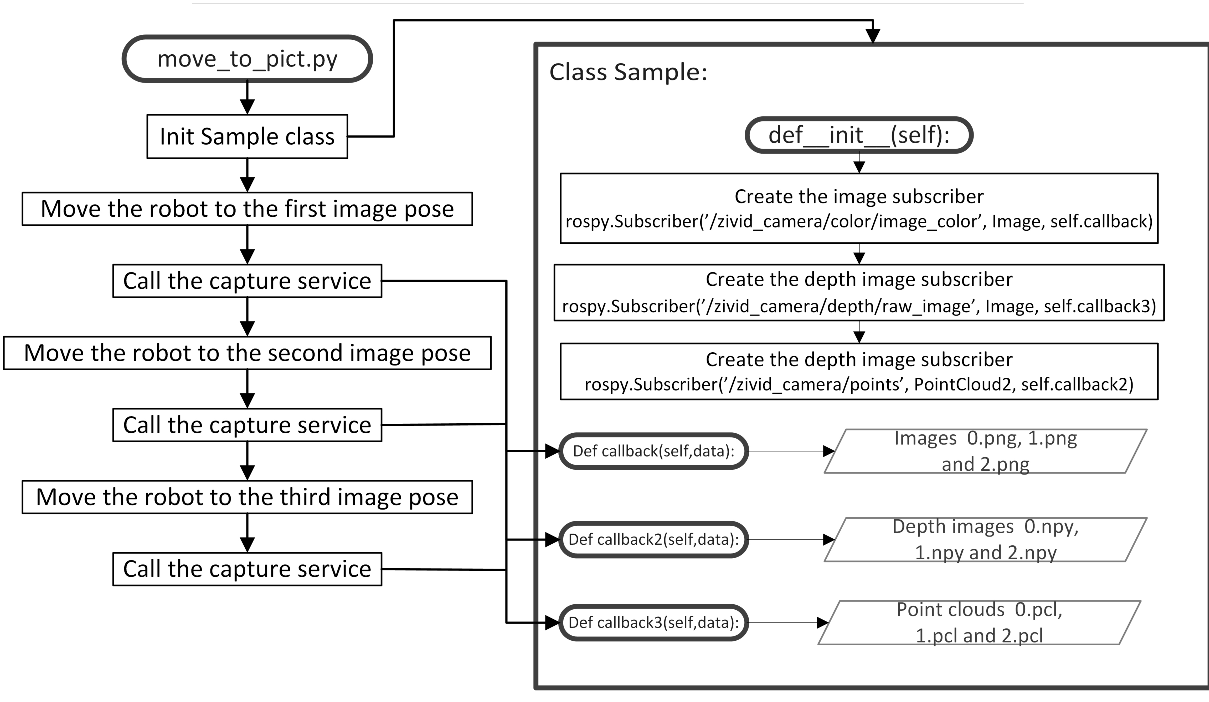


Figure 7.8: Task planner move_to_pict flow chart

Function: num_components

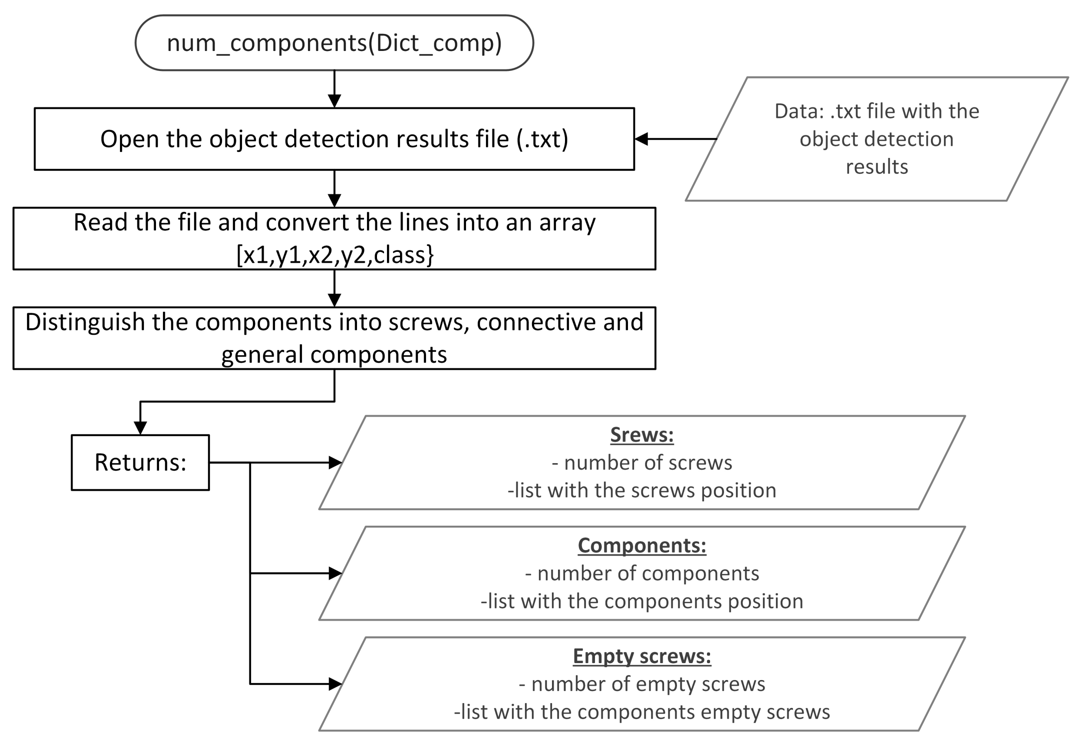


Figure 7.9: Task planner num_components function flow chart

Function: what_component

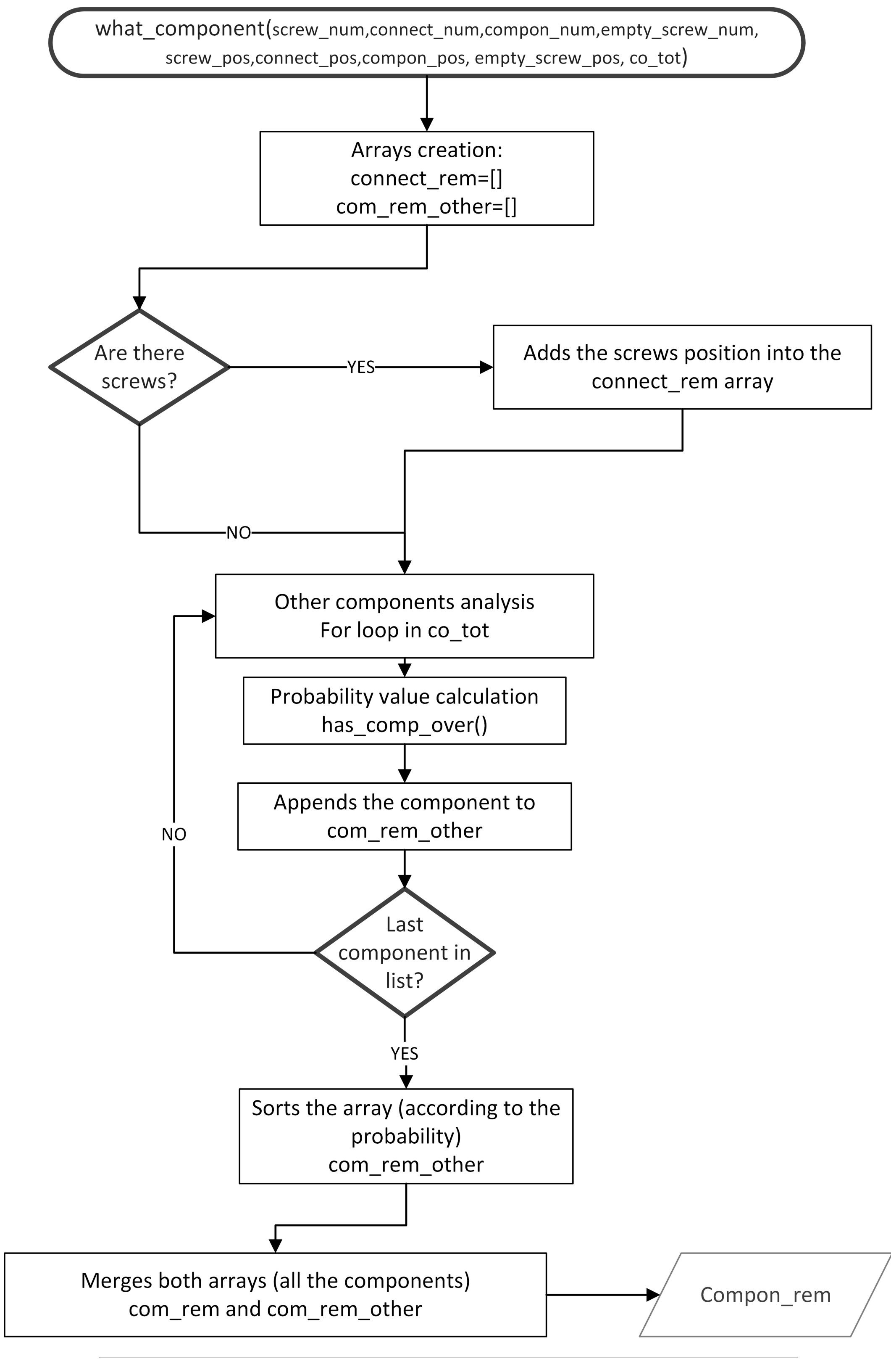


Figure 7.10: Task planner what_component function flow chart

Function: has_comp_over

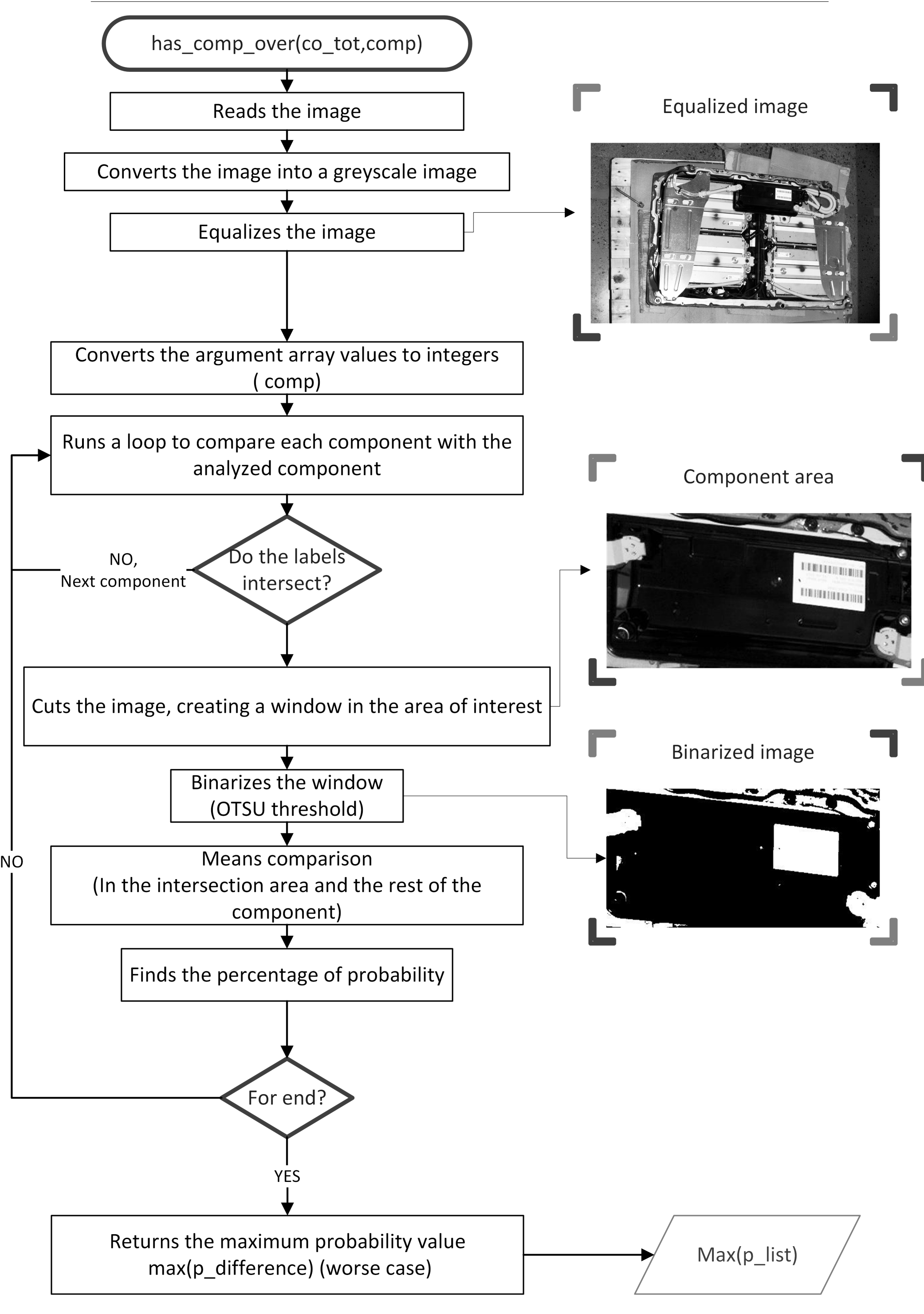


Figure 7.11: Task planner has_comp_over function flow chart

Function: merge_detection

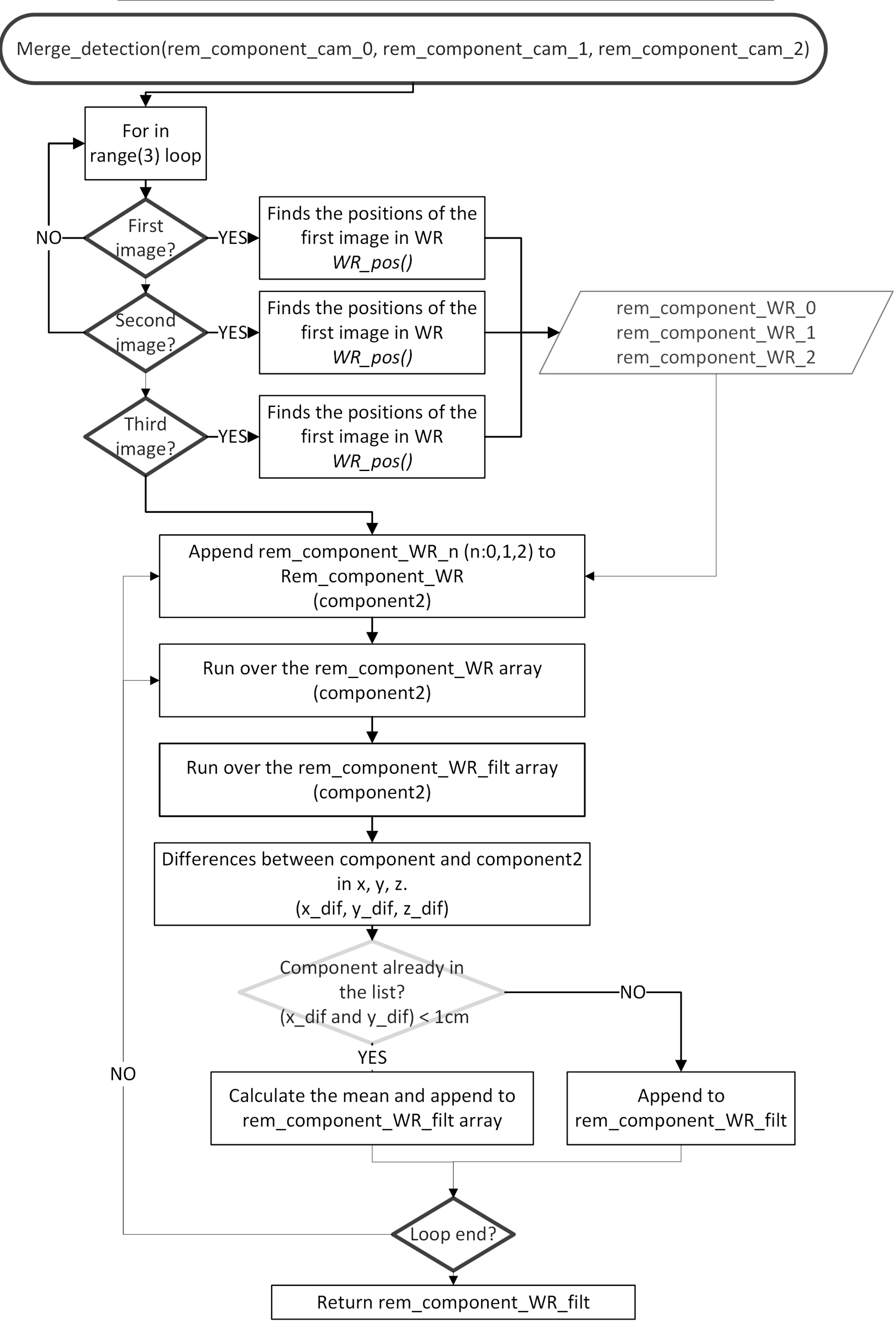


Figure 7.12: Task planner merge_detection function flow chart

Function: remove

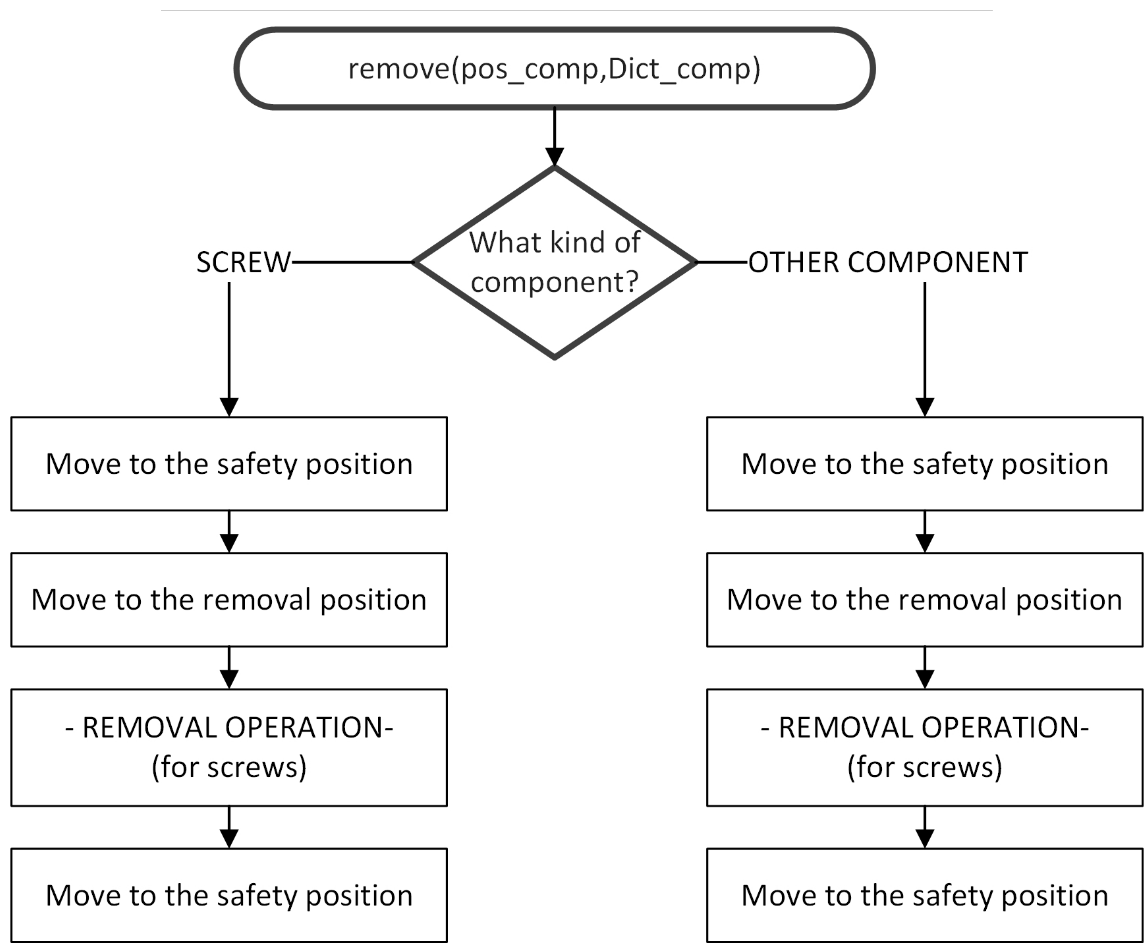


Figure 7.13: Task planner remove function flow chart

Functions explained in other sections: camR_pos and WR_pos

Flow charts in section 6.4.

7.3.4 Running the task planner

The aim of this subsection, is to explain how to run the task planner.

First of all, open a linux terminal and **launch the robot** (to configure the ROS master and the catkin workspace, follow the steps 1-2 in the Guidance, see chapter 4) :

– Working in the simulated mode:

```
$ roslaunch righty_moveit_config demo.launch
```

– Working with the real robot:

```
$ roslaunch wp3_robots load_robot_mod.launch
```

Launch the **camera node**.

```
$ ROS_NAMESPACE=zivid_camera rosrun zivid_camera zivid_camera_node
```

Finally, run the task planner.

```
$ python3 task_planner.py
```

Parameters configuration

When running the task planner, four parameters can be configured in order to modify the behaviour of the task planner. These are:

- - - autotrain (default=False) -> Enables (True) and disables (False) the training data auto generator (subsection 5.3.6).
- - - take_images (default=True) -> Enables (True) and disables (False) the image capturing (subsection 7.3.1).
- - - detect (default=True) -> Enables (True) and disables (False) the object detection (YOLO).
- - - debug (default=False) -> Runs the dismantling in "debug mode", asking for permission to the user to move the robot from the safety positions to the dismantling positions and vice versa.

To configure these parameters when launching the task planner:

```
$ python3 task_planner.py --autotrain False --take_images False --detect False --
    debug True
```

Feedback given by the task planner

The system prints some feedback to let the user know what is the task planner doing at each moment. (See a feedback example in the appendix D)

7.3.5 Conclusion

The proposed task planer is able to recognise which component to remove first and the complete disassembly plan without prior knowledge of the disassembly strategy. This method is therefore well suited for products with large variations and hence increases the disassemblability as defined in chapter 2.

Chapter 8

Project results

The aim of this section is to evaluate the accuracy, timing and capacity of decision making of the task planner. To do so, a Volkswagen battery pack is taken as basis.

8.1 Object detection results

Figure 8.1 shows the results of the YOLOv3 algorithm implementation, where the red filled boxes indicate the position of the connective components, the blue-filled boxes indicate the screws positions, the pink-filled boxes indicate the position of the BMS and the black-filled boxes indicate the position of the battery modules.

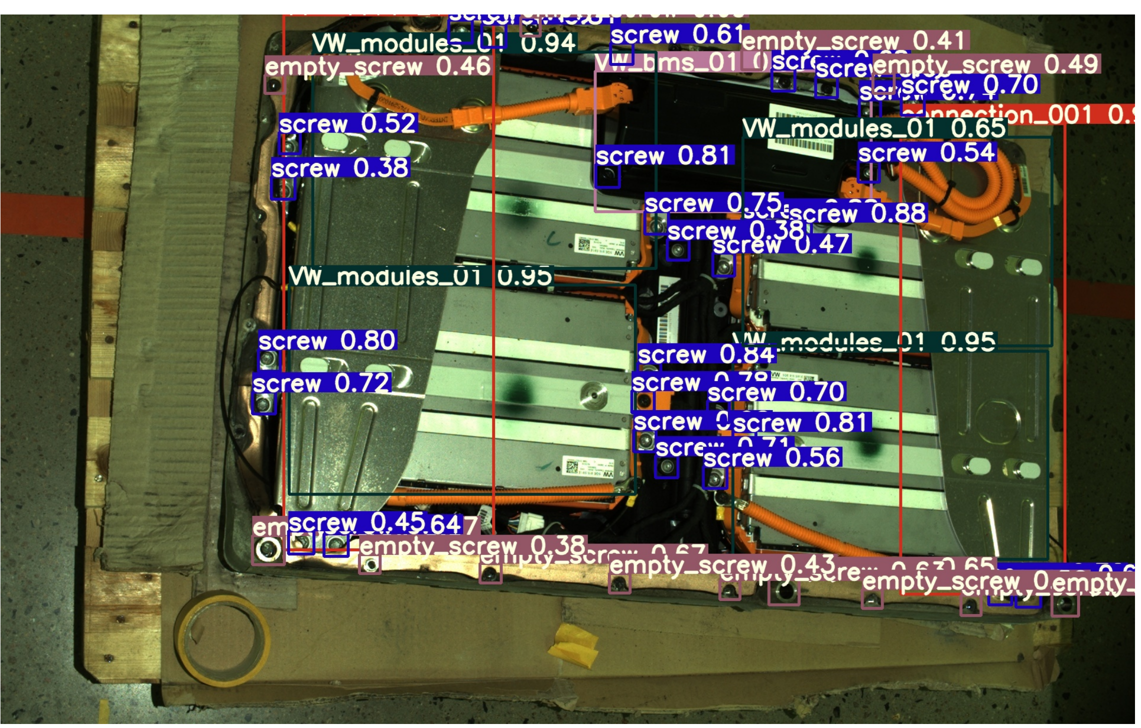


Figure 8.1: YOLOv3 output results

8.2 Time analysis

The aim of this subsection is to analyse the timings of the operations realized by the system. These are:

- Image capture.
- Image detection.
- Data analysis and decision making.
- Move the robot to pose.

Image capture (mean time: 29.1 seconds)

In this case, the image capture process refers to the robot movement into the image positions plus the image capturing. In order to do the analysis, the process has been divided into seven different actions. The first action refers to the robot model loading, and the rest refer to the robot movements and image captures, see Figure 8.2.

- Load the robot model (MoveIt!). Mean time: 3.2 seconds.
- Move to the first position. Mean time: 6.7 seconds. (1) in Figure 8.2.
- First image capture. Mean time: 3.6 seconds. (2) in Figure 8.2.
- Move to the second position. Mean time: 3.0 seconds. (3) in Figure 8.2.
- Second image capture. Mean time: 3.5 seconds. (4) in Figure 8.2.
- Move to the third position. Mean time: 6.0 seconds. (5) in Figure 8.2.
- Third image capture. Mean time: 3.6 seconds. (6) in Figure 8.2.

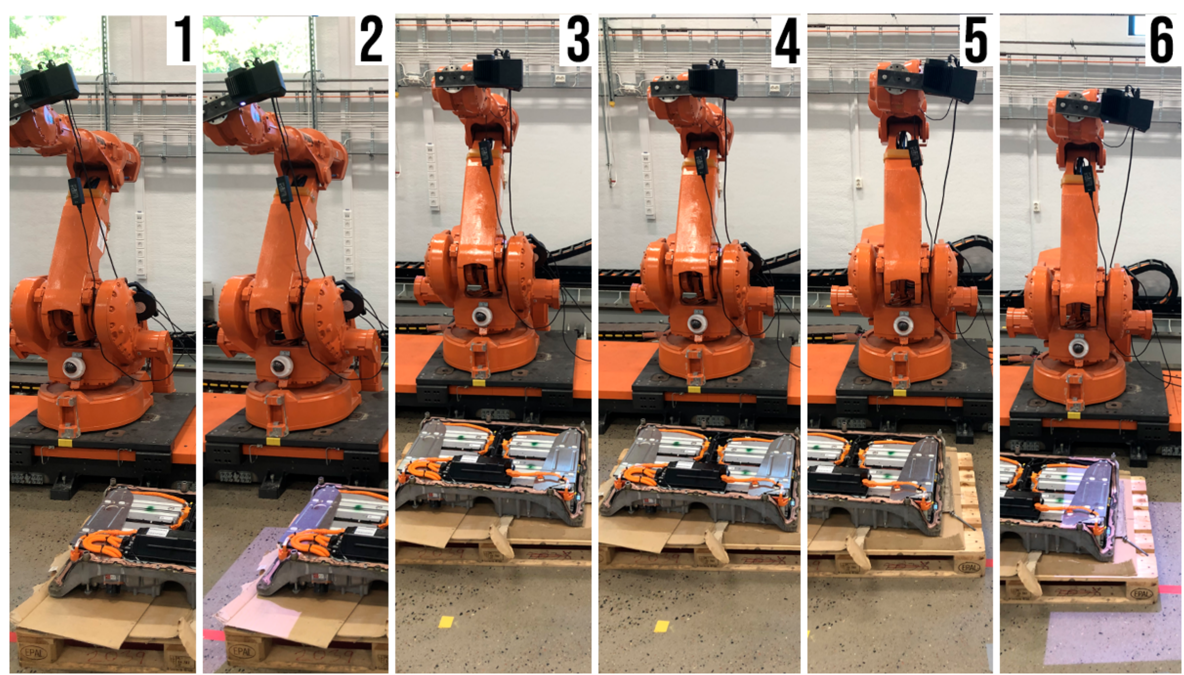


Figure 8.2: Image capturing actions

Image detection (mean time: 4.8 seconds)

The image detection time is the necessary timing to achieve the object detection. Thus, to apply the YOLO algorithm to the three taken images and find the different components in the 2D images. Mean time: 4.8 seconds.

Data analysis and decision making. (mean time 9.2 seconds)

Data analysis and decision-making time refer to the timing of calculating the positions in world frame coordinates and decide the optimal path for the operations. Mean time: 9.2 seconds. Note that this is the main objective of the proposed system.

Move to the desired positions (mean time: 13.1 seconds)

The robot approaches the components to realize the removal operation. As explained before, to implement this approach, the robot first moves to a safety position displaced thirty centimetres in the Z-axis (world frame coordinates) and then moves to the desired location. After that, the robot moves back to the safety position. The timings for this operation have been divided into six sub-processes.

- Load the robot model (MoveIt!). Mean time: 3.2 seconds.
- Move to the safety position. Mean time: 1.8 seconds.
- Move to the component position. Mean time: 2.2 seconds.

- Removal operation. Mean time: not applicable, since it depends on the removal operations, which is not considered in this project.
- Load the robot model (MoveIt!). Mean time: 3.2 seconds.
- Move to the safety position. Mean time: 2.6 seconds.

Timing summary

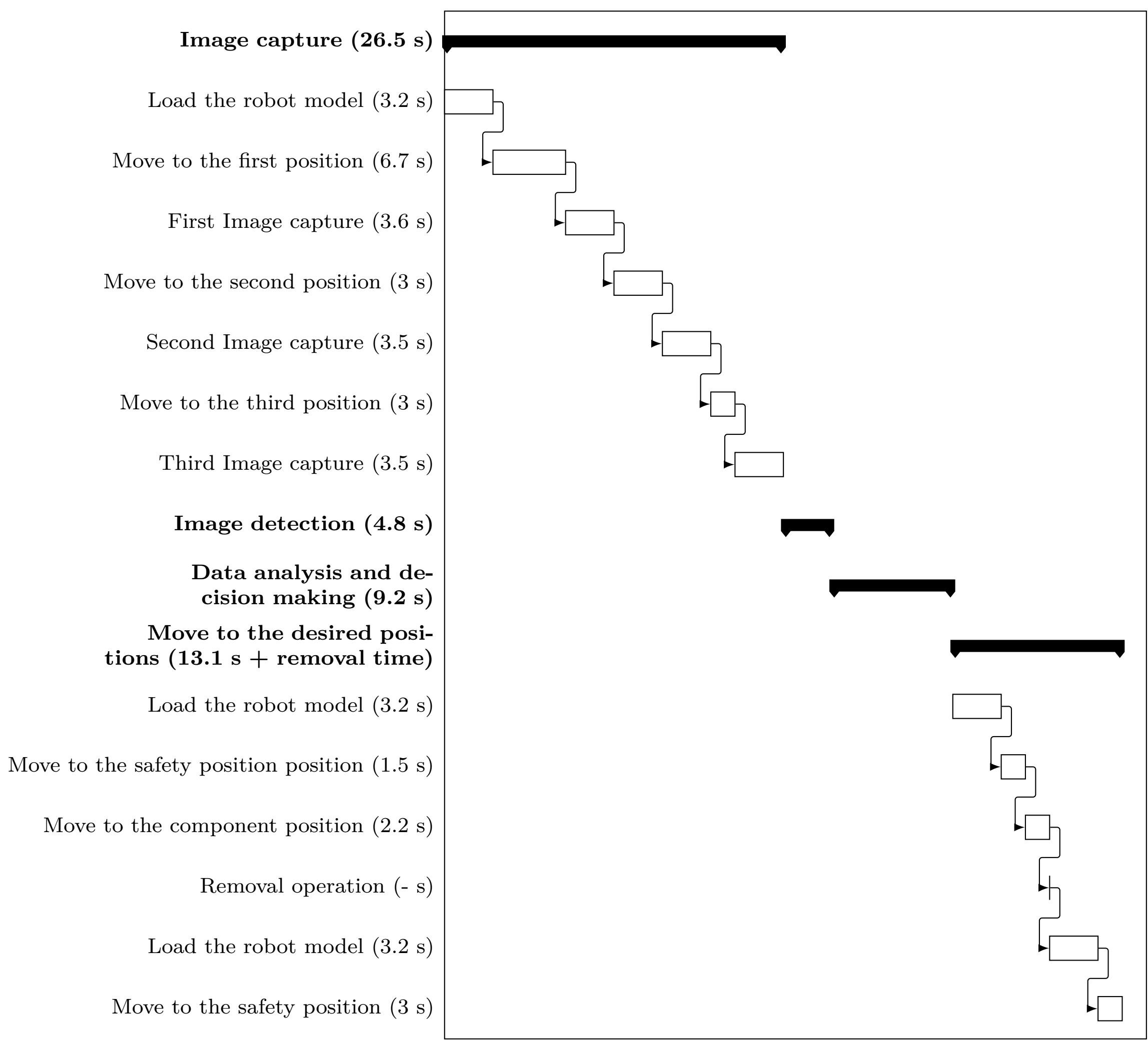


Figure 8.3: Timing summary

8.3 Decision-making: Optimal path

The decision-making of the system (the order of the removal operations) affects directly onto the dismantling time. For this reason, the aim of this section is to analyse the order suggested by the task planner for the Volkswagen LIB pack [1].

[1]Note: It was expected to realize more tests with different battery pack models, but due to the coronavirus lockdown and the difficulties accessing the lab, it has finally been tested with one battery pack.

Optimal dismantling plans

After manually dismantling and analysing the battery pack, the optimal dismantling plans have been defined (see Figure 8.4).

In the optimal path the screws need to be removed. This step is always the first one. Therefore, decision-making does not have a direct influence here. As shown in Figure 8.4, the second step is to remove the connective components and the battery management system. In this case, the order of the removal operations for these three elements is not critical for a successful removal (right connective - BMS - left connective, left connective - BMS - right connective or BMS - both connective).

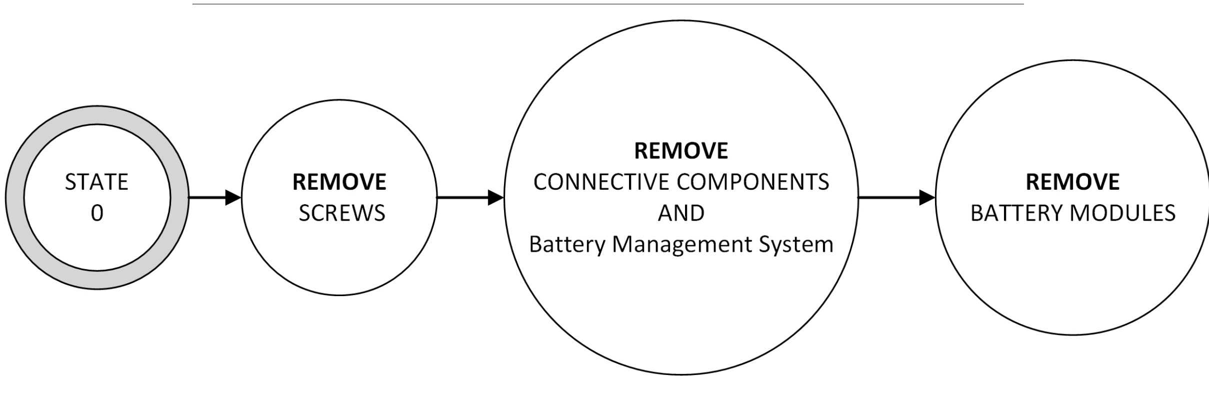


Figure 8.4: Optimal dismantling plans

Dismantling plans proposed by the system

A set of tests have been carried out under different conditions (i.e different orientations, different ambient lights conditions, etc..), the system has given a good response.

It has been observed, within the proposed dismantling plans, that the system follows the guidelines defined in section 8.3. Because the BMS and the two connective components (left and right) are not overlapping, the system is proposing two different plans that are equivalent. These are referred as the first and the second plans and are illustrated in Figure 8.5 and 8.6 respectively.

In the first plan, the system begins removing the screws (important to highlight that the screws are always the first components to be removed), and then decides to remove one connective component (in some tests the left one in other tests the right one), the BMS, and the other connective component in that order. Finally, it removes the battery modules. See Figure 8.5.

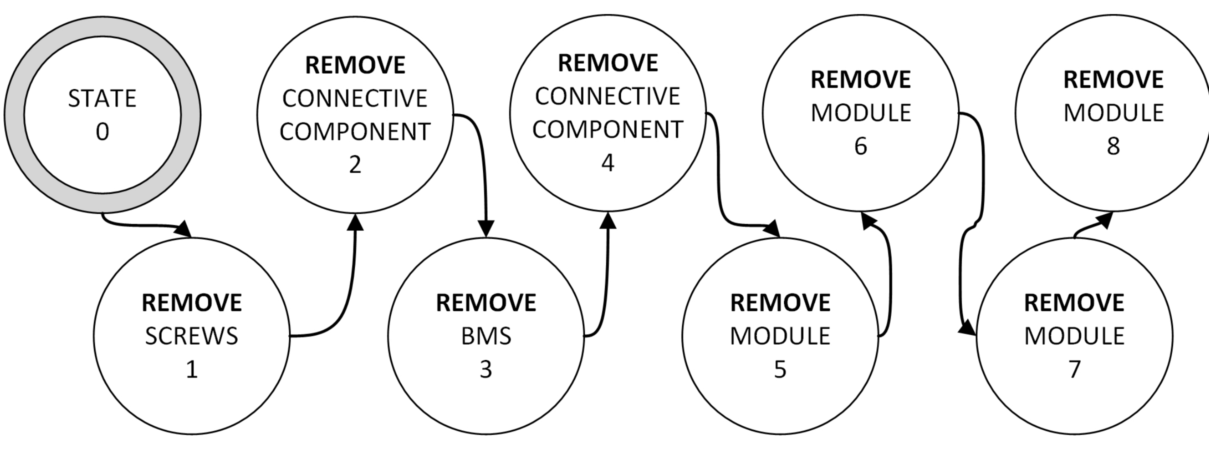


Figure 8.5: Dismantling plan proposed by the system (A)

In the second plan, screws are still the first component to be removed. Then, the system proposes to remove the BMS and the connective components in that order. As in the previous plan, the battery modules are removed the last. Thus, the main difference observed between the first plan (Figure 8.5) and the second one (Figure 8.6) is that in the second plan the BMS is removed the first (after the screws). This has no impact on the final disassembly process.

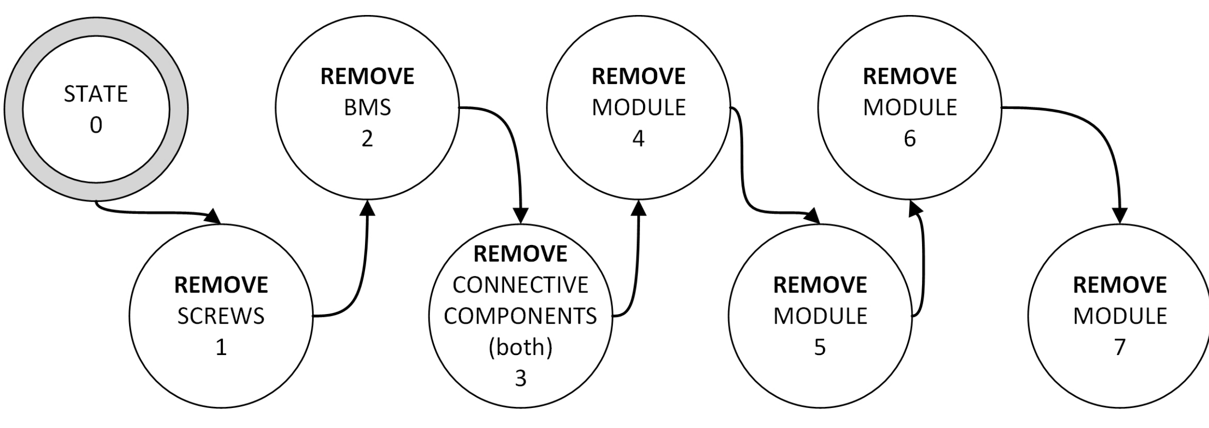


Figure 8.6: Dismantling plan proposed by the system (B)

8.4 Accuracy

To analyse the system's accuracy, a 3D printed pointer [2] simulating the tool has been used. The simulated tool consists of a thin bar of 25 cm long with a sharp end (simulating the tool center point). The tool is illustrated on Figure 8.7.

[2]It has been considered that the 3D printed tool is good enough for testing the accuracy because of its resilience and production facilities.

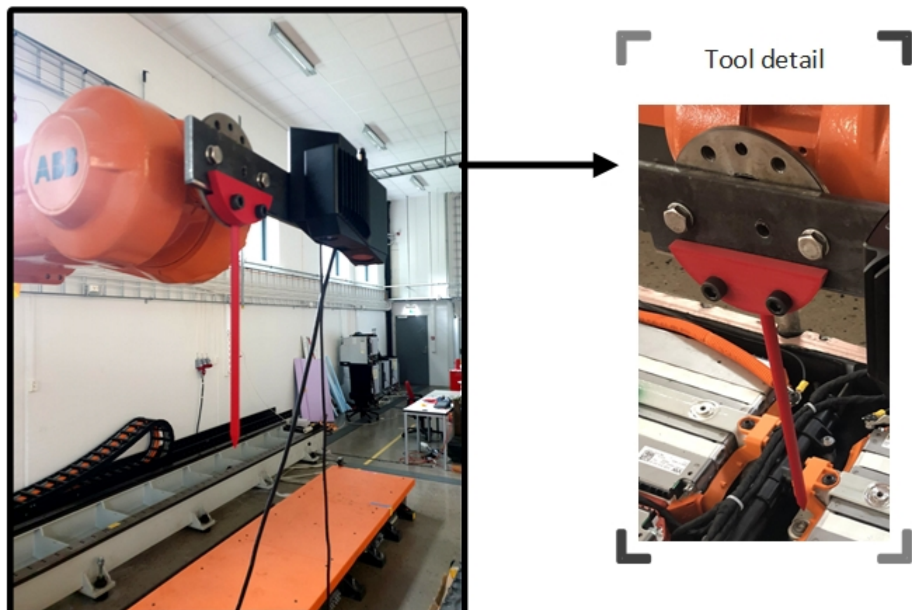


Figure 8.7: Tool

With the tool mounted, it has been run the task planner in debug mode. For safety reasons, the robot TCP has been moved 3cm above in the Z-axis (word base frame) in order to avoid collisions with the battery pack in case of failure. Some of the tests are shown in Figure 8.8. In the majority of the cases, the system has an accuracy of (<0.5mm). The error, considering the diameter of the screws heads (from 1 to 2.5 cm), is acceptable.

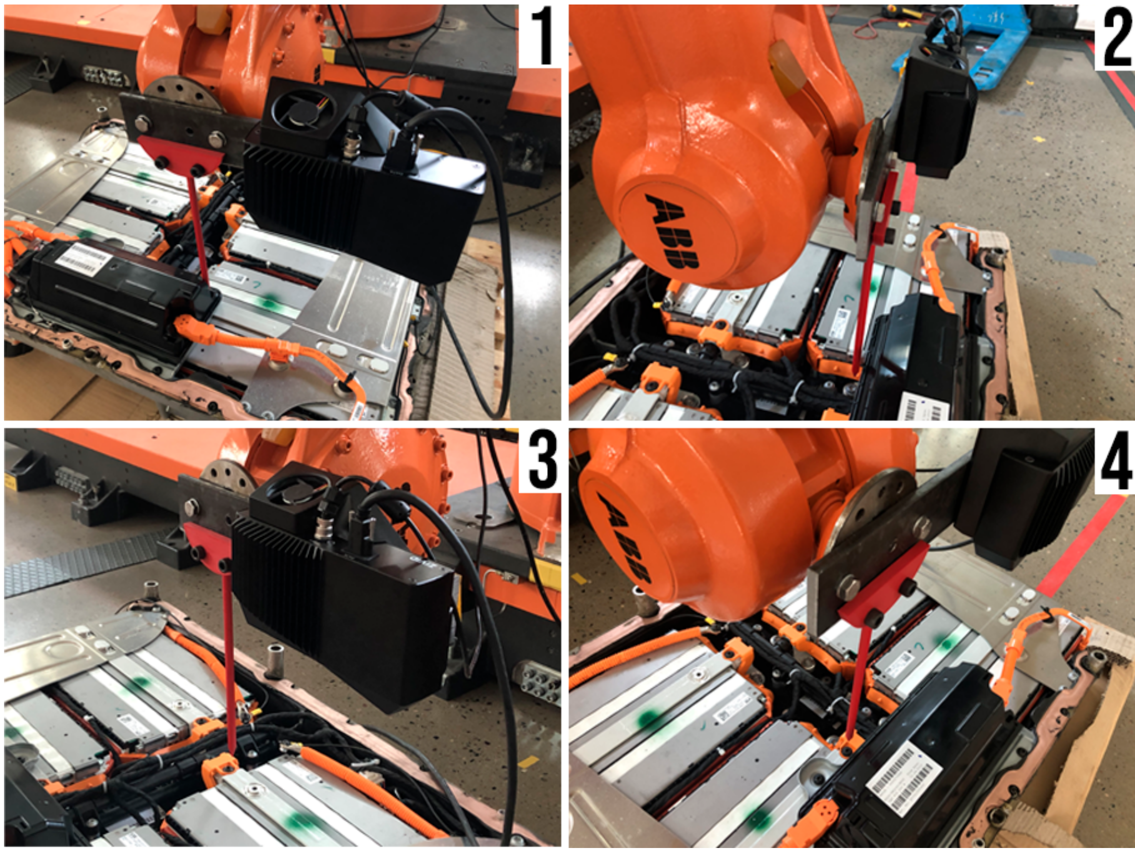


Figure 8.8: Accuracy tests

In order to show the system accuracy, the previous images have been zoomed in Figure 8.9. Thus, the positions 1-4 in Figure 8.9 correspond to the same positions in images 1-4 in Figure 8.8.

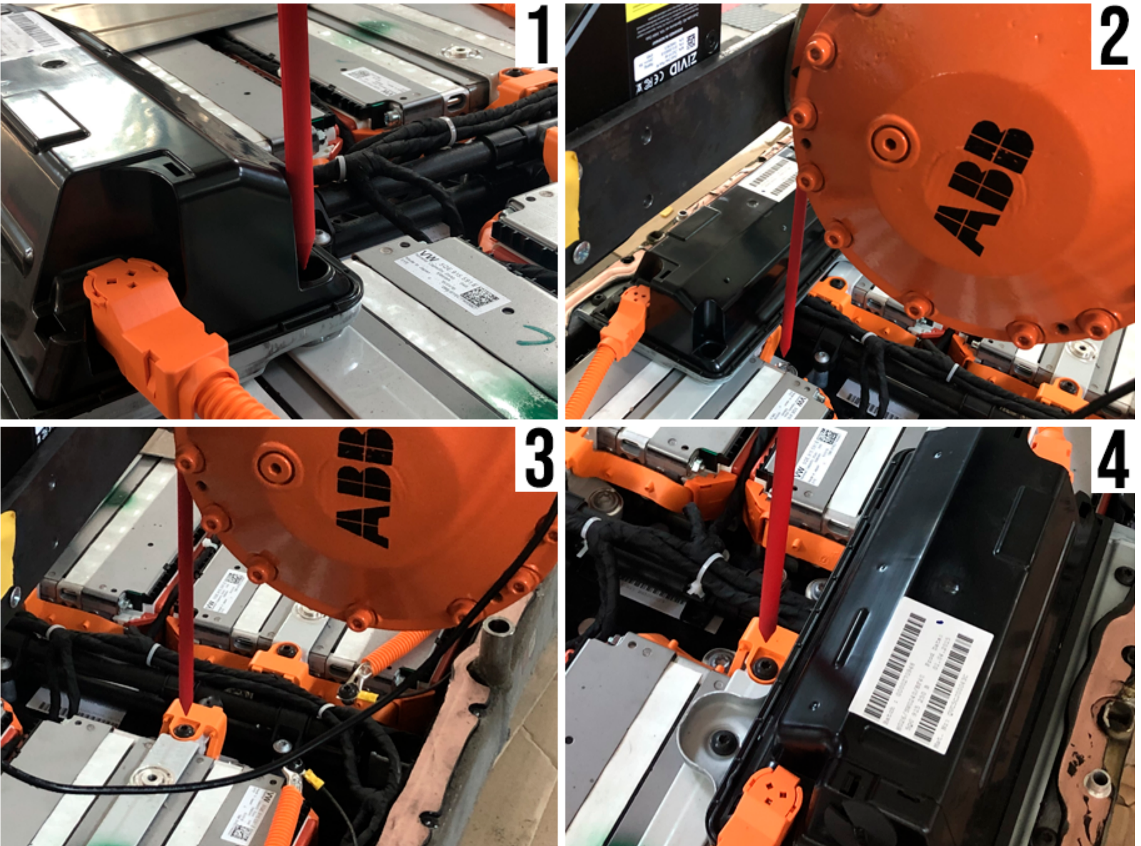


Figure 8.9: Detailed images: Accuracy tests

Chapter 9

Conclusions, future work and discussion

With the intention of proposing an approach to an automated operating LIB pack dismantling system, the main objectives of the project have been achieved. Thus, the fields of computer vision, robotics and battery disassembly have been successfully unified, resulting in a designed and tested automated disassembly system.

Following this purpose, the assigned main hardware elements (the robot, the 3D camera and the computer) have been interconnected to carry out the principal system tasks, such as object detection, pose estimation, decision-making and to move the robot. Therefore, the system is able to recognise the dismantling object main components, to find their position, and to move the robot to the defined positions in a specific order. Lab tests have been used to validate the designed task planner. In this case, the testing object has been a Volkswagen LIB pack.

Regarding the object detection part, the algorithm You Only Look Once is implemented. The principal contribution of the object detection part is to detect and find the components placed in the dismantling scene. The results show that the algorithm performs well, giving expected results and detecting the main components.

The information extracted from the object detection was used in pose estimation to find the centre point coordinates of the different components, where 2D images and the YOLO results have been matched with the 3D datasets.

The results obtained in the tests demonstrate that the obtained solution accomplish the fixed requirements and suggest that future research on this topic should follow in this direction.

The MoveIt! package for the group (ABB IRB4400 manipulator + IRBT4004 track) was created. This package has been used by the task planner to move the manipulator to the desired positions. Moreover, a manual to set the UiA Robotics Lab distributed ROS core and the ABB IRB 4400 robots has also been provided.

The results presented in this thesis cast a new light on the use of automation in the EV LIB batteries disassembly process. The experience in this field could also be adapted to be used for other dismantling processes and opens new doors and research challenges to other fields directly related to robotics like tool design.

Note that due to the Covid-19 situation, the initially assigned lab hours have been reduced significantly and this issue has directly affected the project's design and testing stages.

Future work:

The efforts in future stages of the project should be focused on instrumentation and tool design for the dismantling system, the improvement of the task planner and its capacity of dealing with different battery packs and continue the research in the object detection and pose estimation topics.

In addition, the system should be tested and trained for different LIB pack models and the robot should be run, always following the safety conditions, in auto mode. It is also important to consider a change of the robot speed for some operations, for example rapid motion.

In the future, in order to ensure a robust dismantling process, the task planner should be able to deal with different tool designs, therefore the final removal operation functions must be specially adjusted and integrated. Then, it is essential to detect flexible bodies such as wires and cables. Thus, the direction of the research on the object detection and pose estimation part should concern how to find a feasible solution for such the objects.

Discussion

Based on the experience achieved in this project, some comments, recommendations and discussions regarding operational time-saving, wires and the camera positioning in future project stages are given below.

- Time saving

In this project, the robot model is loaded every time when the task planner wants to move the robot. Thus, in future stages of the project the robot model should be loaded just once in the beginning. If this is done, the system will be at least 9.6 seconds faster.

Moreover, as explained in section 8.2, during all the project tests, the ROS main has been run in manual mode. Since the manual mode is running at a 25% of the maximum speed of the robot, in future stages of the project, all the movements will be faster (running in automatic mode). Figure 9.1 shows the process time based on a full speed. In addition, ROS 2 system might be considered instead of ROS presented in this report. Based on the author's knowledge, ROS 2 developers claimed it will be a real-time system.

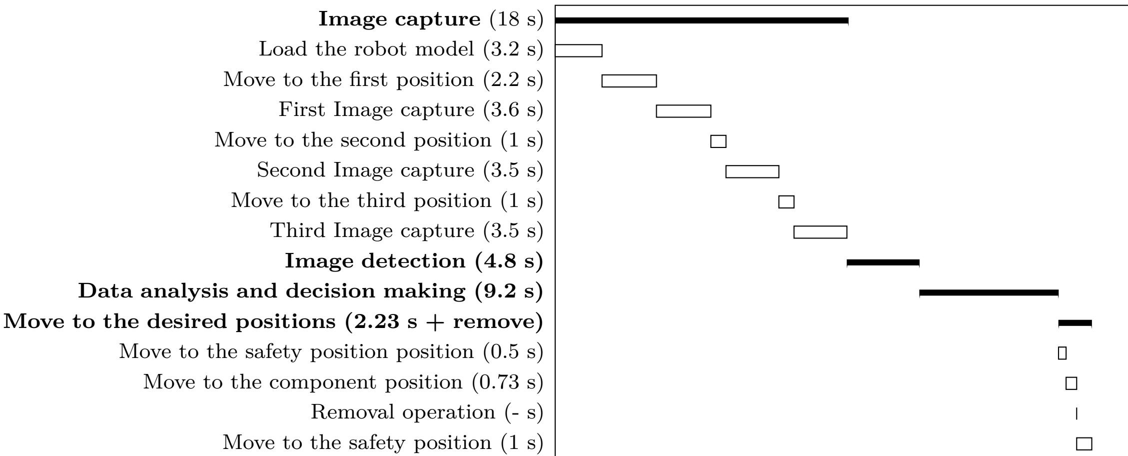


Figure 9.1: Desired task planner timings

- Wires

Wires detection is one of the main challenges for the future stages of the project and in computer vision in general. The future versions of the task planner should be able to handle wires, i.e. to cut them. To do this, the information required by the task planner to find the wires should be provided by a computer vision part. Thus, new algorithms should be implemented in the computer vision part.

A short-term solution to solve the wires problem would be a training the YOLO algorithm to detect electrical connections (see Figure 9.2), and cut the connections directly.

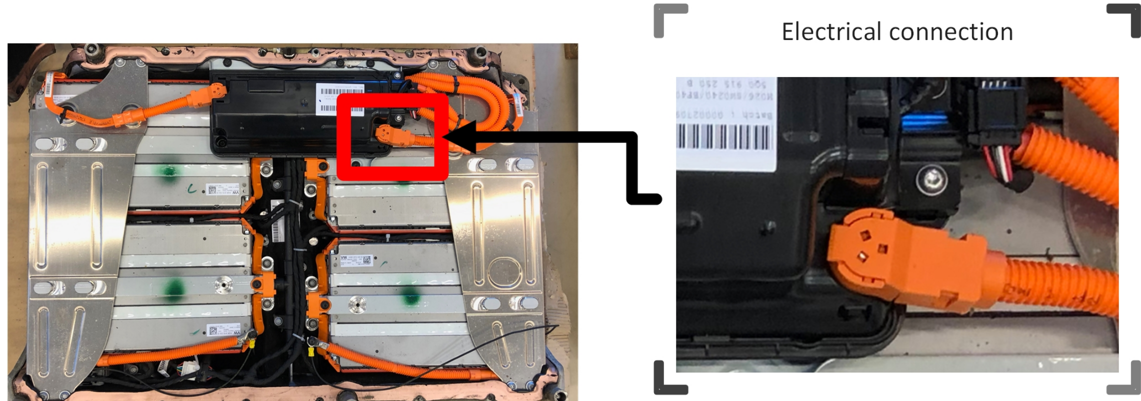


Figure 9.2: Electrical connection detail

- Camera positioning

In case of having access to more than one camera, a good option to consider is to place the cameras in fixed positions. Using an eye-in-hand configuration performs well, and has some advantages (i.e. only one camera is needed), but it has some disadvantages too. In case of implementing the methodology in the industry, the continuous moves and removal operations could have negative consequences like unexpected collisions of the camera with the environment, miscalibrations, etc...

Bibliography

[1] *A Guide to Understanding Battery Specifications*. http://web.mit.edu/evt/summary_battery_specifications.pdf. Accessed: 2020-02-05.

[2] Pilzecker A. et al. "Annual European Union greenhouse gas inventory 1990-2018 and inventory report 2020." In: *Copenhagen: European Environment Agency* (2020).

[3] *ABB IRB 4400*. https://new.abb.com/products/robotics/industrial-robots/irb-4400. Accessed: 2020-03-15.

[4] *Anchor Boxes for Object Detection*. https://se.mathworks.com/help/vision/ug/anchor-boxes-for-object-detection.html#:~:text=Anchor%20boxes%20are%20a%20set,sizes%20in%20your%20training%20datasets. Accessed: 2020-04-16.

[5] Daniel Apley et al. "Diagnostics in Disassembly Unscrewing Operations." In: *International Journal of Flexible Manufacturing Systems* 10 (Apr. 1998). DOI: 10.1023/A:1008089230047.

[6] Andreas Bihlmaier and Heinz Wörn. "Hands-on Learning of ROS Using Common Hardware." In: *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Ed. by Anis Koubaa. Cham: Springer International Publishing, 2016, pp. 29–50. ISBN: 978-3-319-26054-9. DOI: 10.1007/978-3-319-26054-9_2. URL: https://doi.org/10.1007/978-3-319-26054-9_2.

[7] Sachin Chitta. "MoveIt!: An Introduction." In: *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Ed. by Anis Koubaa. Cham: Springer International Publishing, 2016, pp. 3–27. ISBN: 978-3-319-26054-9. DOI: 10.1007/978-3-319-26054-9_1. URL: https://doi.org/10.1007/978-3-319-26054-9_1.

[8] A. D'Souza, S. Vijayakumar, and S. Schaal. "Learning inverse kinematics." In: *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180)*. Vol. 1. 2001, 298–303 vol.1.

[9] *Epoch vs Batch Size vs Iterations*. https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9. Accessed: 2020-05-25.

[10] Klaus Feldmann, Stefan Trautner, and Otto Meedt. "Innovative disassembly strategies based on flexible partial destructive tools." In: *Vaccine* 23 (Dec. 1999), pp. 159–164. DOI: 10.1016/S1367-5788(99)90079-2.

[11] James E. Gentle. "Matrix Transformations and Factorizations." In: *Matrix Algebra: Theory, Computations and Applications in Statistics*. Cham: Springer International Publishing, 2017, pp. 227–263. ISBN: 978-3-319-64867-5. DOI: 10.1007/978-3-319-64867-5_5. URL: https://doi.org/10.1007/978-3-319-64867-5_5.

[12] Serafim GuimarÃ£es et al. "Relation between the amount of smooth muscle of venous tissue and the degree of supersensitivity to isoprenaline caused by inhibition of catechol-O-methyl transferase." In: *Naunyn-Schmiedeberg's archives of pharmacology* 286 (Feb. 1975), pp. 401–12. DOI: 10.1007/BF00506654.

[13] Surendra M. Gupta and Charles R. McLean. "Disassembly of products." In: *Computers Industrial Engineering* 31.1 (1996). Proceedings of the 19th International Conference on Computers and Industrial Engineering, pp. 225–228. ISSN: 0360-8352. DOI: https://doi.org/10.1016/0360-8352(96)00146-5. URL: http://www.sciencedirect.com/science/article/pii/0360835296001465.

[14] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. 2nd ed. USA: Cambridge University Press, 2003. ISBN: 0521540518.

[15] IEA. *Global EV Outlook 2019: Scaling up the transition to electric mobility*. URL: https://www.iea.org/reports/global-ev-outlook-2019. (accessed: 2020-03-05).

[16] Ehud Kroll, Brent Beardsley, and Antony Parulian. "A Methodology to Evaluate Ease of Disassembly for Product Recycling." In: *IIE Transactions* 28.10 (1996), pp. 837–846. DOI: 10.1080/15458830.1996.11770736. eprint: https://doi.org/10.1080/15458830.1996.11770736. URL: https://doi.org/10.1080/15458830.1996.11770736.

[17] K. Masui et al. "Development of products embedded disassembly process based on end-of-life strategies." In: *Proceedings First International Symposium on Environmentally Conscious Design and Inverse Manufacturing*. 1999, pp. 570–575.

[18] H.S. Mok, H.J. Kim, and K.S. Moon. "Disassemblability of mechanical parts in automobile for recycling." In: *Computers Industrial Engineering* 33.3 (1997). Selected Papers from the Proceedings of 1996 ICCIC, pp. 621–624. ISSN: 0360-8352. DOI: https://doi.org/10.1016/S0360-8352(97)00207-6. URL: http://www.sciencedirect.com/science/article/pii/S0360835297002076.

[19] N. Otsu. "A Threshold Selection Method from Gray-Level Histograms." In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66.

[20] J. Pan, S. Chitta, and D. Manocha. "FCL: A general purpose library for collision and proximity queries." In: *2012 IEEE International Conference on Robotics and Automation*. 2012, pp. 3859–3866.

[21] R. Parasuraman, T. B. Sheridan, and C. D. Wickens. "A model for types and levels of human interaction with automation." In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 30.3 (2000), pp. 286–297.

[22] J. Redmon and A. Farhadi. "YOLO9000: Better, Faster, Stronger." In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 6517–6525.

[23] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement*. 2018. arXiv: 1804.02767 [cs.CV].

[24] *Robot Operating System Introduction*. http://wiki.ros.org/ROS/Introduction. Accessed: 2020-02-20.

[25] *Robot Operating System Master*. http://wiki.ros.org/Master. Accessed: 2020-02-202020-02-20.

[26] *Robot Operating System Messages*. http://wiki.ros.org/Messages. Accessed: 2020-02-20.

[27] *Robot Operating System Nodes*. http://wiki.ros.org/Nodes. Accessed: 2020-02-20.

[28] *Robot Operating System Topics*. http://wiki.ros.org/Topics. Accessed: 2020-02-20.

[29] J. Schmitt et al. "Disassembly automation for lithium-ion battery systems using a flexible gripper." In: *2011 15th International Conference on Advanced Robotics (ICAR)*. 2011, pp. 291–297.

[30] G. Seliger et al. "Flexible disassembly tools." In: Feb. 2001, pp. 30–35. ISBN: 0-7803-6655-7. DOI: 10.1109/ISEE.2001.924498.

[31] *Semantic Robot Description Format (SRDF)*. http://wiki.ros.org/srdf. Accessed: 2020-03-10.

[32] *Socket Interface*. https://www.sciencedirect.com/topics/computer-science/socket-interface. Accessed: 2020-04-02.

[33] Peter Sturm. "Pinhole Camera Model." In: *Computer Vision: A Reference Guide*. Ed. by Katsushi Ikeuchi. Boston, MA: Springer US, 2014, pp. 610–613. ISBN: 978-0-387-31439-6. DOI: 10.1007/978-0-387-31439-6_472. URL: https://doi.org/10.1007/978-0-387-31439-6_472.

[34] I. A. Sucan, M. Moll, and L. E. Kavraki. "The Open Motion Planning Library." In: *IEEE Robotics Automation Magazine* 19.4 (2012), pp. 72–82.

[35] Wei Hua Chen Supachai Vongbunyong. *Disassembly Automation. Automated Systems with Cognitive Abilities.*

[36] *The PCD (Point Cloud Data) file format*. https://vml.sakura.ne.jp/koeda/PCL/tutorials/html/pcd_file_format.html. Accessed: 2020-03-17.

[37] Kathrin Wegener et al. "Disassembly of Electric Vehicle Batteries Using the Example of the Audi Q5 Hybrid System." In: *Procedia CIRP* 23 (Dec. 2014). DOI: 10.1016/j.procir.2014.10.098.

[38] Kathrin Wegener et al. "Robot Assisted Disassembly for the Recycling of Electric Vehicle Batteries." In: vol. 29. Apr. 2015. DOI: 10.1016/j.procir.2015.02.051.

[39] A. Weigl-Seitz et al. "On strategies and solutions for automated disassembly of electronic devices." In: *The International Journal of Advanced Manufacturing Technology* 30 (Sept. 2006), pp. 561–573. DOI: 10.1007/s00170-005-0043-8.

[40] *What is .npy files and why you should use themâ*. https://towardsdatascience.com/what-is-npy-files-and-why-you-should-use-them-603373c78883. Accessed: 2020-03-25.

[41] *You Only Look Once alogrithm*. https://medium.com/@ODSC/overview-of-the-yolo-object-detection-algorithm-7b52a745d3e0. Accessed: 2020-03-15.

[42] Yin Zhou and Oncel Tuzel. *VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection*. 2017. arXiv: 1711.06396 [cs.CV].

[43] *Zivid official documentation*. http://www.zivid.com/downloads. Accessed: 2020-03-25.

[44] *Zivid: Introduction to stops*. https://zivid.atlassian.net/wiki/spaces/ZividKB/pages/98763232/Introduction+to+Stops. Accessed: 2020-04-25.

Appendix A

Hardware Specifications

A.1 Zivid 3D Camera

	Small	Medium	Large
Optimal Range (m)	0.3 - 0.8	0.6 - 1.6	1.2 - 2.6
Maximum Range (m)	1.0	2.0	3.0
FOV (mm)	170 x 140 @ 0.3m 650 x 480 @ 1.0m	420 x 270 @ 0.6m 1370 x 900 @ 2.0m	850 x 530 @ 1.2m 2110 x 1360 @ 3.0m
Spatial Resolution (mm)	0.12 @ 0.3m 0.40 @ 1.0m	0.23 @ 0.6m 0.75 @ 2.0m	0.45 @ 1.2m 1.11 @ 3.0m
Precision / z-noise (mm)	0.03 @ 0.3m <0.2 @ 1.0m	0.07 @ 0.6m <1.0 @ 2.0m	0.3 @ 1.2m <2.0 @ 3.0m
Acquisition Rate	13 Hz (full resolution)		
Output	3D (XYZ) + Color (RGB) + Quality (Q) for each pixel		
Image Size	1920 x 1200 (2.3 Mpixel)		
Imaging	Color, 3D HDR, Reflection and noise filters		
Simplicity	Factory-calibrated 3D camera, easy and intuitive GUI Zivid Studio, Modern and high level API.		
Software APIs	C++ / C# / .NET GeniCam/HALCON (RC)		
OS	Windows 7 / 8 / 10 Ubuntu 16.04 / Ubuntu 18.04		
Physical Interface	USB3.0 (5m / 10m / 25m options) 24V DC		
Dimensions and weight	226 x 165 x 86 mm 2 kg		
Environmental	10-40°C / 5G Sinus / 25G Shock IP65 aluminum housing		
Safety and EMC	CE / EN60950 / CB / FCC class A		

All product, product specifications and data are subject to change without notice to improve reliability, function or design or otherwise.

zivid.com

A.2 IRB4400

Specification

Robot version	Reach (m)	Handling capacity (kg)
IRB 4400/60	1.96	60
IRB 4400/L10	2.53	10
Supplementary load		
on axis 2	35 kg	
on axis 3	15 kg	
on axis 4	0-5 kg	
Number of axes	6	
Protection	Standard version IP 54, Foundry Plus 2 IP 67 and high pressure steam washable	
Mounting	Floor	
Controller	IRC5 Single Cabinet	
Integrated signal supply	23 signals and 10 power on upper arm	
Integrated air supply	Max. 8 bar on upper arm	

Performance (according to ISO 9283)

	Position repeatability	Path repeatability*
IRB 4400/60	0.06 mm	0.09 mm
IRB 4400/L10	0.05 mm	0.16 mm

*At 1.6 m/s.

Technical information

Electrical Connections	
Supply voltage	200-600 V, 50/60 Hz
Rated power transformer rating	7.8 kVA
Physical	
Robot base	920 x 640 mm
Robot weight	1040 kg
Environment	
Ambient temperature for mechanical unit	
During operation	+5° C (41° F) to + 45°C (113°F)
Relative humidity	Max. 95%
Noise level	Max. 70 dB (A)
Safety	Double circuits with supervision, emergency stops and safety functions, 3-position enable device
Emission	EMC/EMI-shielded

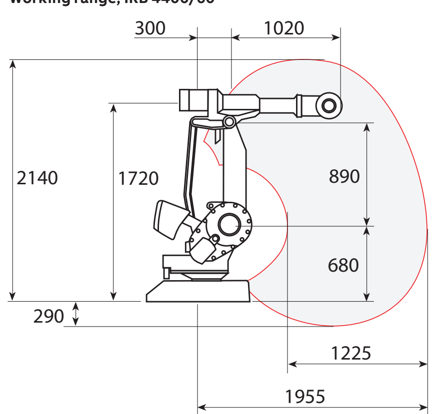
Data and dimensions may be changed without notice.

Movement

Axis movement	Working range	Axis max speed IRB 4400/60	Axis max speed IRB 4400/L10
Axis 1, Rotation	+165° to -165°	150°/s	150°/s
Axis 2, Arm	+95° to -70°	120°/s	150°/s
Axis 3, Arm	+65° to -60°	120°/s	150°/s
Axis 4, Rotation	+200° to -200°	225°/s	370°/s
Axis 5, Bend	+120° to -120°	250°/s	330°/s
Axis 6, Turn	+400° to -400° Max. rev: +200°[1] to -200[2]	330°/s	381°/s

[1] Max. rev: +183 to -183 valid for IRB 4400/L10
[2] The default working range for axis 6 can be extended by changing parameter values in the software.
There is a supervision function to prevent overheating in applications with intensive and frequent movements.

Working range, IRB 4400/60



Working range, IRB 4400/L10

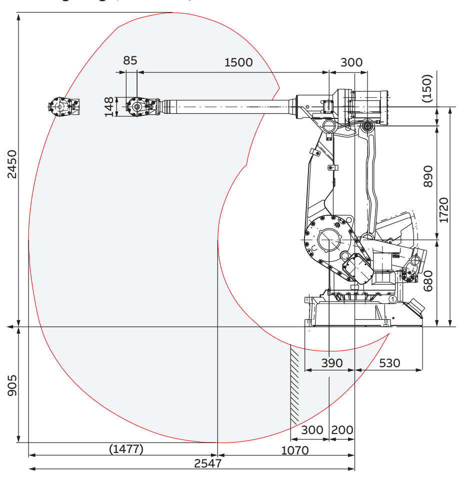


abb.com/robotics

We reserve the right to make technical changes or modify the contents of this document without prior notice. With regard to purchase orders, the agreed particulars shall prevail. ABB does not accept any responsibility whatsoever for potential errors or possible lack of information in this document.

We reserve all rights in this document and in the subject matter and illustrations contained therein. Any reproduction, disclosure to third parties or utilization of its contents – in whole or in parts – is forbidden without prior written consent of ABB.
Copyright© 2019 ABB
All rights reserved

PR10035EN_R8 Rev.F April 2019

A.3 IRB4400

	Robot	Travel length	No of robots	Mounting pos
IRBT 4004	IRB 4400/ 4450S/4600	1.9... 19.9 m/1 m step	One or two/track	Floor
IRBT 6004/ 7004	IRB 6620/6650S/6700	1.7... 19.7 m/1 m step	One or two/track	Floor
IRBT 7004	IRB 7600	1.7... 19.7 m/1 m step	One or two/track	Floor

Cable arrangement
Plastic with cover - standard

Pos to Pos time (s) *)	1 m	2 m	3 m	4 m	5 m
IRBT 4004	< 1.2	< 1.7	< 2.2	< 2.7	< 3.2
IRBT 6004	< 1.5	< 2.1	< 2.8	< 3.4	< 4.0
IRBT 7004	< 1.7	< 2.6	< 3.4	< 4.2	< 5.0

*) With max load

Acceleration/Retardation (m/s2) X004	
IRB 4004	2.5*
IRB 6004	2.0*
IRB 7004	1.8*

* Dep. on actual load

Speed (m/s)	
IRB 4004	2.0
IRB 6004	1.6
IRB 7004	1.2

© Copyright ABB Robotics. PR10335EN_R3 Aug 2016.

For more information please contact:

ABB AB
Robotics
Hydrovägen 10
SE-721 36 Västerås, Sweden
Phone: +46 21 325000

www.abb.com/robotics

Note
We reserve the right to make technical changes or modify the contents of this document without prior notice. With regard to purchase orders, the agreed particulars shall prevail. ABB does not accept any responsibility whatsoever for potential errors or possible lack of information in this document.

We reserve all rights in this document and in the subject matter and illustrations contained therein. Any reproduction, disclosure to third parties or utilization of its contents - in whole or in parts – is forbidden without prior written consent of ABB.

Copyright© 2016
ABB All rights reserved

Appendix B

Code Documentation

B.1 [main()]: Task planner core

```
def main():
    i=0
    #Declaration of the variables
    screw_num=0
    connect_num=0
    screw_pos=[]
    connect_pos=[]
    compon_pos=[]
    co_tot=[]
    #rem_success is the responsible of ending the disasssembly loop in case of detecting 0
        components in the image
    rem_success='None'
    removed_components=[]
    names_file= open('/yolov3/data/LIBRES_classes_1.names','r')
    names_file_lines=names_file.readlines()
    Dict_counter=0
    #adds the classes names in the dictionary
    for names in names_file_lines:
        Dict_comp[Dict_counter]=names
        Dict_counter=Dict_counter+1
    Dict_counter=0

    #Disassembly loop
    while rem_success!='Goal state reached':

        # Move the robot to take pictures (takes the pictures and saves them)
        print('=================IMAGE ACQUISITION=================')
        if opt.take_images=='True':
            os.system('rosrun lefty_move move_to_pict3.py')
        #analyses the image using the YOLOv3 algorithm (Object detection).
        print('=================IMAGE DETECTION=================')
        if opt.detect=='True':
            os.system('python3 /yolov3/detect.py --save-txt')
        if opt.autotrain=='True':
```

```
    training_data_autogenerator()
print('=================PLANNING REMOVAL=================')
#For loop running through the 3 images
for num_im in range(3):
    #num_im is the image number (there are 3 images 0.jpg, 1.jpg and 2.jpg)

    (screw_num,connect_num,compon_num,empty_screw_num, screw_pos, connect_pos,
        compon_pos, empty_screw_pos, co_tot) = num_components(Dict_comp,
        num_im)
    rem_component=[]

    #obtains the components positions (camera reference base frame) for all the images
    if num_im==0:
        print('=================Analysing the first image positions
            =================')
        rem_component=screw_pos
        # rem_component_cam_0 contains the positions in camera reference of the screws
            found in the first image
        rem_component_cam_0=CamR_pos(rem_component,Dict_comp,num_im)
    if num_im==1:
        print('=================Analysing the second image positions
            =================')
        #desition-making
        rem_component=what_component(screw_num,connect_num,compon_num,
            empty_screw_num, screw_pos, connect_pos,compon_pos, empty_screw_pos,
            co_tot)
        # rem_component_cam_1 contains the positions of the components in order of
            removal.
        rem_component_cam_1=CamR_pos(rem_component,Dict_comp,num_im)
    if num_im==2:
        print('=================Analysing the third image positions
            =================')
        rem_component=screw_pos
        # rem_component_cam_0 contains the positions in camera reference of the screws
            found in the last image
        rem_component_cam_2=CamR_pos(rem_component,Dict_comp,num_im)

print('=================Merging results=================')
# Merge the results obtained in the previous step (Returns the list of the positions in
    world reference base frame)
rem_component_final=merge_detection(rem_component_cam_0,
    rem_component_cam_1,rem_component_cam_2)

# Removes all the components
print('=================REMOVING COMPONENTS
    =================')
for rem_specif in rem_component_final:
    #remove function (main part to fulfill in future stages of the project)
    rem_success=remove(rem_specif,Dict_comp)
    if rem_success:
        #Array containing all the components succesfully removed.
```

```
            removed_components.append(rem_specif)

        #if the rem_component list is == [] Goal state reached
        if rem_component_final==[]:
            rem_success='Goal state reached'
    #call the next battery
```

B.2 [CamR_pos()]: Find the positions in camera reference base frame

```
# The main purpose of camR_pos is to find the positions in camera base frame
def CamR_pos(rem_component,Dict_comp,num_im):
    #Finds the position of the components in camera reference base frame
    print('=======pointcloud analysis=======')
    #Loads the depth image
    depth_im=np.load('/yolov3/data/depth/depth'+str(num_im)+'.npy')
    (rows,cols) = depth_im.shape
    # rem_component_cam is an array containing values of the postition of the different
        components in camera reference
    rem_component_cam=[]

    #Intrinsic parameters definition
    fx=2765.132568359375
    fy=2764.61767578125
    fx_inv=1/fx
    fy_inv=1/fy
    cx=946.9010620117188
    cy=579.4774780273438

    #Runs the results of object detection analysing the detected labels.
    for comp in rem_component:
        # sq contains the position in pixels in the 2D image (square two corners) [x1,y1,x2,y2]
        sq=comp[0:4]
        counter=0
        # Means (in x-axis, y-axis, z-axis)
        x_mean=0
        y_mean=0
        z_mean=0
        n=0
        # Converts sq to int list (was an string)
        while n<=3:
            sq[n]=int(sq[n])
            n=n+1
        sq_1=sq[0:2]
        sq_2=sq[2:4]
        # Finds a mean of the pixels positions (inside the label)
        for i in range(sq_1[0],sq_2[0]):
            for e in range(sq_1[1],sq_2[1]):
```

```
                # Using the intrinsic parameters fins (x,y,z)
                z=depth_im[e,i]
                x=(i-cx)*z*fx_inv
                y=(e-cy)*z*fy_inv
                if z>=0.5 and z<=20:
                    z_mean=z_mean+z
                    x_mean=x_mean+x
                    y_mean=y_mean+y
                    counter=counter+1
        if counter>0:
            x_mean=x_mean/counter
            y_mean=y_mean/counter
            z_mean=z_mean/counter
            pos=[x_mean,y_mean,z_mean,comp[4]]
            rem_component_cam.append(pos)
    print('-Pointcloud analysed-')
    #Returns the detections converted.
    return rem_component_cam
```

B.3 [WR_pos]: World reference base frame position detection:

```
# Converts to World reference base frame the array containing the positions in camera reference
    base frame
def WR_pos(rem_component_cam,num_im):
    pos=[]
    # Converts the list to float
    for e in rem_component_cam[0:3]:
        pos.append(float(e))
    pos.append(1)
    # Loads the transform matrix containing the camera position at the moment of the image
        capturing
    tf_matrix=np.load('/yolov3/data/transf_matrix/matrix'+str(num_im)+'.npy')
    pos=np.matrix(pos)
    pos=pos.transpose()
    tf_matrix=np.matrix(tf_matrix)
    wr_pos=tf_matrix*pos
    #wr_pos=[[x],[y],[z]]
    #transforms to a list wr_pos_def=[x,y,z]
    wr_pos_def=[]
    for e in wr_pos:
        wr_pos_def.append(float(e[0]))

    wr_pos_def[3]=rem_component_cam[3]

    # Returns the list containing the wr positions
    return wr_pos_def
```

B.4 [remove()]: Removal Operation

```
# This function aims to move the robot to the desired positions (in future stages of the
    project it should be fulfilled with the necessary code)
def remove(pos_comp,Dict_comp):
    print('=======Remove operation=======')
    #removal operation depending on the component.
    #pso_comp=[x,y,z,mod]
    print(str(pos_comp))
    #if the component is an screw
    if pos_comp[3]=='3':
        #remove operation for screwprint('moving to a safety position')
        x=pos_comp[0]
        y=pos_comp[1]
        z=pos_comp[2]
        # Move the robot to the desired position
        os.system('rosrun lefty_move move_pos_1.py --x '+str(x)+' --y '+str(y)+' --z '+
            str(z))
        #PLACE THE REMOVAL FUNCTION FOR SCREWS HERE!
        print('removing screw')
        if opt.debug=='True':
            input("Press Enter to move the robot to the next position ... ")
        print('moving to a safety position')
        x=pos_comp[0]
        y=pos_comp[1]
        z=pos_comp[2]+0.4
        os.system('rosrun lefty_move move_pos.py --x '+str(x)+' --y '+str(y)+' --z '+str(
            z))

    #General removing operation (rest of the components)
    else:
        print('Moving robot to the removal position')
        x=pos_comp[0]
        y=pos_comp[1]
        z=pos_comp[2]
        print(y)
        os.system('rosrun lefty_move move_pos.py --x '+str(x)+' --y '+str(y)+' --z '+str(
            z))
        print('removing'+Dict_comp[pos_comp[3]])
        #PLACE THE REMOVAL FUNCTION HERE!
        if debug=='True':
            input("Press Enter to move the robot to the next position ... ")
        time.sleep(1.0)
        print('moving to a safety position')
        x=pos_comp[0]
        y=pos_comp[1]
        z=pos_comp[2]+0.3
        os.system('rosrun lefty_move move_pos.py --x '+str(x)+' --y '+str(y)+' --z '+str(
            z))
        return True
```

B.5 [num_components]: Components analysis

```
#counts the number of detected components
def num_components(Dict_comp,num_im):
    print('=======Component analysis=======')
    # Reads txt file
    f= open('/yolov3/output/'+str(num_im)+'.jpg.txt','r')
    f1= f.readlines()
    # Variable to know if there are screws or connection components
    screw_num=0
    empty_screw_num=0
    # Number of connective components
    connect_num=0
    compon_num=0
    #position of the screws and connect_num
    connect_pos=[]
    screw_pos=[]
    compon_pos=[]
    empty_screw_pos=[]
    # Empty array with all the components detected in the image (screws not included)
    co_tot=[]
    # For component line in file
    for co_lin in f1:
        co_lin= co_lin.split()
        #co_name=name of the component (screw, VW_modules, etc..)
        co_name=Dict_comp[int(co_lin[4])]
        # The variable pos contains the position in the image and the class pos=[x,y,z,c]
        pos=co_lin[0:5]
        if co_name[0:5]=='screw':
            screw_pos.append(pos)
            #there is a screw
            screw_num=screw_num+1

        elif co_name[0:10]=='connection':
            #there is a connection component
            connect_pos.append(pos)
            connect_num = connect_num + 1
            co_tot.append(co_lin)
        elif co_name[0:5]=='empty':
            #there is an empty screw
            empty_screw_pos.append(pos)
            empty_screw_num = empty_screw_num + 1

        else:
            co_tot.append(co_lin)
            compon_pos.append(pos)
            compon_num=compon_num+1
    print('there are '+ str(screw_num)+ ' screws')
    print('there are '+ str(connect_num)+ ' connective components')
    print('there are '+ str(compon_num)+ ' components (modules, bms, etc..)')
    print('there are '+ str(empty_screw_num)+ 'empty screws')
```

```
    return (screw_num,connect_num,compon_num,empty_screw_num, screw_pos,
        connect_pos,compon_pos, empty_screw_pos, co_tot)
```

B.6 [what_component and has_comp_over]: Decision-making

```
#Responsible of the desition−making
def what_component(screw_num,connect_num,compon_num,empty_screw_num, screw_pos,
    connect_pos,compon_pos, empty_screw_pos, co_tot):
#decide what component should be removed next
    #creation of the variables with the positions of the components that can be removed
    print('=======What components should be removed=======')
    connect_rem=[]
    screw_num=[]
    compon_rem=[]
    co_an=[]
    com_rem_other=[]
    # The screws are always the first to be removed
    if screw_num!=0:
        compon_rem=screw_pos
        print('screws should be removed')

    # Analyses the rest of components using the computer vision based function has_comp_over
    for comp in co_tot:
        comp.append(has_comp_over(co_tot,comp))
        com_rem_other.append(comp)
    # Sorts the list depending on the "probabilities" returned by the function has_comp_over
    com_rem_other.sort(key=itemgetter(6))
    for comp in com_rem_other:
        compon_rem.append(comp)
    #returns the components in removal order
    return compon_rem


# Given a certain component (in this case co_an) returns the probability of the different
    components of having a component over
def has_comp_over(co_tot,co_an):
    comp=co_an
    i=0
    # Converts to integer
    while i<5:
        comp[i]=int(comp[i])
        i=i+1
    p_list=[]

    #Runs over the list containg all the components
    for comp2_str in co_tot:
```

```
comp2=[]
inside=False
comp2_str=comp2_str[0:5]
#convert to int
for e in comp2_str:
    comp2.append(int(e))
# Looks if comp and comp2 intersect in the image
for x in range(comp2[0],comp2[2]):
    for y in range(comp2[1],comp2[3]):
        if x>comp[0] and x<comp[2] and y>comp[1] and y<comp[3]:

            inside=True
            break
        if inside==True:
            break
#if they intersect
if inside==True:
    #reads the image
    img_0 =cv2.imread('/yolov3/data/samples/1.jpg');
    #window in the area of comp
    img_1= img_0[comp[1]:comp[3],comp[0]:comp[2]]
    img_grey = cv2.cvtColor(img_0, cv2.COLOR_BGR2GRAY)
    #find the intersection area label
    area=[0,0,0,0]
    #first the x1 and x2 values
    if comp2[0]<comp[0]:
        area[0]=comp[0]
    else:
        area[0]=comp2[0]

    if comp2[2]<comp[2]:
        area[2]=comp2[2]
    else:
        area[2]=comp[2]
    #find the y1 and y2 values
    if comp2[1]<comp[1]:
        area[1]=comp[1]
    else:
        area[1]=comp2[1]
    if comp2[3]<comp[3]:
        area[3]=comp2[3]
    else:
        area[3]=comp[3]

    img_2=img_0[area[1]:area[3],area[0]:area [2]]
    #equalize the full image
    img_equ = cv2.equalizeHist(img_grey)
    #window in the area of interest (the component)
    img_3=img_equ[comp[1]:comp[3],comp[0]:comp[2]]
    #Binarization
```

```
        ret2,thresh1 = cv2.threshold(img_3,0,255,cv2.THRESH_BINARY+cv2.
            THRESH_OTSU)
        #intersection label coordinates
        area_im=[area[0]-comp[0],area[1]-comp[1],area[2]-comp[0],area[3]-comp[1]]
        #window in the intersection
        intersection_image=thresh1[area_im[1]:area_im[3],area_im[0]:area_im[2]]
        #intersection window area
        (intersection_x, intersection_y) =intersection_image.shape
        intersection_area=intersection_x*intersection_y
        #main window area calculation
        (comp_x, comp_y) =thresh1.shape
        comp_area=comp_x*comp_y
        #calculation of the mean of the pixels of the window excluding the intersection pixels
        intersection_t=intersection_area*intersection_image.mean()
        comp_t=comp_area*thresh1.mean()
        mean_exc=(comp_t-intersection_t)/(comp_area-intersection_area)
        #percentage intersection, and the rest of the area p_exc
        p_int=intersection_image.mean()/255
        p_exc=mean_exc/255
        p_max=max(p_int,p_exc)
        p_min=min(p_int,p_exc)
        #differences of percentage
        p_difference=p_max-p_min
        p_list.append(p_difference)
    if len(p_list)==0:
        p_list.append(0.0)
    #returns the maximum percentage for each component
    return max(p_list)
```

B.7 [training_data_autogenerator()]: Automatic YOLO training data creator

```
#Use and covert the detections to create new training images
def training_data_autogenerator():
    #load the txt (containing the detections) and the image
    img_0 =cv2.imread('/yolov3/data/samples/2.jpg');
    txt_f= open('/yolov3/output/2.jpg.txt','r')
    lines= txt_f.readlines()
    Dict={}
    #image shape
    (y_size,x_size,RGB)=img_0.shape
    line_YOLO_tot=[]
    #run over the lines
    for line in lines:
        line=line. split ()
        # Proportional label sizes x_dif (width), y_dif (height), and label centres x_1, y_1
        x_dif_p=int(line[2])-int(line[0])
        y_dif_p=int(line[3])-int(line[1])
        x_1=(float(line[0])+float(line [2]) )/(2*x_size)
```

```
        x_dif=float(x_dif_p)/x_size
        y_1=(float(line[1])+float(line [3]) )/(2*y_size)
        y_dif=float(y_dif_p)/y_size
        # Array containing the infromation of a yolo line
        line_YOLO=[line[4],round(x_1,6),round(y_1,6),round(x_dif,6),round(y_dif,6)]
        line_YOLO_tot.append(line_YOLO)
    #Sorts the lines depending on the component
    line_YOLO_tot.sort(key=itemgetter(0))

    #Path definition and txt creation
    path, dirs, files = next(os.walk('/more_trai/autocreated_training_images/'))
    file_count = len(files)
    num=int(file_count/2)
    YOLO_txt_file= open('/more_trai/autocreated_training_images/'+str(num)+'.txt','w')

    # Write the lines
    for line_YOLO_w in line_YOLO_tot:
        i=0
        for e in line_YOLO_w:
            line_YOLO_w[i]=str(line_YOLO_w[i])
            i=i+1

        YOLO_txt_file.write(" ".join(line_YOLO_w) + "\n")

    #add the path to the data file.
    autogenerated_data=open('/yolov3/data/autogenerated_data.txt','w')
    autogenerated_data.write('/more_trai/autocreated_training_images/'+str(num)+'.jpg'+"\n
        ")
    cv2.imwrite('/more_trai/autocreated_training_images/'+str(num)+'.jpg', img_0)
    #close all the files
    YOLO_txt_file.close()
    autogenerated_data.close()
```

B.8 [merge_detection()]: Merge detection

```
#merge_detection
def merge_detection(rem_component_cam_0,rem_component_cam_1,
      rem_component_cam_2):
    #rem_component_cam_1,(rem_component_Cam_1) contains the list with all the
        components screws+other components
    #the aim of the function is to merge the screw detection of the three images (taken with
        different inclinations)
    rem_component_WR_0=[]
    rem_component_WR_1=[]
    rem_component_WR_2=[]
    rem_component_WR=[]
    rem_component_WR_filt=[]
    num_rep=0
    #obtain the WR frame coordinates of the components (for each image)
```

```
print('--Finding the WR frame coordinates for each image--')
for num_im in range(3):
    if num_im==0:
        print('-First iamge-')
        for rem_specif in rem_component_cam_0:
            pos_comp= WR_pos(rem_specif,num_im)
            rem_component_WR_0.append(pos_comp)
    if num_im==1:
        print('-Second iamge-')
        for rem_specif in rem_component_cam_1:
            pos_comp= WR_pos(rem_specif,num_im)
            rem_component_WR_1.append(pos_comp)
    if num_im==2:
        print('-Second iamge-')
        for rem_specif in rem_component_cam_2:
            pos_comp= WR_pos(rem_specif,num_im)
            rem_component_WR_2.append(pos_comp)

#Appends all the postions into rem_component_WR
for component in rem_component_WR_0:
    rem_component_WR.append(component)

for component in rem_component_WR_2:
    rem_component_WR.append(component)

for component in rem_component_WR_1:
    rem_component_WR.append(component)

#Merges the closer positons (<1cm)
print('--merging positions--')
# For loop runing over rem_component_WR
for component in rem_component_WR:
    repeated=False
    x_sum=component[0]
    y_sum=component[1]
    z_sum=component[2]
    count=1
    #For loop comparing the components already in the filtered list with the component
        being analysed
    for component_2 in rem_component_WR_filt:
        x_dif=abs(component[0]-component_2[0])
        y_dif=abs(component[1]-component_2[1])
        z_dif=abs(component[2]-component_2[2])
        #Checks if the differences between both positions are lower than 1cm
        if x_dif <=0.01 and y_dif<=0.01 and component[3]==component_2[3]:
            repeated=True
            num_rep=num_rep+1
            x_sum=x_sum+component_2[0]
            y_sum=y_sum+component_2[1]
            z_sum=z_sum+component_2[2]
            count=count+1
```

```
        #if the component was not in the list
        if repeated==False:
            rem_component_WR_filt.append(component)
        #if the component was in the list
        if repeated==True:
            component_mean=[x_sum/float(count),y_sum/float(count),z_sum/float(count),
                component[3]]
            rem_component_WR_filt.append(component_mean)

    #returns the definitive array containing the components positions
    return rem_component_WR_filt
```

B.9 [move_to_pict.py]: Image capturing

```
#!/usr/bin/env python

import sys
import os
import copy
import rospy
import tf
import numpy as np
import dynamic_reconfigure.client
from zivid_camera.srv import *
from sensor_msgs.msg import PointCloud2
import cv2
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import time
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
from math import pi
from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list
from sensor_msgs import point_cloud2
import pcl
from tf.transformations import quaternion_matrix

#from pyntcloud import PyntCloud

def main():
#intialize movitcommander and the node
    im_num=0
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('robot_irb', anonymous=True)
#Instantiate a RobotCommander object. Provides information
    s = Sample_01()
    s.im_num=0
```

```
    robot = moveit_commander.RobotCommander()
#Instantiate a PlanningSceneInterface object. This provides a remote interface for getting,

    scene = moveit_commander.PlanningSceneInterface()

#define the groupname
    group_name = 'righty_tcp'
    move_group = moveit_commander.MoveGroupCommander(group_name)
    move_group.set_planner_id('RRTconnect')
    move_group.set_goal_position_tolerance(0.0001)
    move_group.set_goal_orientation_tolerance(0.0001)

    #define the Publisher
    display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',
                                                   moveit_msgs.msg.DisplayTrajectory,
                                                   queue_size=20)

    print "====== Moving robot to the first image pose ======"

    print move_group.get_current_pose()
    print move_group.get_current_rpy()
    #pose goal in quaternions
    pose_goal = geometry_msgs.msg.Pose()
    #pose_goal = move_group.get_random_pose()
    q = tf.transformations.quaternion_from_euler(-0.656599232586658, 1.396548903822593,
        -0.6)
    #q=[ 0, 0.6085625, 0.0405708, 0.7924681 ]
    pose_goal.orientation.x = q[0]
    pose_goal.orientation.y = q[1]
    pose_goal.orientation.z = q[2]
    pose_goal.orientation.w = q[3]
    pose_goal.position.x = 3.65
    pose_goal.position.y = 6.65
    pose_goal.position.z = 0.9
    # Move the robot to the first postion
    move_group.set_pose_target(pose_goal)

    plan = move_group.go(wait=True)
    # Calling `stop()` ensures that there is no residual movement
    move_group.stop()


    # Call the capture service (to obtain the first picture)

    s.capture()
    print('sleeping')
    time.sleep(1.5)

    print "====== Moving robot to the second image pose ======"
    move_group.clear_pose_targets()
```

```
    q = tf.transformations.quaternion_from_euler(0.21155554664879492, 1.2930785482720229,
        0.2258100592267419)
    #q=[ 0, 0.6085625, 0.0405708, 0.7924681 ]
    pose_goal.orientation.x = q[0]
    pose_goal.orientation.y = q[1]
    pose_goal.orientation.z = q[2]
    pose_goal.orientation.w = q[3]
    pose_goal.position.x = 3.55
    pose_goal.position.y = 6.4
    pose_goal.position.z = 1.0
    # Move the robot to the first postion
    move_group.set_pose_target(pose_goal)

    plan = move_group.go(wait=True)
    # Calling `stop()` ensures that there is no residual movement
    move_group.stop()
    s.im_num=s.im_num+1
    s.capture()
    print('sleeping')
    time.sleep(1.5)

    print "====== Moving robot to the third image pose ======"
    move_group.clear_pose_targets()

    q = tf.transformations.quaternion_from_euler(1.3087151694609415, 1.1710170611405144,
        1.32071516512365)
    #q=[ 0, 0.6085625, 0.0405708, 0.7924681 ]
    pose_goal.orientation.x = q[0]
    pose_goal.orientation.y = q[1]
    pose_goal.orientation.z = q[2]
    pose_goal.orientation.w = q[3]
    pose_goal.position.x = 3.70
    pose_goal.position.y = 6.05
    pose_goal.position.z = 0.90

    # Move the robot to the first postion
    move_group.set_pose_target(pose_goal)

    plan = move_group.go(wait=True)
    # Calling `stop()` ensures that there is no residual movement
    move_group.stop()


    # Call the capture service (to obtain the first picture)
    s.im_num=s.im_num+1
    s.capture()
    print('sleeping')
    time.sleep(1.5)
    move_group.clear_pose_targets()

class Sample_01:
```

```
def __init__(self):
    #wait until the service is available
    rospy.loginfo('waiting for the service')
    rospy.wait_for_service("/zivid_camera/capture", 30.0)

    self.tf_listener = tf.TransformListener()
    #create a subscriber

    self.depth_sub = rospy.Subscriber('/zivid_camera/depth/image_raw', Image, self.
        callback3)
    self.image_sub = rospy.Subscriber('/zivid_camera/color/image_color', Image, self.
        callback)

    self.Pointcloud_sub = rospy.Subscriber('/zivid_camera/points', PointCloud2, self.
        callback2)
    #call the capture service
    self.capture_service = rospy.ServiceProxy("/zivid_camera/capture", Capture)
    self.bridge = CvBridge()
    rospy.loginfo("Enabling the reflection filter")
    general_config_client = dynamic_reconfigure.client.Client(
        "/zivid_camera/capture/general/"
    )
    general_config = {"filters_reflection_enabled": True}
    general_config_client.update_configuration(general_config)

    rospy.loginfo("Enabling and configure the first frame")
    frame0_config_client = dynamic_reconfigure.client.Client(
        "/zivid_camera/capture/frame_0"
    )
    frame0_config = {"enabled": True, "iris": 17, "exposure_time": 60000, "brightness":1.0,
         "gain": 4.0}
    frame0_config_client.update_configuration(frame0_config)

    rospy.loginfo("Enabling and configure the second frame")
    frame1_config_client = dynamic_reconfigure.client.Client(
        "/zivid_camera/capture/frame_1"
    )
    frame1_config = {"enabled": True, "iris": 17, "exposure_time": 60000, "brightness":1.0,
         "gain": 4.0}
    frame1_config_client.update_configuration(frame1_config)


def callback(self,data):
    rospy.loginfo('Taking image')
    (trans,rot) = self.tf_listener.lookupTransform('/world','/zivid_optical_frame', rospy.
        Time(0))
    matrix = quaternion_matrix(rot)

    transf_mat=np.zeros([4, 4], dtype = float)
    transf_mat[0:4,0:4]=matrix
    i=0
```

```
        for e in trans:
            transf_mat[i,3]=e
            i=i+1
        directory_matr = r'/home/eduard/LIBRES_SYS/yolov3/data/transf_matrix'
        file_name= 'matrix'+str(self.im_num)+'.npy'
        np.save(os.path.join(directory_matr, file_name), transf_mat)

        try:
            cv_image =self.bridge.imgmsg_to_cv2(data, "bgr8")
        except CvBridgeError as e:
           print(e)
        (rows,cols,channels) = cv_image.shape
        #cv_image= HoG_1.test1(cv_image)

        #self.im_num=self.im_num+1
        directory_im = r'/home/eduard/LIBRES_SYS/yolov3/data/samples'
        im_name= str(self.im_num)+'.jpg'
        cv2.imwrite(os.path.join(directory_im , im_name), cv_image)
        cv2.waitKey(100)
        rospy.loginfo( 'Image saved')
        cv2.destroyAllWindows()

    def callback2( self ,ros_cloud):

        points_list = []
        for data in point_cloud2.read_points(ros_cloud, skip_nans=True):
            points_list.append([data[0], data[1], data[2], data[3]])

        pcl_data = pcl.PointCloud_PointXYZRGB()
        pcl_data.from_list(points_list)
        pc_num= str(self.im_num)+'.pcd'
        rospy.loginfo(pc_num)
        pcl_data.to_file('/home/eduard/LIBRES_SYS/yolov3/data/samples/'+pc_num)

        rospy.loginfo( 'PointCloud2 saved')


    def callback3( self ,data):
        try:
            NewImg = self.bridge.imgmsg_to_cv2(data, "passthrough")
            depth_array = np.array(NewImg, dtype=np.float32)
            directory_im2 = r'/home/eduard/LIBRES_SYS/yolov3/data/depth/'
            im_name= 'depth'+str(self.im_num)+'.npy'
            np.save(os.path.join(directory_im2, im_name), depth_array )
            cv2.waitKey(100)
            print('npy saved')
        except CvBridgeError as e:
            print(e)

        cv2.destroyAllWindows()
        rospy.loginfo( 'depth image saved')
```

```
    def capture(self):
        rospy.loginfo("Calling capture service")
        self.capture_service()

if __name__ == '__main__':
    main()
```

B.10 [move_pos.py]: Move the robot TCP to a specific position

```
#!/usr/bin/env python

import sys
import os
import copy
import rospy
import tf
import numpy as np
import dynamic_reconfigure.client
from zivid_camera.srv import *
from sensor_msgs.msg import PointCloud2
import cv2
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import time
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
from math import pi
from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list
from sensor_msgs import point_cloud2
import pcl
from tf.transformations import quaternion_matrix
import argparse
#from pyntcloud import PyntCloud

def main():
# intialize  movitcommander and the node
    im_num=0
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('robot_irb', anonymous=True)
#Instantiate a RobotCommander object. Provides information

    robot = moveit_commander.RobotCommander()
#Instantiate a PlanningSceneInterface object.  This provides a remote interface  for  getting,

    scene = moveit_commander.PlanningSceneInterface()

#define the groupname
```

```
    group_name = 'righty_tcp2'
    move_group = moveit_commander.MoveGroupCommander(group_name)
    move_group.set_planner_id('RRTconnect')
    move_group.set_planning_time(7)
    move_group.set_goal_position_tolerance(0.001)
    move_group.set_goal_orientation_tolerance(0.005)

    #define the Publisher
    display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',
                                                   moveit_msgs.msg.DisplayTrajectory,
                                                   queue_size=20)

    print " Moving robot"
    #print robot.get_current_state()
    pose_goal = move_group.get_random_pose()
    #pose goal in quaternions
    pose_goal = geometry_msgs.msg.Pose()
    #pose_goal = move_group.get_random_pose()
    pose_goal.orientation.x = 0.0
    pose_goal.orientation.y = 0.7071068
    pose_goal.orientation.z = 0.0
    pose_goal.orientation.w = 0.7071068
    pose_goal.position.x = float(opt.x)
    pose_goal.position.y = float(opt.y)
    pose_goal.position.z = float(opt.z)

    # Move the robot to the first position
    move_group.set_pose_target(pose_goal)
    plan2=move_group.plan()

    plan = move_group.execute(plan2,wait=True)
    # Calling `stop()` ensures that there is no residual movement
    move_group.stop()
    move_group.clear_pose_targets()
    print " Robot moved"

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--x', type=str, default=3.2, help='desired x pos')
    parser.add_argument('--y', type=str, default=3.2, help='desired y pos')
    parser.add_argument('--z', type=str, default=2.0, help='desired z pos')
    opt = parser.parse_args()
    main()
```

B.11 [move_to_cal.py]: Calibration images capturing

```
#!/usr/bin/env python

import sys
```

```
import os
import copy
import rospy
import tf
import numpy as np
import dynamic_reconfigure.client
from zivid_camera.srv import *
from sensor_msgs.msg import PointCloud2
import cv2
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import time
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
from math import pi
from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list
from sensor_msgs import point_cloud2
import pcl
from tf.transformations import quaternion_matrix
#from pyntcloud import PyntCloud


def main():
# intialize  movitcommander and the node
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('robot_irb', anonymous=True)
#Instantiate a RobotCommander object. Provides information
    s = Sample_01()

    robot = moveit_commander.RobotCommander()
#Instantiate a PlanningSceneInterface object.  This provides a remote interface  for  getting,

    scene = moveit_commander.PlanningSceneInterface()

#define the groupname
    group_name = 'righty_tcp'
    move_group = moveit_commander.MoveGroupCommander(group_name)
    move_group.set_planner_id('RRTconnect')
    move_group.set_goal_position_tolerance(0.006)
    move_group.set_goal_orientation_tolerance(0.008)

    #define the Publisher
    display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',
                                                   moveit_msgs.msg.DisplayTrajectory,
                                                   queue_size=20)

    pose_goal = geometry_msgs.msg.Pose()
```

```
    ql=[[ 0, 0.6427792, 0.0080347, 0.7660094 ],[ 0, 0.6419443, 0.080243, 0.7625409 ], [ 0,
        0.6746999, 0.0793765, 0.7338116 ],[ 0, 0.8033253, 0.052554, 0.5932171 ],[ 0, 0.8188746,
         0.0372216, 0.5727643 ],[ 0, 0.8038569, 0, 0.5948228 ],[ 0, 0.7524433, -0.1151699,
        0.6485098 ],[ 0, 0.7768853, -0.0380826, 0.6284894 ],[ 0, 0.7061704, -0.0784634,
        0.7036809 ],[ 0, 0.6746999, -0.0793765, 0.7338116 ],[ 0, 0.6419443, -0.080243,
        0.7625409 ],[ 0, 0.7071068, 0, 0.7071068 ]]
    for q in ql:
        pose_goal.orientation.x = q[0]
        pose_goal.orientation.y = q[1]
        pose_goal.orientation.z = q[2]
        pose_goal.orientation.w = q[3]
        pose_goal.position.x = 3.4
        pose_goal.position.y = 6.0
        pose_goal.position.z = 0.7
        #joint_goal = move_group.get_current_joint_values()
        #print joint_goal
        #joint_goal[5] = 0

        move_group.set_pose_target(pose_goal)
        #plan= move_group.go(joint_goal, wait=True)
        plan = move_group.go(wait=True)
        os.system(' python3 /home/yolov3/detect.py')
        # Calling `stop()` ensures that there is no residual movement
        move_group.stop()
        s.capture()
        time.sleep(35)

        print "------------------------------"
        # It is always good to clear your targets after planning with poses.
        # Note: there is no equivalent function for clear_joint_value_targets()
        move_group.clear_pose_targets()
class Sample_01:
    def __init__(self):
        #Node declaration

        #wait until the service is available
        rospy.loginfo( 'waiting for the service')
        rospy.wait_for_service("/zivid_camera/capture", 30.0)

        self.tf_listener = tf.TransformListener()
        #create a subscriber
        self.image_sub = rospy.Subscriber('/zivid_camera/color/image_color', Image, self.
            callback)
        self.Pointcloud_sub = rospy.Subscriber('/zivid_camera/points', PointCloud2, self.
            callback2)
        #call the capture service
        self.capture_service = rospy.ServiceProxy("/zivid_camera/capture", Capture)
        self.bridge = CvBridge()
        rospy.loginfo("Enabling the reflection  filter ")
        general_config_client = dynamic_reconfigure.client.Client(
            "/zivid_camera/capture/general/"
```

```
        )
        general_config = {"filters_reflection_enabled": True}
        general_config_client.update_configuration(general_config)

        rospy.loginfo("Enabling and configure the first frame")
        frame0_config_client = dynamic_reconfigure.client.Client(
            "/zivid_camera/capture/frame_0"
        )
        frame0_config = {"enabled": True, "iris": 21, "exposure_time": 20000}
        frame0_config_client.update_configuration(frame0_config)

    def callback(self,data):
        print 'ok'
    def callback2(self,ros_cloud):
        print 'ok2'
    def capture(self):
        rospy.loginfo("Calling capture service")
        self.capture_service()




if __name__ == '__main__':
    main()
```

B.12 Moving the robot basic code

```
#intialize movitcommander and the node
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('robot_irb', anonymous=True)
#Instantiate a RobotCommander object. Provides information
    robot = moveit_commander.RobotCommander()
#Instantiate a PlanningSceneInterface object. This provides a remote interface for getting,
    scene = moveit_commander.PlanningSceneInterface()
#define the groupname
    group_name = 'righty_tcp2'
    move_group = moveit_commander.MoveGroupCommander(group_name)
#set the planner (RRT, RRTconnect, PRM...)
    move_group.set_planner_id('RRTconnect')
    move_group.set_planning_time(7)
#set tolerances
    move_group.set_goal_position_tolerance(0.001)
    move_group.set_goal_orientation_tolerance(0.005)
#define the Publisher
    display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',
                                                   moveit_msgs.msg.DisplayTrajectory,
                                                   queue_size=20)
    pose_goal = move_group.get_random_pose()
#pose goal in quaternions
```

```
    pose_goal = geometry_msgs.msg.Pose()
# Move the robot to the position
    move_group.set_pose_target(pose_goal)
    plan2=move_group.plan()
    plan = move_group.execute(plan2,wait=True)
# Calling ‘stop()‘ ensures that there is no residual movement
    move_group.stop()
```

B.13 [training_data.py]: Training images capture

```
#!/usr/bin/env python

import sys
import os
import copy
import rospy
import tf
import numpy as np
import dynamic_reconfigure.client
from zivid_camera.srv import *
from sensor_msgs.msg import PointCloud2
import cv2
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import time
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
from math import pi
from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list
from sensor_msgs import point_cloud2
import pcl
from tf.transformations import quaternion_matrix
#from pyntcloud import PyntCloud

def main():
#intialize movitcommander and the node
    im_num=0
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('robot_irb', anonymous=True)
#Instantiate a RobotCommander object. Provides information
    s = Sample_01()
    s.im_num=0
    robot = moveit_commander.RobotCommander()
#Instantiate a PlanningSceneInterface object. This provides a remote interface for getting,

    scene = moveit_commander.PlanningSceneInterface()
```

```
#define the groupname
    group_name = 'righty_tcp'
    move_group = moveit_commander.MoveGroupCommander(group_name)
    move_group.set_planner_id('RRTconnect')
    move_group.set_goal_tolerance(0.001)

    #define the Publisher
    display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',
                                            moveit_msgs.msg.DisplayTrajectory,
                                            queue_size=20)

    print "====== Moving robot to the first image pose ======"

    pose_goal = move_group.get_random_pose()
    #pose goal in quaternions
    pose_goal = geometry_msgs.msg.Pose()
    y=[0.992713,0.997495,1.0,1.0,0.996865]
    w=[0.1205028,0.0707372,0.0,0.0,-0.0791209]
    #pose_goal = move_group.get_random_pose()
    pose_goal.orientation.x = 0.0
    pose_goal.orientation.y = y[0]
    pose_goal.orientation.z = 0.0
    pose_goal.orientation.w = w[0]
    pose_goal.position.x = 3.5
    pose_goal.position.y = 7.20
    pose_goal.position.z = 1.8
    # Move the robot to the first postion
    for e in range(0,4):
        print '--------------'
        pose_goal.orientation.x = 0.0
        pose_goal.orientation.y = y[e]
        pose_goal.orientation.z = 0.0
        pose_goal.orientation.w = w[e]

        for i in range(0,20):
            pose_goal.position.y = pose_goal.position.y-0.02
            move_group.set_pose_target(pose_goal)
            plan = move_group.go(wait=True)
            # Calling `stop()` ensures that there is no residual movement
            move_group.stop()
            # Call the capture service
            s.capture()
            s.im_num=s.im_num+1

        pose_goal.position.y = 7.20

        pose_goal.position.x = pose_goal.position.x+0.15


class Sample_01:
    def __init__(self):
```

```
        #wait until the service is available
        rospy.loginfo( 'waiting for the service')
        rospy.wait_for_service("/zivid_camera/capture", 30.0)

        self.tf_listener = tf.TransformListener()
        #create a subscriber
        self.image_sub = rospy.Subscriber('/zivid_camera/color/image_color', Image, self.
            callback)
        #call the capture service
        self.capture_service = rospy.ServiceProxy("/zivid_camera/capture", Capture)
        self.bridge = CvBridge()
        rospy.loginfo("Enabling the reflection filter")
        general_config_client = dynamic_reconfigure.client.Client(
            "/zivid_camera/capture/general/"
        )
        general_config = {"filters_reflection_enabled": True}
        general_config_client.update_configuration(general_config)

        rospy.loginfo("Enabling and configure the first frame")
        frame0_config_client = dynamic_reconfigure.client.Client(
            "/zivid_camera/capture/frame_0"
        )
        frame0_config = {"enabled": True, "iris": 17, "exposure_time": 60000, "brightness":1.0,
             "gain": 4.0}
        frame0_config_client.update_configuration(frame0_config)

    def callback(self,data):
        try:
            cv_image =self.bridge.imgmsg_to_cv2(data, "bgr8")
            cv2.waitKey(50)
        except CvBridgeError as e:
            print(e)
        (rows,cols,channels) = cv_image.shape
        #cv_image= HoG_1.test1(cv_image)

        #self.im_num=self.im_num+1
        directory_im = r'/home/LIBRES_SYS/training_images/'
        #cv2.imshow("Image window", cv_image)
        cv2.waitKey(50)
        im_name= str(self.im_num)+'.jpg'
        cv2.imwrite(os.path.join(directory_im , im_name), cv_image)
        cv2.waitKey(50)
        cv2.destroyAllWindows()
        rospy.loginfo( 'Image saved')

    def capture(self):
        rospy.loginfo("Calling capture service")
        self.capture_service()


if __name__ == '__main__':
```

```
    main()
```

B.14 [obtain_calfiles.py]: Obtain calibration files

```
import sys
sys.path.remove('/opt/ros/kinetic/lib/python2.7/dist-packages')
from scipy.spatial.transform import Rotation as R
import math
import yaml
import numpy as np
import cv2

def transform_matrix(quarti, txyz):
    transf=[]
    offset = np.array([1700,200,900])
    txyz = list((np.array(txyz)+ offset))
    quart= [quarti[0], quarti[1], quarti[2], quarti[3]]
    Rot_part_obj = R.from_quat(quart)
    Rot_part = Rot_part_obj.as_matrix()
    for i in range(3):
        Rot_lst=[]
        for e in range(3):
            Rot_num= float(Rot_part[i][e])
            Rot_lst=Rot_lst+[Rot_num]


        transf= transf + Rot_lst
        transf= transf + [txyz[i]]
    transf= transf +  [0.,0.,0.,1.]
    shape = (4,4)
    transf  = np.array(transf)
    transf_m= transf.reshape(shape)
    return transf_m

def create_yaml(transf,name):
    cv_file  = cv2.FileStorage(name, cv2.FILE_STORAGE_WRITE)
    matrix = np.matrix(transf)
    print("write matrix\n", matrix)
    cv_file.write("PoseState", matrix)
    cv_file.release()


quat_lst=[[ 0, 0.6427792, 0.0080347, 0.7660094 ],[  0, 0.6419443, 0.080243, 0.7625409 ], [  0,
    0.6746999, 0.0793765, 0.7338116 ],[-0.06162842,  0.66446302,  0.24184476,  0.70441603],[  0,
     0.8188746, 0.0372216, 0.5727643 ],[  0, 0.8038569, 0, 0.5948228 ],[-0.02268207,
    0.75995941, -0.09463294, 0.6426444 ],[-0.02268207,  0.75995941, -0.09463294, 0.6426444
     ],[  0.15304592,  0.69034553, -0.15304592,  0.69034553],[  0,  0.6746999, -0.0793765,
    0.7338116 ],[  0,  0.6419443, -0.080243, 0.7625409 ],[  0,  0.7071068,  0,  0.7071068 ]]
```

```
txyz_lst
        =[[3.4,6.0,0.7],[3.4,6.0,0.7],[3.4,6.0,0.7],[3.4,6.0,0.7],[3.4,6.0,0.7],[3.4,6.0,0.7],[3.4,6.0,0.7],[3.4,6

leng=len(quat_lst)
for e in range(leng):

    transf= transform_matrix(quat_lst[e],txyz_lst[e])
    ep=e+1
    num= str(ep)
    pos= 'pos'
    yaml_ext='.yaml'
    name=pos+num+yaml_ext
    create_yaml(transf, name)
```

Appendix C

Extrinsic calibration

C.1 Extrinsic calibration (Zivid Command Line Interface tool)

```
C:\Users\Eduard Marti>ZividExperimentalHandEyeCalibration.exe --eih -d "C:\Users\
    Eduard Marti\Documents\final_cal_files" --tf "C:\Users\Eduard Marti\Documents\
    final_cal_files\calibration transform.yaml" --rf "C:\Users\Eduard Marti\
    Documents\final_cal_files\residual.yaml"
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img01.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos01.yaml
Detecting square centers... OK
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img02.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos02.yaml
Detecting square centers... OK
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img03.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos03.yaml
Detecting square centers... OK
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img04.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos04.yaml
Detecting square centers... OK
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img05.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos05.yaml
Detecting square centers... OK
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img06.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos06.yaml
Detecting square centers... OK
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img07.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos07.yaml
Detecting square centers... OK
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img08.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos08.yaml
Detecting square centers... OK
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img09.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos09.yaml
Detecting square centers... OK
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img10.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos10.yaml
Detecting square centers... OK
```

```
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img11.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos11.yaml
Detecting square centers... OK
Reading frame from C:\Users\Eduard Marti\Documents\final_cal_files\img12.zdf
Reading pose from C:\Users\Eduard Marti\Documents\final_cal_files\pos12.yaml
Detecting square centers... OK
Pose file not found C:\Users\Eduard Marti\Documents\final_cal_files\pos13.yaml
Performing hand-eye calibration ...

--- Hand-Eye calibration transform ---
[ [-0.194842, 0.016312, 0.980699, -1040.664648],
  [-0.980823, 0.001594, -0.194893, 285.822855],
  [-0.004742, -0.999866, 0.015689, 43.799124],
  [ 0.000000, 0.000000, 0.000000, 1.000000] ]

--- Per-pose calibration residuals ---
000: { Residual for rotation in deg: 0.263577, Residual for translation in mm:
    5.53127 }
001: { Residual for rotation in deg: 0.643463, Residual for translation in mm:
    19.4177 }
002: { Residual for rotation in deg: 0.483391, Residual for translation in mm:
    14.181 }
003: { Residual for rotation in deg: 0.535631, Residual for translation in mm:
    21.6278 }
004: { Residual for rotation in deg: 0.485077, Residual for translation in mm:
    14.1401 }
005: { Residual for rotation in deg: 0.477544, Residual for translation in mm:
    22.2801 }
006: { Residual for rotation in deg: 0.424545, Residual for translation in mm:
    14.4672 }
007: { Residual for rotation in deg: 0.28992, Residual for translation in mm:
    10.131 }
008: { Residual for rotation in deg: 0.263035, Residual for translation in mm:
    8.53168 }
009: { Residual for rotation in deg: 0.388036, Residual for translation in mm:
    8.65553 }
010: { Residual for rotation in deg: 0.281926, Residual for translation in mm:
    6.35422 }
011: { Residual for rotation in deg: 0.775168, Residual for translation in mm:
    4.51259 }

Saving hand-eye transform to file C:\Users\Eduard Marti\Documents\final_cal_files
    \calibration transform.yaml

Saving hand-eye residuals report to file C:\Users\Eduard Marti\Documents\
    final_cal_files\residual.yaml
```

C.2 Final calibration transform: calibration_transform.yalm

```
%YAML:1.0
---
PoseState: !!opencv-matrix
   rows: 4
   cols: 4
   dt: d
   data: [ -1.9484199038762506e-01, 1.6311914540471650e-02,
       9.8069899572999109e-01, -1.0406646482792733e+03,
       -9.8082317916272654e-01, 1.5940480551990266e-03,
       -1.9489317645294763e-01, 2.8582285515849640e+02,
       -4.7423621657005643e-03, -9.9986568120664199e-01,
       1.5688516381967554e-02, 4.3799123724683504e+01, 0., 0., 0., 1. ]
```


C.3 Calibration positions

C.3.1 pos01.yaml


```
%YAML:1.0
---
PoseState: !!opencv-matrix
   rows: 4
   cols: 4
   dt: d
   data: [ 1.7354073459727493e-01, -1.2309310747712398e-02,
       9.8474976227687105e-01, 1.7034000000000001e+03,
       1.2309310747712398e-02, 9.9987088719921091e-01,
       1.0329075485191144e-02, 206., -9.8474976227687105e-01,
       1.0329075485191144e-02, 1.7366984739806390e-01,
       9.0070000000000005e+02, 0., 0., 0., 1. ]
```


C.3.2 pos02.yaml


```
%YAML:1.0
---
PoseState: !!opencv-matrix
   rows: 4
   cols: 4
   dt: d
   data: [ 1.6293719307784388e-01, -1.2237713306151238e-01,
       9.7901752201661729e-01, 1.7034000000000001e+03,
       1.2237713306151238e-01, 9.8712212251401188e-01,
       1.0302306803370079e-01, 206., -9.7901752201661729e-01,
       1.0302306803370079e-01, 1.7581507056383178e-01,
       9.0070000000000005e+02, 0., 0., 0., 1. ]
```


C.3.3 pos03.yaml

%YAML:1.0
−−−
PoseState: !! opencv−matrix
rows: 4
cols : 4
dt: d
data: [7.6958876780053198e−02, −1.1649478733060624e−01,
9.9020517864205781e−01, 1.7034000000000001e+03,
1.1649478733060624e−01, 9.8739874310170650e−01,
1.0711062807194831e−01, 206., −9.9020517864205781e−01,
1.0711062807194831e−01, 8.9560133678346809e−02,
9.0070000000000005e+02, 0., 0., 0., 1.]

C.3.4 pos04.yaml

%YAML:1.0
−−−
PoseState: !! opencv−matrix
rows: 4
cols : 4
dt: d
data: [−2.9618693627839693e−01, −6.2351863678280883e−02,
9.5309262082646606e−01, 1.7034000000000001e+03,
6.2351863678280883e−02, 9.9447615410319701e−01,
8.4435916622959947e−02, 206., −9.5309262082646606e−01,
8.4435916622959947e−02, −2.9066309038159394e−01,
9.0070000000000005e+02, 0., 0., 0., 1.]

C.3.5 pos05.yaml

%YAML:1.0
−−−
PoseState: !! opencv−matrix
rows: 4
cols : 4
dt: d
data: [−3.4388211420053721e−01, −4.2638407278654221e−02,
9.3804427281323377e−01, 1.7034000000000001e+03,
4.2638407278654221e−02, 9.9722910499072115e−01,
6.0959645538217144e−02, 206., −9.3804427281323377e−01,
6.0959645538217144e−02, −3.4111121919125836e−01,
9.0070000000000005e+02, 0., 0., 0., 1.]

C.3.6 pos06.yaml

%YAML:1.0

```
---
PoseState: !!opencv-matrix
   rows: 4
   cols: 4
   dt: d
   data: [ -2.9237172915775939e-01, 0., 9.5630474849249925e-01,
       1.7034000000000001e+03, 0., 1.0000000000000002e+00, 0., 206.,
       -9.5630474849249925e-01, 0., -2.9237172915775939e-01,
       9.0070000000000005e+02, 0., 0., 0., 1. ]
```


C.3.7 pos07.yaml


```
%YAML:1.0
---
PoseState: !!opencv-matrix
   rows: 4
   cols: 4
   dt: d
   data: [ -1.5887006706504336e-01, 1.4937761967995802e-01,
       9.7593372138147683e-01, 1.7034000000000001e+03,
       -1.4937761967995802e-01, 9.7347178790393163e-01,
       -1.7331764161178836e-01, 206., -9.7593372138147683e-01,
       -1.7331764161178836e-01, -1.3234185496897521e-01,
       9.0070000000000005e+02, 0., 0., 0., 1. ]
```


C.3.8 pos08.yaml


```
%YAML:1.0
---
PoseState: !!opencv-matrix
   rows: 4
   cols: 4
   dt: d
   data: [ -2.1000213213137930e-01, 4.7869021821041920e-02,
       9.7652837196374975e-01, 1.7034000000000001e+03,
       -4.7869021821041920e-02, 9.9709943109557309e-01,
       -5.9171625453264126e-02, 206., -9.7652837196374975e-01,
       -5.9171625453264126e-02, -2.0710156322695228e-01,
       9.0070000000000005e+02, 0., 0., 0., 1. ]
```


C.3.9 pos09.yaml


```
%YAML:1.0
---
PoseState: !!opencv-matrix
   rows: 4
   cols: 4
```

dt: d
data: [−9.6663304535539174e−03, 1.1042639760023416e−01,
9.9383729692973277e−01, 1.7034000000000001e+03,
−1.1042639760023416e−01, 9.8768698908061003e−01,
−1.1081706688914875e−01, 206., −9.9383729692973277e−01,
−1.1081706688914875e−01, 2.6466804658361598e−03,
9.0070000000000005e+02, 0., 0., 0., 1.]

C.3.10 pos10.yaml

%YAML:1.0
−−−
PoseState: !! opencv−matrix
rows: 4
cols: 4
dt: d
data: [7.6958876780053198e−02, 1.1649478733060624e−01,
9.9020517864205781e−01, 1.7034000000000001e+03,
−1.1649478733060624e−01, 9.8739874310170650e−01,
−1.0711062807194831e−01, 206., −9.9020517864205781e−01,
−1.0711062807194831e−01, 8.9560133678346809e−02,
9.0070000000000005e+02, 0., 0., 0., 1.]

C.3.11 pos11.yaml

%YAML:1.0
−−−
PoseState: !! opencv−matrix
rows: 4
cols: 4
dt: d
data: [1.6293719307784388e−01, 1.2237713306151238e−01,
9.7901752201661729e−01, 1.7034000000000001e+03,
−1.2237713306151238e−01, 9.8712212251401188e−01,
−1.0302306803370079e−01, 206., −9.7901752201661729e−01,
−1.0302306803370079e−01, 1.7581507056383178e−01,
9.0070000000000005e+02, 0., 0., 0., 1.]

C.3.12 pos12.yaml

%YAML:1.0
−−−
PoseState: !! opencv−matrix
rows: 4
cols: 4
dt: d
data: [0., 0., 1.0000000000000002e+00, 1.7034000000000001e+03, 0.,

1.0000000000000002e+00, 0., 206., −1.0000000000000002e+00, 0., 0.,
9.0070000000000005e+02, 0., 0., 0., 1.]

Appendix D

Task planner feedback

```
================IMAGE ACQUISITION================
================IMAGE DETECTION================
Namespace(agnostic_nms=False, cfg='/home/eduard/LIBRES_SYS/yolov3/cfg/yolov3.cfg', classes=None, conf_thres
     =0.3, device='', fourcc='mp4v', half=False, img_size=416, iou_thres=0.6, names='/home/eduard/LIBRES_SYS/
     yolov3/data/LIBRES_classes_1.names', output='/home/eduard/LIBRES_SYS/yolov3/output', save_txt=True,
     source='/home/eduard/LIBRES_SYS/yolov3/data/samples', view_img=False, weights='/home/eduard/LIBRES_SYS/
     yolov3/weights/last.pt')
Using CUDA device0 _CudaDeviceProperties(name='GeForce RTX 2070', total_memory=7952MB)

Model Summary: 222 layers, 6.19491e+07 parameters, 6.19491e+07 gradients
image 1/3 /home/eduard/LIBRES_SYS/yolov3/data/samples/0.jpg: 288x416 4 VW_modules_01s, 1 VW_bms_01s, 29 screws
     , 13 empty_screws, 2 connection_001s, Done. (0.026s)
image 2/3 /home/eduard/LIBRES_SYS/yolov3/data/samples/1.jpg: 288x416 4 VW_modules_01s, 1 VW_bms_01s, 22 screws
     , 13 empty_screws, 2 connection_001s, Done. (0.019s)
image 3/3 /home/eduard/LIBRES_SYS/yolov3/data/samples/2.jpg: 288x416 4 VW_modules_01s, 1 VW_bms_01s, 21 screws
     , 10 empty_screws, 2 connection_001s, Done. (0.019s)
Results saved to /home/eduard/LIBRES_SYS//home/eduard/LIBRES_SYS/yolov3/output
Done. (0.332s)
================PLANNING REMOVAL================
=======Component analysis=======
there are 29 screws
there are 2 connective components
there are 5 components (modules, bms, etc..)
there are 13empty screws
================Analysing the first image positions ================
=======pointcloud analysis=======
-Pointcloud analysed-
=======Component analysis=======
there are 22 screws
there are 2 connective components
there are 5 components (modules, bms, etc..)
there are 13empty screws
================Analysing the second image positions================
=======What components should be removed=======
screws should be removed
task_planner_v6.py:384: RuntimeWarning: invalid value encountered in double_scalars
  mean_exc=(comp_t-intersection_t)/(comp_area-intersection_area)
=======pointcloud analysis=======
-Pointcloud analysed-
=======Component analysis=======
there are 21 screws
there are 2 connective components
there are 5 components (modules, bms, etc..)
there are 10empty screws
================Analysing the third image positions================
=======pointcloud analysis=======
-Pointcloud analysed-
================Merging results================
--Finding the WR frame coordinates for each image--
-First iamge-
-Second iamge-
-Second iamge-
```

```
--merging positions--
================REMOVING COMPONENTS================
=======Remove operation=======
[3.726657567683155, 6.488918064461358, 0.2827778053023142, '3']
Failed to import pyassimp, see https://github.com/ros-planning/moveit/issues/86 for more info
[ INFO] [1592344877.276959264]: Loading robot model 'righty'...
[ INFO] [1592344877.425951391]: Loading robot model 'righty'...
[ INFO] [1592344877.487651533]: Loading robot model 'righty'...
[ INFO] [1592344878.620464443]: Ready to take commands for planning group righty_tcp2.
 -----Moving robot to the safety position-----
 SAFETY POSITION REACHED
 -----Moving robot to the desired position-----
 Robot moved
removing screw
Press Enter to move the robot to the next position...
```

Appendix E

Guidance Code

E.1 moverobot.py

```
!/usr/bin/env python

import sys
import copy
import rospy
import numpy as np
import dynamic_reconfigure.client
import time
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
from math import pi
from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list

def main():
#intialize  movitcommander and the node
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('robot_irb', anonymous=True)
#Instantiate a RobotCommander object. Provides information

    robot = moveit_commander.RobotCommander()
#Instantiate a PlanningSceneInterface object.

    scene = moveit_commander.PlanningSceneInterface()

#define the groupname
    group_name = 'lefty_tcp'
    move_group = moveit_commander.MoveGroupCommander(group_name)
    move_group.set_planner_id('RRTConnectkConfigDefault')

    #define the Publisher
    display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',
                                                   moveit_msgs.msg.DisplayTrajectory,
```

```
                                                queue_size=20)

    print "============ Printing robot state"
    print robot.get_current_state()
    pose_goal = move_group.get_random_pose()

    #pose goal in quaternions
    pose_goal = geometry_msgs.msg.Pose()
    pose_goal = move_group.get_random_pose()
    #q= [0.2677762,0.1967738,0.9304776,0.1542317]
    #txyz= [-559.017,-181.636,809.017]
    #txyz = np.array(txyz)*0.001
    #pose_goal.orientation.x = q[1]
    #pose_goal.orientation.y = q[2]
    #pose_goal.orientation.z = q[3]
    #pose_goal.orientation.w = q[0]
    #pose_goal.position.x = txyz[0]+1.7
    #pose_goal.position.y = txyz[1]+0.2
    #pose_goal.position.z = txyz[2]+0.9


    move_group.set_pose_target(pose_goal)
    print pose_goal
    plan = move_group.go(wait=True)
    print '2'
    # Calling `stop()` ensures that there is no residual movement
    move_group.stop()

    # It is always good to clear your targets after planning with poses.
    # Note: there is no equivalent function for clear_joint_value_targets()
    move_group.clear_pose_targets()


if __name__ == '__main__':
    main()
```