Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

POSICIONAMIENTO DEL EFECTOR FINAL DEL ROBOT BASADO EN PLANOS EXTRAÍDOS DE UNA NUBE DE PUNTOS

Autor:

ANÍBARRO BLANCO, ANA

Responsable de Intercambio en la UVa:

LÓPEZ DE LA FUENTE, EUSEBIO

Universidad de destino:

IVERSITY COLLAGE LEUVEN-LIMBURG (UCLL)

Valladolid, Julio 2021.

TFG REALIZADO EN PROGRAMA DE INTERCAMBIO

TÍTULO: Robot end effector positioning based on planes extracted from a point cloud

ALUMNO: Ana Aníbarro Blanco

FECHA: 17 de junio de 2021

CENTRO: Grupo de investigación ACRO

UNIVERSIDAD: University Collage Leuven- Limburg

TUTOR: Wim Claes

ABSTRACT (ESPAÑOL)

El grupo de investigación ACRO, de la universidad KU Leuven, está actualmente participando en el proyecto ARCHER, cuyo objetivo principal es conseguir la navegación autónoma de un robot en un área determinada, haciendo un mapa del entorno para identificar las posibles fuentes de radiación presentes.

Esta tesis se centra en dos objetivos principales, el análisis del brazo robot Kinova, implementado en el robot ARCHER, usando para ello el framework ROS y un caso práctico de estudio. El brazo debe ser modificado para incorporar los elementos necesarios para determinar el nivel de contaminación, entre los que se encuentran la cámara Intel Realsense L515 LiDAR. En esta tesis se ha propuesto un caso práctico basado en el procesamiento de imágenes 3D obtenidas con dicha cámara. El objetivo es la determinación de una serie de puntos en una imagen 3D de forma que el brazo robot pueda llevar a cabo el escaneo de una superficie.

KEYWORDS: Brazo robot, nube de puntos, LiDAR, PCL, ROS

ABSTRACT (INGLÉS)

The research group ACRO (Automation, Computer Vision and Robotics) from KU Leuven is located in the technology center at campus Diepenbeek, Belgium. Currently, this group is participating in the ARCHER project, which main goal is to make a mobile robot navigate autonomously in a given area, mapping the entire environment, to determine the location of possible sources of radiation present. For this purpose, a mobile platform to which a robotic arm has been incorporated will be used. The robotic arm will be equipped with a camera and a probe to scan the surfaces and localise the nuclear hotspots.

This thesis focuses on two main objectives, the analysis of the Kinova robot arm using the middleware framework ROS (Robot Operating System) and a practical case study. The arm needs to be modified to incorporate the necessary elements to determine the contamination level, among which is the Intel Realsense L515 LiDAR camera. In this thesis a practical case study has been proposed based on the processing of 3D images obtained with the aforementioned camera. The objective is determining a series of points in the 3D image for the robot arm to carry out the scanning task.

Using the PCL library, a plane on which a number of points have been plotted has been defined. Subsequently, a simulation of the robot arm manually moving to the defined points using MoveIt! has been done.

KEYWORDS: Robot arm, ROS, MoveIt!, point cloud, camera, LiDAR, PCL

BACHELOR THESIS

ROBOT END EFFECTOR POSITIONING BASED ON PLANES EXTRACTED FROM A POINT CLOUD

DEGREE IN INDUSTRIAL ELECTRONIC AND AUTOMATIC ENGINEERING

Student:
Ana Aníbarro Blanco
ana.anibarro@alumnos.uva.es

Lector:
Wim Claes

Company promoter:
Eric Demeester

Supervisors:
Hendrik Clijsters
David De Schepper

Hasselt 17th June 2021, Belgium
Academic year 2020-2021

INDEX

PREFACE

This present document constitutes the report of my final bachelor thesis corresponding to the degree of Industrial Electronics and Automation Engineering.

This thesis has entailed a great challenge for me. It has been developed during an Erasmus programme in Belgium, therefor I have worked according to a different country's methodologies. Furthermore, I do not have any experience in the field of industrial robotics, which led to a number of interesting challenges and hardships. It is also worth mentioning that it took place during the covid 19 pandemic, which prevented me from going to the research centre as often as I would have liked.

Nevertheless, it has been a very rewarding and satisfying experience. I have learned about a field of engineering that is located within the areas of great future in the industry, which will help me in my professional career.

Firstly, I would like to thank my promoter, Eric Demeester, for always providing me with the necessary assistance, and my lector, Wim Claes, for making this Erasmus experience possible and helping me throughout the whole process. I also wish to thank my supervisors, Hendrik Clijsters and David De Schepper, for their excellent guidance and support during this process. Finally, I would like to thank personnel of the research group ACRO whose help has been very useful to me in different parts of the thesis.

Finally, I want to thank my family and friends for supporting me even when my strength was failing. I would particularly like to mention my parents and siblings, who were always there to give me support and self-confidence. This journey would not have been possible without them by my side.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACRO	Automation, Computer Vision and Robotics
ARCHER	Autonomous Robotic platform for CHaractERrisation
ROS	Robot Operating System
URDF	Unified Robot Description Format
TF	TransForm
LiDAR	Light Detection And Ranging
MEMS	Micro-Electro Mechanical System
TLS	Terrestrial Laser Scanner
ALS	Airborne Laser Scanner
MLS	Mobile Laser Scanner
GPS	Global Positioning System
IMU	Inertial Measurement Units
PCL	Point Cloud Library
BSD	Berkeley Source Distribution
PDAL	Point Data Abstraction Library
SDK	Software Development Kit

ABSTRACT

The research group ACRO (Automation, Computer Vision and Robotics) from KU Leuven is located in the technology center at campus Diepenbeek, Belgium. Currently, this group is participating in the ARCHER project, which main goal is to make a mobile robot navigate autonomously in a given area, mapping the entire environment, to determine the location of possible sources of radiation present. For this purpose, a mobile platform to which a robotic arm has been incorporated will be used. The robotic arm will be equipped with a camera and a probe to scan the surfaces and localise the nuclear hotspots.

This thesis focuses on two main objectives, the analysis of the Kinova robot arm using the middleware framework ROS (Robot Operating System) and a practical case study. The arm needs to be modified to incorporate the necessary elements to determine the contamination level, among which is the Intel Realsense L515 LiDAR camera. In this thesis a practical case study has been proposed based on the processing of 3D images obtained with the aforementioned camera. The objective is determining a series of points in the 3D image for the robot arm to carry out the scanning task.

Using the PCL library, a plane on which a number of points have been plotted has been defined. Subsequently, a simulation of the robot arm manually moving to the defined points using MoveIt! has been done.

KEYWORDS: Robot arm, ROS, MoveIt!, point cloud, camera, LiDAR, PCL.

1. INTRODUCTION

1.1. General Description

This thesis is part of various studies on autonomous mobile robots at the research center ACRO from KU Leuven located in Diepenbeek, Belgium. The acronym ACRO signifies the expertise of this research group, namely automation, computer vision and robotics. It is formed by the academic and teaching staff as well as PhD researchers. Their projects are based on different fields, e.g. vision-based and model-based automation, product manipulation and robotic grippers, (semi-)autonomous assembly and disassembly, functional programming and cloud computing, collision-free trajectory generation and navigation and human-robot collaboration [1].

This thesis is developed as part of the ARCHER project (Autonomous Robotic platform for CHaractERrisation). The project aims to autonomously navigate and map unknown environments, indicating potentially contaminated regions on the created maps using an autonomous mobile vehicle.

The ARCHER project is an ongoing study between the ACRO research group and the nuclear technology research group at UHasselt, NuTeC. The academic research is supported by the two major contributors to the ARCHER project, which are the companies Tecnubel and Magics. Tecnubel is the company in charge of the design and construction of the mobile manipulator and receives help from Magics, in charge of sensors and electronics [2].

1.2. Problem statement

The robot used in the ARCHER project is an autonomous mobile manipulator. It consists of a mobile platform with continuous tracks on which a robot arm has been mounted. The following images (Figure 1.1 and 1.2) show photos captured from the ARCHER robot. It should be noted that to autonomously navigate, the platform uses wheel encoders and a LiDAR camera. The camera has been mounted on the front of the sensor, seen in Figure 1.2.

Figure 1.1. Side view of the ARCHER robot.

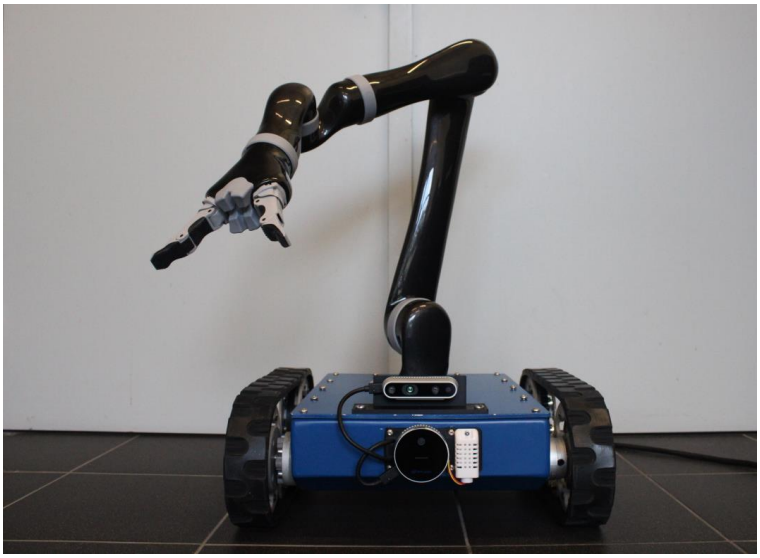

Figure 1.2. Front view of the ARCHER robot.

The project is currently at the following stage: the mobile platform can map the environment, locate the vehicle, and perform the trajectory from point A to point B in real time autonomously.

At this stage, the work carried out has been focused on the mobile platform however the arm has not been the subject of much research nor development. The aim is to study the functioning of the robot arm incorporated in the platform. The arm is intended to scan a surface, on which it must measure the amount of nuclear contamination present. Thus, it will be necessary to modify the arm's structure to incorporate a LiDAR camera together with a probe. It is noteworthy that no work with this probe will be performed.

1.3. Objectives

The objectives of this thesis can be separated into two main sections. The first is the analysis of the robot arm implemented in the mobile platform. It will be necessary to study and understand in depth how the arm and its interface with ROS (Robot Operating System) work.

The second is the practical case, focused on the processing of a point cloud taken with a LiDAR camera that will be included in the robot arm. This will require an initial literature study focusing on point cloud processing. The original objectives of the case study were:

- Modification of the robot arm to include the Realsense Intel L515 camera.
- Capturing a 3D point cloud of a wall.
- Insertion of a plane through the point cloud.
- Definition of a series of points separated a specified distance from the previously defined plane.
- Performing a coordinate change to refer the defined points of the plane to the robot's frame.
- Moving the robot arm towards these points, with orientation perpendicular to the plane, without colliding with the wall.

However, due to limited time the scope of the thesis has been reduced. Only a case study with the camera will be covered, but it will not be connected to the robot nor used in conjunction with the probe.

1.4. Report structure

The thesis will be structured in five chapters as follows. The initial chapter (current chapter) introduces the overall project together with the specific objectives of the thesis. The second chapter focuses on the description of the technological environment in which the thesis has been developed. It includes an explanation of the programming environment ROS and the MoveIt! extension, an analysis of the Kinova robot arm and the technology behind the Realsense Intel L515 LiDAR camera. Chapter three contains the literature study on point cloud processing. It includes an analysis of the different existing libraries for this purpose. Chapter four develops the proposed case study. The last chapter includes conclusions drawn from the development of the thesis and possible future work.

2. TECHNOLOGICAL ENVIRONMENT

In this second chapter, an introduction to the technological environment in which the work has been developed during the months that the thesis has been carried out will be presented.

Initially, the programming environment of the robot arm, i.e. ROS, will be described, together with one of its most useful packages that controls the arm's movement, MoveIt!. This will be followed by a description of the arm Kinova arm, and a brief explanation of how its ROS interface work. The last part will be a brief description of the camera used for the practical case study.

2.1. Programming environment

2.1.1. ROS

The Robot Operating System (ROS) is a flexible middleware framework based on a collection of tools, libraries and conventions which aim is to simplify the task of developing software for robots [3].

It is an open-source project which provides the typical services of an operating system such as hardware abstraction, low-level device control, implementation of commonly used functionality, message passing between process and management of packages. At the same time, it incorporates a series of tools and libraries to obtain, compile, write and run code across multiple computers [4].

Building robust software for general use, is a difficult and complex task. Problems vary greatly depending on the task to be performed and the environment where the robot is located. ROS was built from scratch to foster collaborative robot software development. The main objective is that everyone can build using the work of others, avoiding the cost of constantly re-inventing the same software by different groups of people [5].

The ROS framework is easy to implement in any modern programming language, such as C++, Python or Java. This helps it to be used by a greater number of robots and, as it is free software, it is constantly evolving and developing [6].

2.1.1.1. Architecture and concepts

To fully understand ROS, it is necessary to distinguish three sections or levels of concepts [7] [8]:

- ROS Filesystem Level: This level indicates the folder structure, how it is formed, and the minimum number of files that ROS needs to work. Software in ROS is organized in packages. The goal of these packages is to provide a useful functionality in an easy-to-consume manner so that software can be easily reused. These packages may contain processes (nodes), libraries, scripts, configuration files (Makefiles), etc. Figure 2.1 shows a diagram of how the ROS file system level is organised.

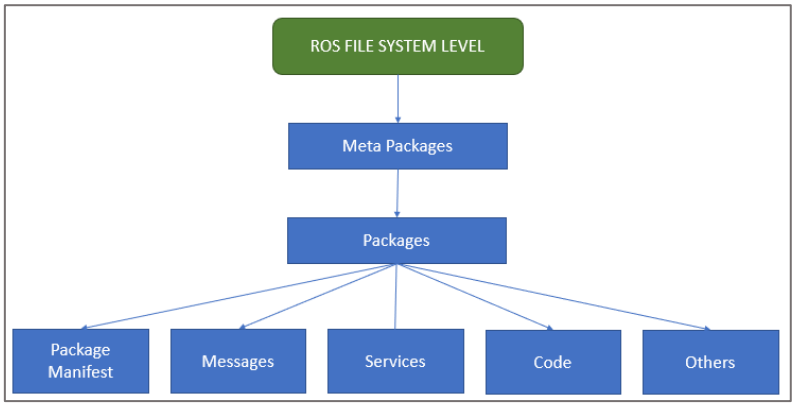

Figure 2.1.. ROS Filesystem Level Description [9].

- ROS Computation graph: The graph structure shows the communication between the different processes of the system. ROS is based on a graph architecture, it has a number of independent nodes that can communicate with the rest of the nodes through the publisher / subscriber model. Figure 2.2 shows the architecture of the computation graph in ROS.

  A node is a process that performs a specific task or function. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server. Due to the modular philosophy of ROS, a system will typically have many nodes to control different specific functions of the robot.

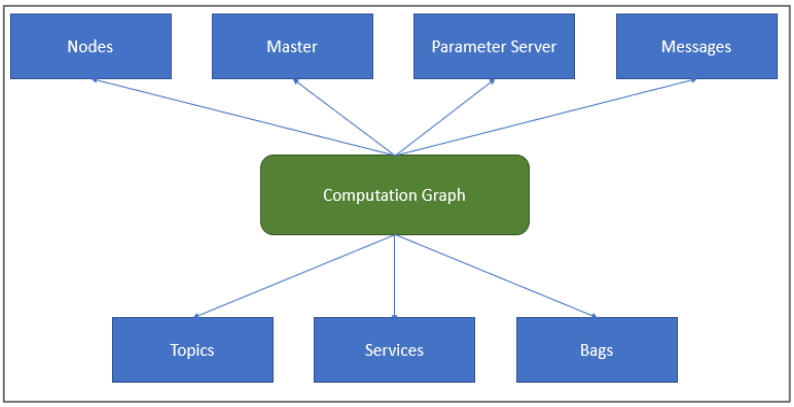

Figure 2.2. ROS Computation Graph concepts [9].

- ROS Community level: Different tools and concepts are used by the ROS community to share knowledge, algorithms and code with any developer. Thanks to this level, ROS is constantly growing and developing.

2.1.1.2. Tools

ROS has multiple tools, from simulators to tools that facilitate the understanding, development and management of the different ROS processes. The most useful tools used in ROS during the development of the project will be shown below.

rviz

Rviz is a 3D visualisation tool for ROS applications. It provides a view of the robot model and captures sensor information from robot sensors. Amongst others, it can visualise pictures and point clouds, obtained from cameras and LiDAR [10]. In addition, this tool has a modular and customisable interface, with the possibility of moving and programming panels, as well as creating new plugins that allow for new functionalities to be added.

URDF

URDF (*Unified Robot Description Format*) is a robot modelling tool. It is responsible for specifying the properties of the robot, such as its dimensions, number of joints, physical parameters, etc. [11].

This information is described in the form of a tree in an XML file, distinguishing two main components necessary for the construction of a robot's kinematic chain, i.e., links and joints.

- Link: The links describe the rigid physical part of the robot, such as mass, geometry or inertia, as well as the visual components needed to display the robot in tools such as rviz. Figure 2.3 shows an example of the elements which can be described in a link.

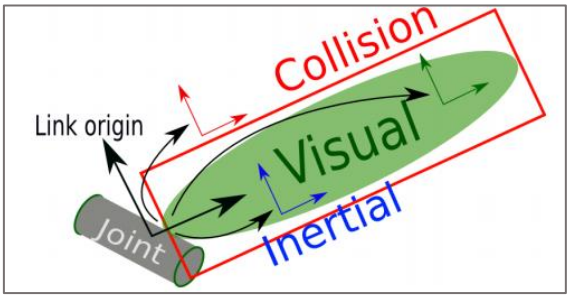

Figure 2.3. Description of a link element within a URDF file [12].

- Joint: The joints indicate the relationship between the different links of the robot. The kinematics and dynamics of each joint are also described, as well as specifying the collision limits of the robot. An example of the attributes which can be described in a joint is shown in Figure 2.4.

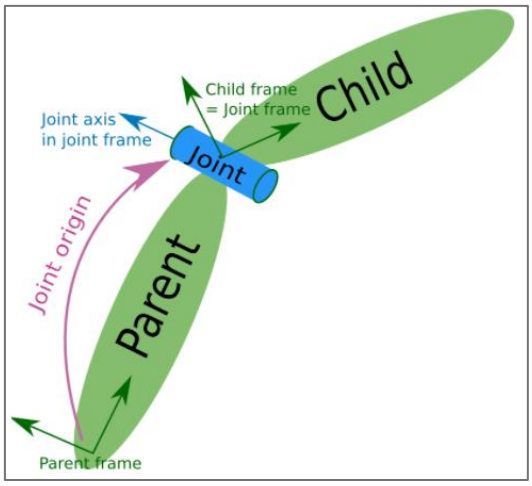

Figure 2.4. Description of a joint element within a URDF file [13].

TF

TF is a package that lets the user keep track of multiple coordinate frames over time. TF maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time [14].

It coordinates and transforms the different reference frames or coordinate axes with respect to a global reference point, and to each other, over time. Figure 2.5 shows an example of a TF structure.

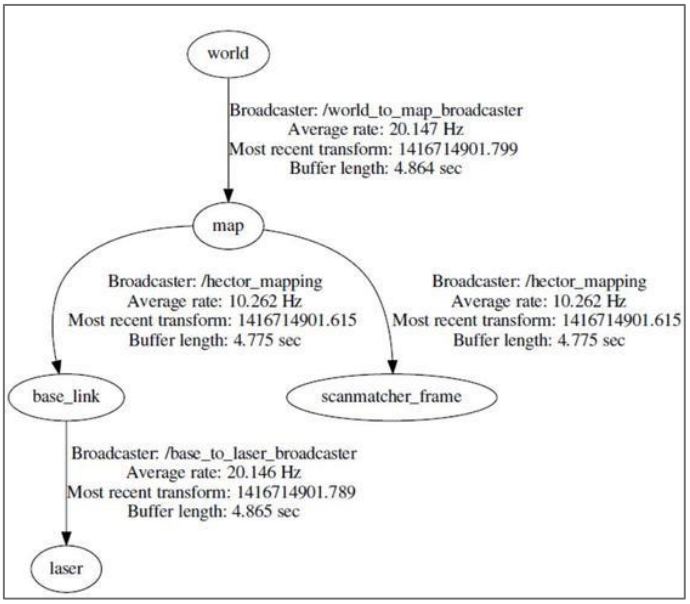

Figure 2.5. TF structure example [14].

2.1.2. MoveIt!

MoveIt! is a software framework in ROS that facilitates trajectory control for robots, with special focus on robots with arms. It incorporates the following functions: motion

planning, manipulation, 3D perception, inverse kinematics, control and collision checking. In addition, it provides the user with an easy-to-use platform for developing new robotic applications [15].

2.1.2.1. High-level architecture of MoveIt!

As can be seen in Figure 2.6, the main core of this software is the node called move_group [16]. To the left of it, the different ways for the user to interact with this node can be seen.

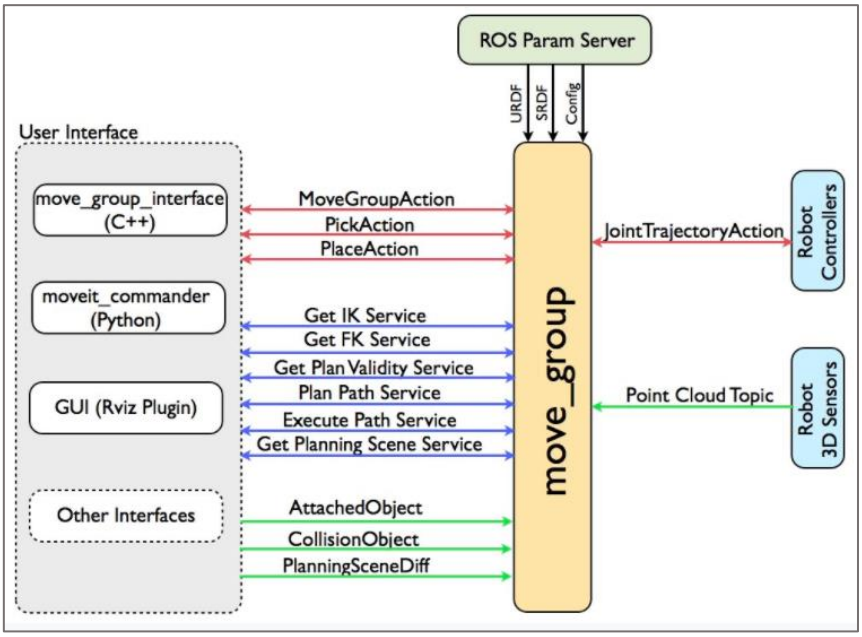

Figure 2.6. High-level system architecture for the move_group node [16].

The most notable ones are:

- Move_group_interface (C++): It consists of a program that sends the information to the move_group node. This code is written in the C++ programming language. It allows, not only to send instructions to the robot, but also to add objects to the environment, set conditions, flows, warnings, etc.
- Moveit_commander (Python): It is a program that comes with the MoveIt! package and controls the robot in a very simple way through a series of commands.
- Graphical User Interface (GUI, Rviz): This is an intuitive and easy way to control the robot. Rviz loads the robot model along with markers, which changes the position of the arm in the direction the marker is dragged. This allows you to see the movements that the robot will make, and even perform the planning before executing the movement, thus avoiding failures and collisions.

As can be seen in the schematic represented in Figure 2.6, the user interaction is connected by several elements to the move_group node. These elements are the different types of communication that exist in ROS. They send the information generated by the user to the different nodes. Each colour represents a different type of communication.

The green arrows represent communication via topic [17]. The information that is transmitted to the kernel in this way concerns the various objects that may appear in the environment. These objects, which can be attached to the robot, must be taken into account when planning trajectories due to the possibility of collision.

The blue arrows denote communication via service [18]. Services in Moveit! are used to send waypoints to the robot, receive the robot's kinematics, both inverse and direct, acquire changes in the environment, and receive, confirm validity of and execute trajectories.

Finally, the red arrows represent the last and most general form of communication found in ROS, the actions. Actions establish a goal that initiates a behaviour or a process and send a signal when this goal is reached [19] [20]. Among the existing actions the movement of the robot, the pick operation and the place operation can be found.

In the upper part of Figure 2.6, the move_group is connected by three black arrows to another element called ROS Param Server. The Parameter Server is a kind of multivariable, shared "dictionary" or database that is accessible to the operating nodes of the system, and it is used to store and retrieve parameters during execution. The information stored in this Parameter Server is the one corresponding to the URDF, SRDF and Config files from the robot [21].

Lastly, on the right side, the move_group is connected through an action with the robot controllers. These receive the movement instructions from the move_group node and return the result when the desired position has been reached or respond with an error when it cannot be reached [16]. On the other hand, if there are sensors that receive information about the robot's environment, such as a camera, it can send the information to the move_group via topic.

2.1.2.2. Motion Planning in MoveIt!

Motion planning deals with the problem of moving the arm to a certain configuration, allowing the end-effector to reach a position without the robot colliding with any obstacle. This can be either an external object or the robot's own parts that may get in the way of the movement [7]. These schedulers, responsible for organising motion planning, are incorporated into MoveIt! in the form of plugins, this makes it easier to communicate with and use various types of schedulers. The move_group node connects to these schedulers through an action or a ROS service [20].

The planning works as follows: first, a motion planning request, that clearly specifies the action to be performed by the robot, is sent, then the planner finds a trajectory for all joints in which collisions are taken into account and which reaches the specified target position. MoveIt! also allows taking into account possible objects that can be picked up when calculating robot trajectories.

Moveit! has a large number of tutorials on its website [22], on which examples of code and use of its different functionalities can be obtained.

2.2. Kinova robot arm analysis

2.2.1. JACO arm

The robotic arm used during the development of this thesis is the model j2s6s200 of the JACO series from Kinova. It is a light-weight robot composed of six inter-linked segments. The user can move the robot in three-dimensional space either through the controller or using the computer. It is equipped with a two-finger gripper with which it can grasp or release objects.

In the current ARCHER project, the aim is to use the arm to perform a radiological measurement on a contaminated surface. To perform this task, the structure of the robot must be modified so that both the camera and probe can be incorporated. This will be explained in more detail in the case study.

2.2.1.1. Robot configuration

The JACO arm is a 6 degree of freedom robotic arm with a spherical wrist. Figure 2.7 shows the model j2s6s200 robot arm used for this thesis.

Figure 2.7. Kinova JACO model j2s6s200 robot arm [23].

Figure 2.8 displays the external connectors located on the base of the robot controller. The on/off switch activates the arm, the power connector provides electrical power, the USB and Ethernet port which allow for communication and pins to connect wired controllers for the arm [23].

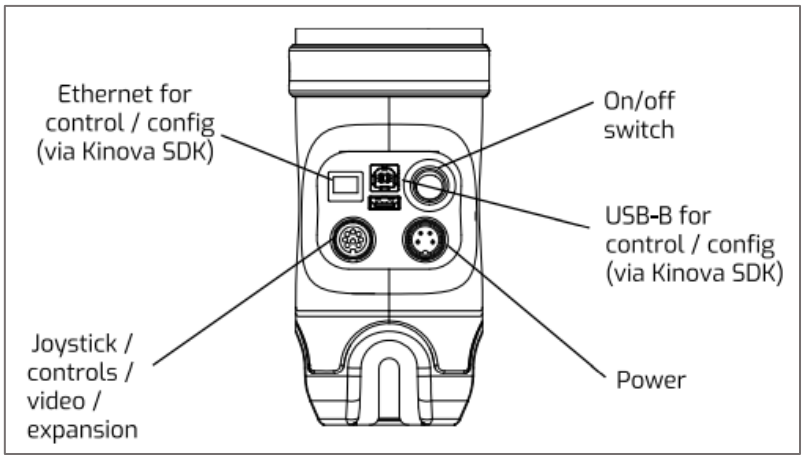

Figure 2.8. External connectors of the JACO arm [23].

2.2.1.2. Robot configurations specifications

Table 2.1 shows the JACO arm configuration specifications. It is essential to take them into account when working with the actual robotic arm.

Total Weight	4.4 kg
Reach	98.4 cm
Maximum payload	2.6 kg (mid-range continuous) 2.2 kg (full reach peak / temporary)
Materials	Carbon fiber (links), Aluminium (actuators)
Joint range (software limitation)	±27.7 turns
Maximum linear and arm speed	20 cm/s
Power supply voltage	18 to 29 VDC
Average power	25 W (5 W in standby)
Peak power	100 W
Communication protocol	RS485
Communication cables	20 pins flex cable
Water resistance	IPX2
Operating temperature	-10 °C to 40 °C

Table 2.1. JACO arm configuration specifications [23].

2.2.1.3. Controlling the robot

The actuators in the Kinova arm can be controlled based on end effector position, actuators' angular position or actuators' torque. Below the different control mode options offered by the robot are specified [23]:

- Cartesian position: Specifies end effector's position and orientation in the base frame.
- Cartesian velocity: Specifies end-effector's translational velocities in the base frame and end-effector's rotational velocities in the end effector's frame.
- Angular position: Specifies each actuator's angle.
- Angular velocity: Specifies each actuator's angular (rotational) velocity.
- Cartesian admittance (Reactive Force control in Cartesian space): Forces and torques are applied on the end-effector to perform a certain translation and rotation (Cartesian motion).
- Angular admittance (Reactive Force control in joint space): Torques are applied on actuators to perform a certain joint rotation (angular motion).
- Direct torque control: Each actuator's torque is specified.
- Force control: Specifies forces and torques at the end-effector. The torque at each actuator to generate the appropriate forces/torques at the end-effector is automatically computed.

Kinova provides three different options to operate with the robotic arm:

- Joystick control: Sends cartesian or angular velocity motion commands. The cartesian mode is set by default.
- Kinova software control: Two different software control panels (the Development Center and the Torque Console) allow users to send position, velocity, and trajectory commands to the robot. In addition to that, the Development Center allows for the activation of admittance control and the Torque Console allows direct torque/force control. They both control the arm via a graphical user interface.
- API control: Kinova provides a C++ library, referred to as Kinova API, to control its robots. It is downloadable as part of the Kinova software development kit (SDK) and supported on both Windows and Ubuntu. It also offers the possibility to control the robot through a ROS interface.

2.2.2. Kinova-ROS

The kinova-ros stack provides a ROS interface for the Kinova Robotic manipulator arm JACO. It is developed above the Kinova C++ API functions, which communicate with the DSP (digital signal processor) inside the robot base.

The first step to use the stack is making kinova-ros part of a workspace. To start working with the arm it is necessary to establish the connection via Ethernet, setting the exact parameters in the robot parameters file.

Once the stack is downloaded it is time to work with the driver. To communicate with the robotic arm, the robot type needs to be specified. This thesis worked with the type j2s6s200, which refers to [24]:

- Robot category: JACO arm (j).
- Version use: 2.
- Wrist type: Spherical (s).
- Degrees of Freedom: 6.
- Robot mode: Service (s).
- Fingers in the gripper: 2.
- The last two positions (00) are not defined and reserved for further features.

To begin with, once connected to ethernet, the driver needs to be launched. The file kinova_robot.launch, located in the kinova_bringup folder, is in charge of launching the essential drivers and configurations. The robot type that is being used needs to be specified in the kinova_robotType argument. As stated before, this project works with the JACO arm type j2s6s200. To launch the driver the following command needs to be typed [24]:

roslaunch kinova_bringup kinova_robot.launch kinova_robotType:=j2s6s200

The arm can be commanded in three different ways: joint position control, cartesian position control and ROS Service Commands.

2.2.2.1. Joint Position Control

The Joint Position Control sends a desired angle to each joint. This position is controlled using PID control to specify the effort to the joint. The safety limits of the robots will constrain the commands from the position controller, therefore, position commands near the joint limits may cannot be achieved [25].

The kinova-ros stack can perform the joint position control in two different ways. The first one is by calling the node joints_action_client.py, located in the kinova_demo package. To run the node:

rosrun kinova_demo joints_action_client.py -v -r j2s6s200 degree -- 70 0 0 0 0 0

This code will drive the 1st joint of the robot to rotate 70 degrees from its current angle. In the digital version of the document, you can visualise how this command is executed on the ARCHER robot arm, by clicking in the following image (Figure 2.9).

Figure 2.9. Execution of joint position control on the ARCHER robot arm.

By echoing the following two topics, joint position control can be observed, as shown in Figure 2.10:

- */'j2s6s200_driver'/out/joint_angles (in degrees)*
- */' j2s6s200_ driver'/out/state/position (in radians)*

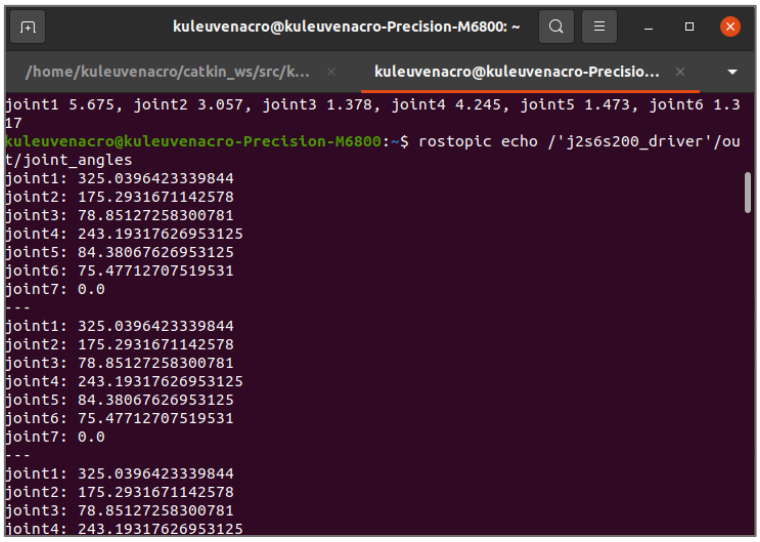

Figure 2.10. Outcome after echoing the topic /'j2s6s200_driver'/out/joint_angles.

The second way to control joint position is by using interactive markers in Rviz, explained in section 2.1.1.2. To do this, following steps need to be taken:

- Launch the driver as stated above.
- Start the interactive node control:

  rosrun kinova_driver kinova_interactive_control j2s6s200

- Open Rviz:

  rosrun rviz rviz

- Once Rviz is open, interactives markers need to be added and the desired topic selected, which in this case would be /j2s6s200_interactive_control_Joint as shown in Figure 2.11. Figure 2.12 shows the interactive markers that allow the robot to be controlled.

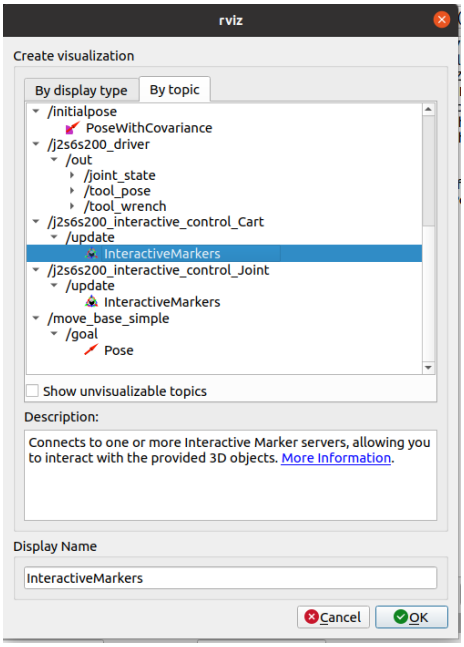

Figure 2.11. Selection of Interactive Markers using RViz.

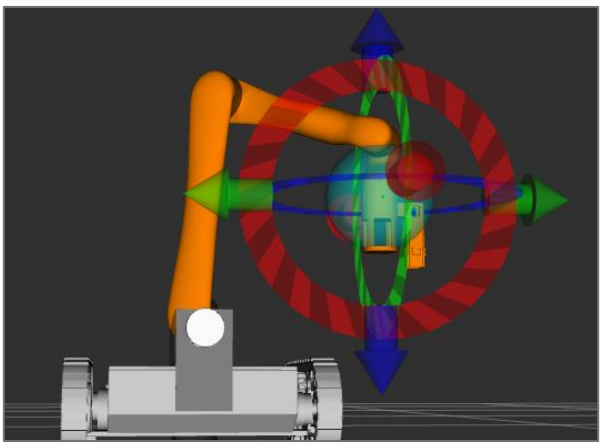

Figure 2.12. Interactive markers to control the robot arm using RViz.

2.2.2.2. Cartesian Position Control

Cartesian control finds the joint configuration required to achieve a position/orientation of some part of the robot. The goal of the inverse kinematics problem is to calculate the

values to be adopted by the articular coordinates of the robot ($q_1,\ldots,q_n$), so that its end-effector is positioned and oriented according to a given spatial location [26].

As with the joint position control, the kinova-ros stack performs the cartesian position control in two different ways, by calling the node pose_action_client.py or running rviz. Both are executed in the same way as Joint Position Control.

2.2.2.3. ROS Service Commands

To send the robot to a pre-defined position, the Kinova-ROS stack provides a service:

rosservice call /'j2s6s200_driver'/in/home_arm

The execution of this command directly moves the robot to a home position. In the digital version of the document, you can visualise how this command is executed on the ARCHER robot arm, by clicking in the following image (Figure 2.13).

Figure 2.13. Execution of service command on the ARCHER robot arm.

2.3. Intel Realsense LiDAR camera L515

The Kinova robot arm implemented on the ARCHER robot platform will be modified to incorporate a LiDAR camera and a probe to determine the radiation of the scanned surface. For the development of this thesis the focus will be on the Intel Realsense LiDAR camera L515. To better understand its operation, a brief description of LiDAR technology will be exposed.

2.3.1. LiDAR Technology

A device incorporating LiDAR (Light Detection and Ranging) technology can measure the distance from a laser emitter to an object or surface using a pulsed laser beam. This distance is calculated by measuring the time delay between the emission of the pulse and its detection through the reflected signal [27].

The L515 uses solid-state LIDAR technology. Traditional LiDAR systems rely on the movement of various parts to obtain precise and accurate measurements, i.e. they are electromechanical. On the other hand, solid-state LiDAR technology does not contain moving parts, it is a system build entirely on a silicon chip. These devices are more resilient to vibrations and can be made smaller than the traditional LiDAR systems [28].

As explained in the Intel Realsense LiDAR camera L515 datasheet [29], the camera "uses an IR (infrared) laser, a MEMS (Micro-Electro Mechanical System), an IR photodiode, an RGB imager, a MEMS controller, and a vison ASIC". The laser beam is scanned over the entire field-of-view (FOV) using the MEMS. The reflected beam data is captured by a photodiode, and processed by the L515 vision ASIC, which will output a depth point representing the exact distance of an image point from the camera. The set of all depth points obtained will generate the point cloud representing the whole scene [29].

2.3.2. Camera Description

The L515 (Figure 2.14) is a solid-state LiDAR depth camera that enables highly accurate depth sensing. As it has been previously explained, its LiDAR technology allows for the creation of a 3D map – or "point cloud" – of the world around the sensor.

It incorporates its own tiny MEMS mirror that allows the laser to scan the scene but at reduced power compared to traditional LIDAR techniques, consuming less than 3.5W power for depth streaming and reducing electronics costs. Despite its low consumption, it has a range between 0.25-9 meters. It has a resolution per depth frame of 1024 x 768 pixels with a framerate of 30 Hz. This equals a resolution of 23 million pixels per second. It has a compact and lightweight enclosure, thus being ideal for robotic applications as it can be easily incorporated in any product. This device does not contain an internal power source; it is powered by a USB type-C port [29].

Figure 2.14. Front (left) and side (right) image of the Intel Realsense LiDAR camera L515. The side view shows the USB Type-C connection port [29].

3. POINT CLOUD PROCESSING

This chapter contains a literature study on point cloud processing. This will be necessary in order to carry out the proposed case study, since it will be based on the work with a point cloud obtained with the LiDAR camera previously described.

3.1. What is a point cloud?

A point cloud is a model composed of a set of points positioned three-dimensionally in space, representing the external surface of an entity. The 3D point cloud contains extensive metric information about the scanned surfaces, as explained in [30] "including each point coordinates along the X, Y, and Z-axes, and sometimes additional data such as a colour value, which is stored in RGB format, and luminance value, which determines how bright the point is" [30] [31].

There are various sensors and technologies through which a 3D point cloud can be obtained. These methods include [32]:

- Laser Scanner: It is also known as LiDAR. A mass data acquisition device, which creates a three-dimensional point cloud generated by measuring angles and distances using a laser light beam.
  According to the field of work, 3D laser scanners can be classified into three categories, namely terrestrial laser scanner (TLS) or ground LiDAR, airborne laser scanner (ALS) or aerial LiDAR, and mobile laser scanner (MLS) also known as mobile LiDAR.
- Digital photogrammetry: It uses multiple images of an object from different angles to generate a high metric quality 3D point cloud of the object. Stereo cameras use photogrammetry, consisting of two or more lenses with a separate image sensor. By knowing the relative position and orientation between the two lenses, a 3D point cloud can be obtained based on 2D images.
- Videogrammetry: This technology works in a similar way to photogrammetry, but instead of taking a set of images as input data it takes sequences of videos. It allows the point cloud to be reconstructed progressively, basing the information of each frame on the previous frame.
- RGB-D camera: It consists of an RGB camera and a depth sensor. The RGB camera takes the images, and the depth sensor determines the depth information of each pixel. By mapping the RGB images together with the sensor information, a coloured point cloud is generated. An example of this technology is the Intel Realsense LiDAR camera L515.

Point clouds, such as the example shown in Figure 3.1, can be treated in two ways once they are obtained, either rendered and inspected directly or converted into models using various shapes and patterns.

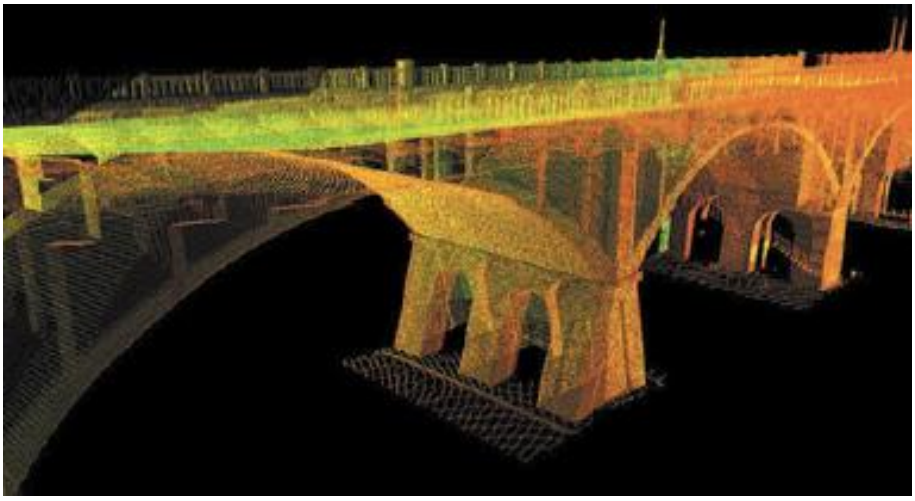

Figure 3.1. Example of a 3D image obtained through a point cloud [33].

The main purpose of a point cloud is to create a 3D model. Visualising the data into a 3D mesh organises the points and sets a foundation that can be used to build a model. Exporting the point cloud creates a file that can be imported into a CAD or BIM system. The point cloud format depends on the software that is used. Some of the most common are [31]:

- PTS: Open format for 3D point cloud data. Since it is open, anyone can make use of it.
- XYZ: Archetypal ASCII (American Standard Code for Information Interchange) format. Compatible with many programs, but lacks unit standardisation, which makes data transfer difficult.
- PTX: It is also an ASCII format. It can only work with organised point clouds and usually stores data from LiDAR scanners.
- LAS (LASer): It is an open format for LiDAR scanning data. It combines GNSS (Global Navigation Satellite System) data, laser pulse range information and Inertial Measurement Units (IMU) to create data that fits on the X, Y and Z axes.
- PLY: Polygon File Format, stores data from 3D scanners. It includes properties such as colour, texture and transparency. It can also contain 3D mesh data.

3.2. Operations performed on a point cloud

The processing of three-dimensional data is the work performed after capturing a point cloud. The processing techniques have many different objectives, from improving the captured data by means of various algorithms or statistical techniques to obtaining relevant information according to the objective. The following is a brief review of some of the most important techniques that can be used to process three-dimensional data.

3.2.1. Visualisation

Once the point cloud has been captured, the first thing that is usually done is visualisation. Although it is the most basic operation, it allows for an initial assessment of the quality of the dataset obtained, as well as the planning of the processing scheme to be followed. Finally, visualisation will allow the observation of the final result [34].

A large majority of point cloud processing programs have a graphical interface that allows the conversion of the point cloud into an image. The simplest form of visualisation allows the observation of the points with a single colour and size, as well as zoom and rotate operations, to observe the cloud from different perspectives. However, there is more advanced software that allows each point to be rendered according to different characteristics. Points can be encoded in brightness according to the intensity of the laser return, or with RGB texture. They can also be colour-coded according to attributes contained in the point cloud structure, such as rank or class.

Among the most important operations that can be performed during the process of visualising a point cloud, the following are highlighted:

- Single point selection. Visualisation allows the selection of individual points within the cloud, using zoom and rotation controls.
- Measurements. Precise point selection provides the option to measure distances between points and to determine the angles between the lines joining the points.

3.2.2. Segmentation

The segmentation process is based on the division of the point cloud into different zones, groups of points called clusters, which is why it is also referred to as clustering [35].

These algorithms are particularly useful when the cloud is made up of several isolated regions, i.e. this process allows the cloud to be broken into its constituent parts so that they can be processed independently. It is a technique commonly used in object recognition. There are numerous segmentation techniques, among which we will explain the following ones:

3.2.2.1. Euclidian segmentation

Euclidean segmentation is the simplest of all. It is based on checking the distance between two points. If this distance is less than a certain threshold, both are considered to belong to the same cluster. The algorithm works as follows: a point is selected from a cloud and its neighbours are selected as part of the same cluster, until no new point can be added. Then, a new cluster is initialised, and the procedure starts again with the remaining unmarked points. It is an iterative process that ends when all points in the cloud have been assigned to a cluster [36].

3.2.2.2. Region growing

This is a type of segmentation that groups points that verify a smoothness constraint. It is classified as a pixel-based segmentation method, since it requires the selection of initial points, called seed points [37].

It examines the neighbouring pixels of the seed points and determines whether they should be added to the cluster. To check whether two points belong to the same smooth surface, the angle between their normals and the difference in curvatures are checked. An example of the use of this algorithm is shown above (Figure 3.2).

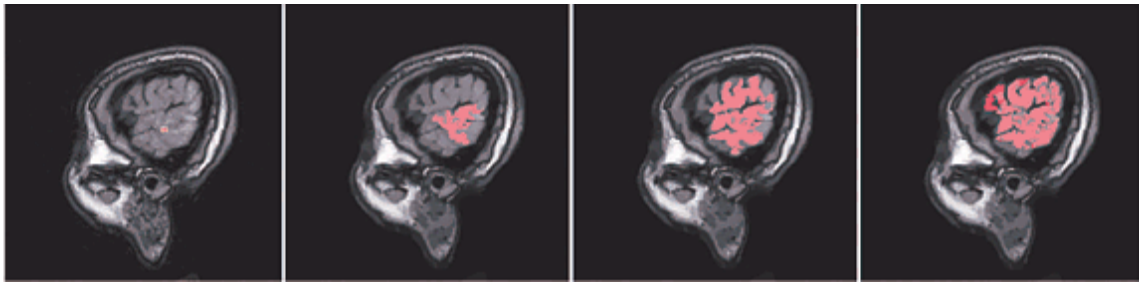

Figure 3.2. Example of the use of the region growing segmentation algorithm [38].

3.2.2.3. Min-cut

The min-cut or minimum cut algorithm performs a binary segmentation, dividing the point cloud into two clusters: one containing the points belonging to the object of interest (foreground points) and another with points that do not belong to the object of interest (background points) [36]. An example can be seen in Figure 3.3.

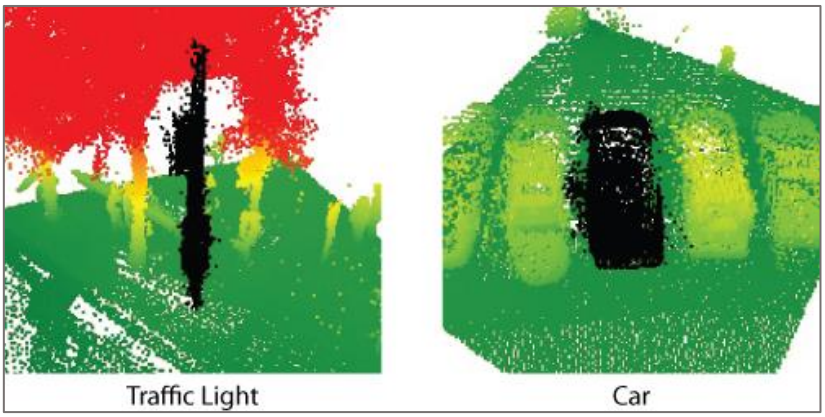

Figure 3.3. Example of the use of the min-cut algorithm where the black points represent the objects of interest, traffic light and car, respectively [39].

The algorithm uses a vertex graph, in which each vertex represents a point, together with two additional vertices that will be connected to each other with edges with different penalties (weights). Subsequently, edges are also established between the neighbouring points, whose weight value depends on the distance separating them. The algorithm will need as input a point in the cloud that is known in advance to be the centre of the object and the radius. The minimum cut will be found, and a list of foreground points will be available.

3.2.3. Filtering

The filtering process is based on the selection of a subset of the data which is considered to be the one that will provide relevant information with respect to the final objective. The aim is to discard any other superfluous data, thus reducing the data set to work with.

The filtering process can be carried out in multiple ways, which can be manual, semi-automatic or automatic. The filtering criteria can be very diverse and depend to a large extent on the information provided by the data capture device used.

The ultimate goal of the filtering process is to have as little data as possible with as little loss of relevant information. This allows a considerable reduction of the computational load when proceeding with the point cloud analysis [35].

Within the filtering techniques the following can be found.

3.2.3.1. Resampling

Resampling aims to modify the number of points in a cloud, either by increasing (upsampling) or decreasing (downsampling) them. One or the other will be used depending on the desired objective [40].

- **Downsampling**

Sensors currently available provide clouds with high resolution. While this means a better result, it also leads to a higher computational load. One option to avoid this problem is to reduce the number of points in the cloud by eliminating those that are not needed for the final goal. There are several methods for this process.

A common way of doing this is downsampling, which produces a cloud equivalent to the original but with a smaller number of points. Downsampling is done using a voxel grid. The cloud is divided into several cube-shaped regions, called voxels, given a desired resolution. The next step is the processing of all points of each voxel so that only one of them remains.

To make the algorithm work more accurately, instead of selecting a random point within each voxel, the centroid of the voxel, i.e. the point whose coordinates are the mean values of all the points in the voxel, can be calculated. Figure 3.4 shows the treatment of an original point cloud on which downsampling has been performed with different resolutions. It can be seen that, depending on the resolution ratio, the result differs significantly from the original cloud, so the final objective must be taken into account when selecting it.

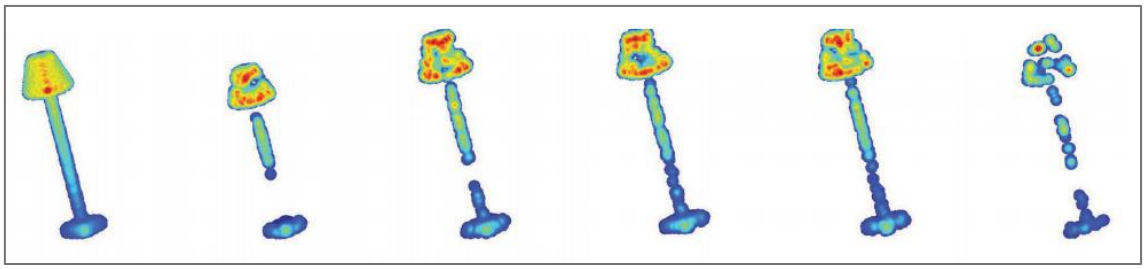

Figure 3.4. Cloud treated with the downsampling algorithm using different resolutions [38].

- **Upsampling**

Upsampling is a form of surface reconstruction, performed when more points than currently possessed are needed. It is based on the interpolation of the points already available to generate new ones and is, th berefore, not a very sophisticated approach to surface reconstruction, as it does not provide very accurate results.

3.2.3.2. Outlier removal

Outliers are considered undesirable noise in the image as they incorporate errors in the operations performed on the point cloud. Consequently, they need to be removed from the point cloud, so that calculations are performed faster, and more accurate results are obtained. This procedure can be done in a variety of ways, including [40]:

- Radius-based: In this algorithm, a search radius and the minimum number of neighbours that a point must have to be considered outlier must be specified. It iterates through all the points in the cloud, checking whether they are outliers. If less than the specified number of points are found within the search radius of that point, it is considered an outlier and is eliminated from the cloud.
- Statistical: There is more than one statistical outlier remover, however only one will be explained. It requires as input data, in addition to the point cloud to be processed, the number of nearest neighbours to a point and the deviation multiplier. The algorithm works as follows, for each point the mean distance to its K neighbours is calculated. Considering that the result follows a Gaussian normal distribution with mean μ and standard deviation σ, all points with mean distances that fall out of the global mean plus deviation can be removed. Figure 3.5 shows the before (left image) and after (right image) of the application of an statistical outlier removal algorithm on a point cloud.

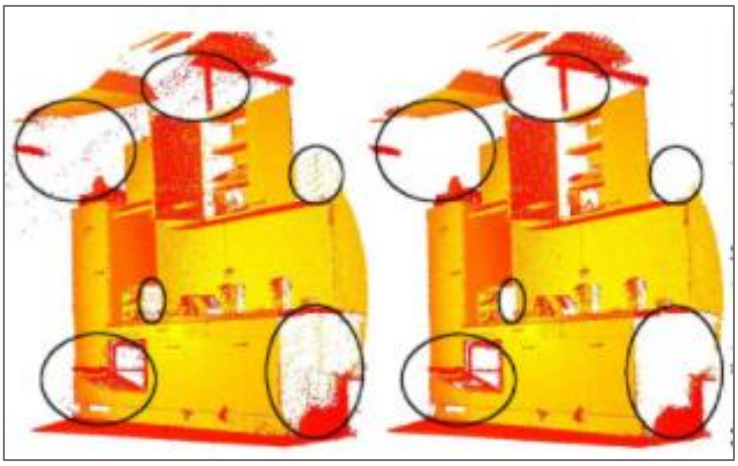

Figure 3.5. Demonstration of outlier removal filter used in a point cloud [41].

3.2.4. Transformations

A very large number of different transformations can be applied to point clouds, including the following [34].

- Translation and rotation: A point cloud can be translated or rotated on one or more coordinate axes.
- Cropping: When scanning an object, there are always certain points that do not belong to the volume of interest. Cropping gives the possibility to remove these points from the 3D space.
- Merging: This process is performed when several point clouds of the same object are obtained from different angles or positions, each in its own coordinate system, and a single coherent point cloud needs to be defined. A point cloud is established as a base reference frame and then common points between the base point cloud and the source are identified. Figure 3.6 shows an example of a point cloud merging application. These clouds were obtained from different scanning angles, the pink point cloud (middle) was rotated to the coordinate system of the white point cloud (left) to obtain a final point cloud (right).

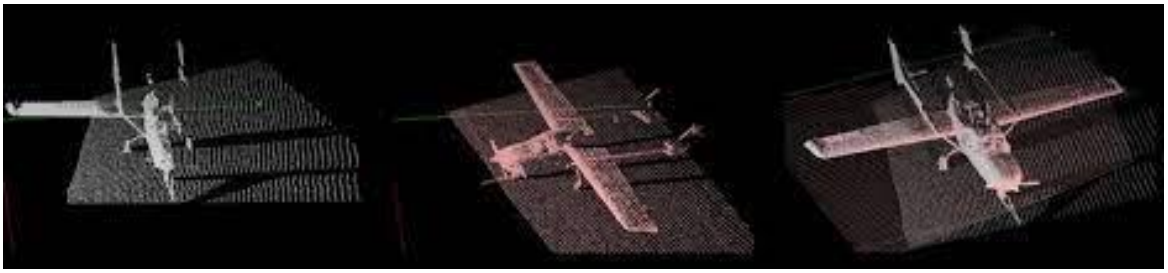

Figure 3.6. Example of a point cloud merging application [34].

3.2.5. Shape recognition

Next, we will discuss parametric shape detection algorithms. These are of special importance in this thesis, since they will be required for the analysis of the point cloud

used in the case study, where a plane needs to be fitted. That is why they will be analysed in detail, trying to find the main advantages of each one of them and their main uses.

There are algorithms based on the search for parametric shapes within a point cloud, i.e. a plane, a sphere, a cylinder etc. Two main algorithms will be analysed and compared: the Hough transform and RANSAC.

3.2.5.1. Hough Transform

Hough Transform is an algorithm used for the isolation of specific shape features in images. The most basic Hough transform detects straight lines (line segments), but it is also used for the detection of objects, such as planes in a point cloud [42] [43].

This is a statistical algorithm and according to the points that are available, the possible lines/planes on which the point can be located are to be found out. It makes use of a parametric representation of geometric form.

To explain the implementation of the algorithm, we will use the detection of a straight line, since it is simpler than a plane.

A line can be represented using the equation 3.1:

$$\rho = x * cos(\theta) + y * sen(\theta) \qquad (3.1)$$

the representation of a line using its polar coordinates, with ρ being the shortest distance between the line and the origin, and θ the angle of the vector from the origin to the nearest point of the line, as shown in the Figure 3.7.

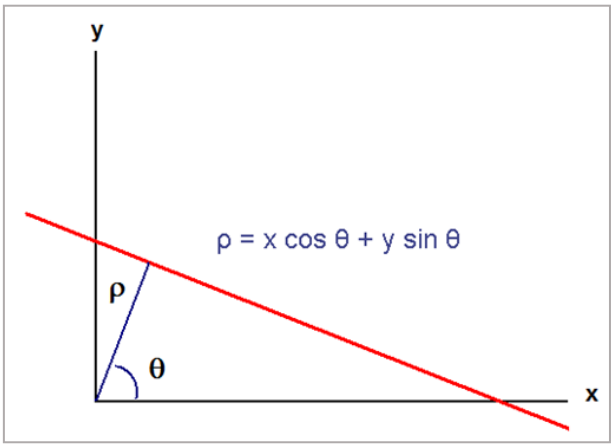

Figure 3.7. Parametric representation of a line with its representative parameters.

It is then possible to associate with each line a pair of coordinates (ρ, θ). This generates a space, called Hough space, for the set of straight lines in two dimensions. Each point in the image corresponds to a single sinusoidal curve in Hough space, because it represents all the lines that can be drawn through this point. If, being in Hough space, the curves

corresponding to two points intersect, this will correspond to a line in image space passing through these two points.

The set of points forming a line will produce sinusoids intersecting at the parameters of that line. Thus, the problem of detecting collinear points can become a problem of finding concurrent curves. Figure 3.8 shows a line in which three points are defined by their x-y coordinates and the corresponding representation of these points in Hough space, corresponding to each of the three half-sinusoids. The point where the curves intersect in Hough space, gives the distance and the angle which define the line intersecting the points.

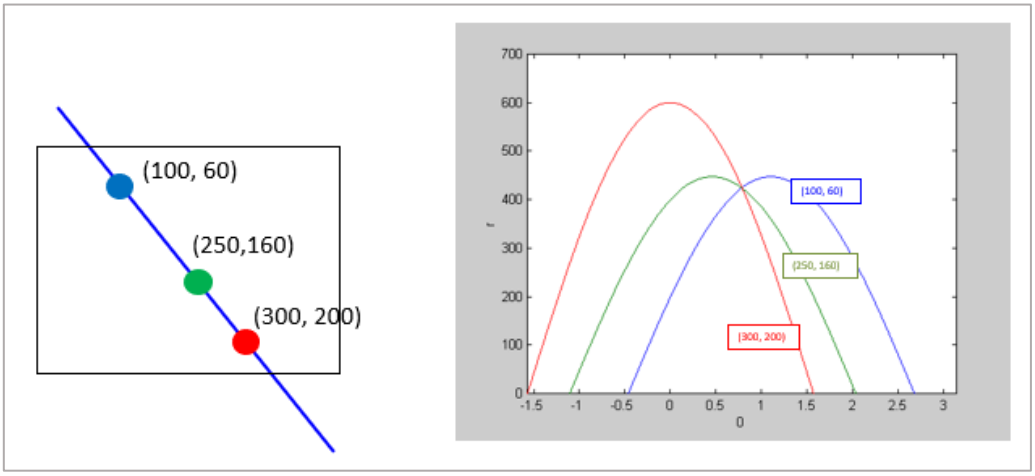

Figure 3.8. A line on which 3 points are defined (left) and a representation of these points in Hough space (right). The parameters of the intersection point of the three curves are the ρ and θ values of the line connecting the three points [44].

In the example above there are no outliers because it is a predefined line. However, it is common to work with images where multiple outliers are present. In this case the Hough transform uses the voting system. The implementation is carried out as follows, first the parameter space is subdivided into vote accumulator cells, as can be seen in Figure 3.9 [44].

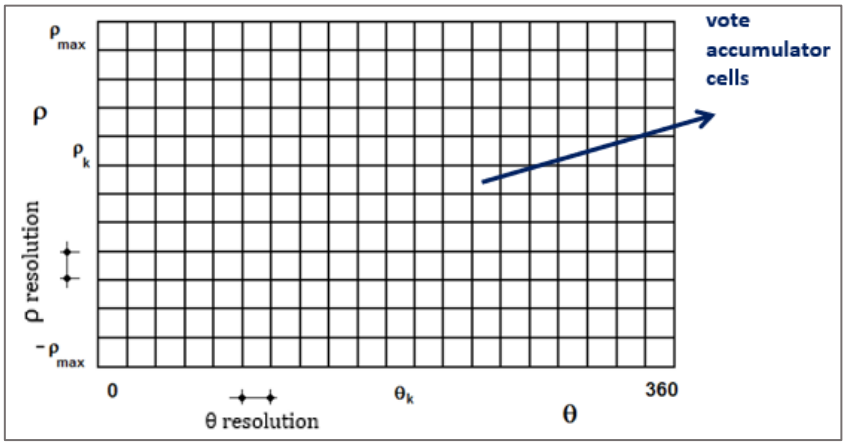

Figure 3.9. Hough space subdivided into vote accumulator cells [44].

Now each pixel (x, y) must vote for the cells of all the lines passing through it, as shown in Figure 3.10.

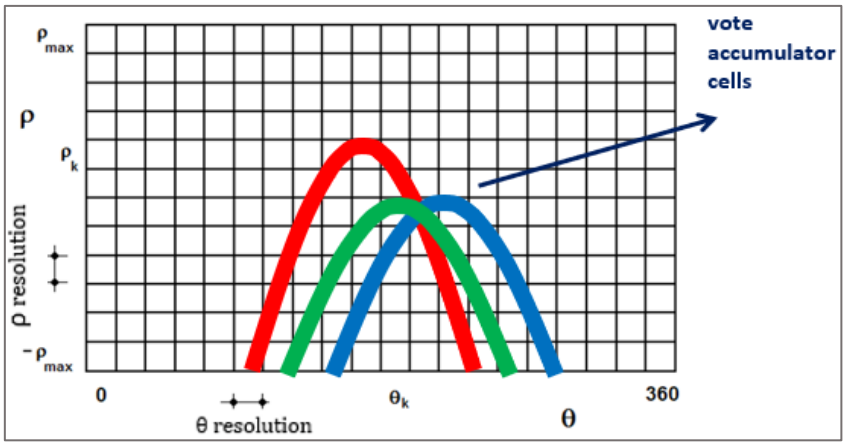

Figure 3.10. Each pixel (x, y) votes for the cells of all the lines passing through it [44].

A cell (ρ_k, θ_k) with many votes indicates that the line with these parameters passes through many points in the image. Figure 3.11 represents where the cell containing most votes is located.

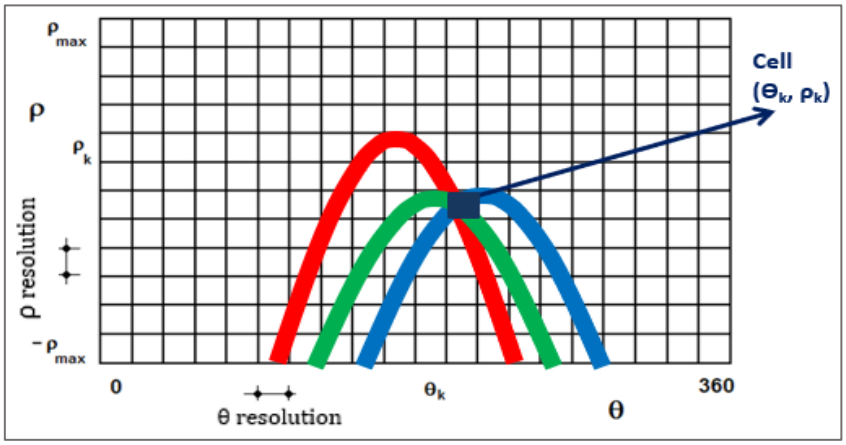

Figure 3.11. A cell (ρ_k, θ_k) containing many votes indicates that the line with these parameters passes through many points in the image [44].

Figure 3.12 represents a set of straight lines with respect to an XY coordinate system (left image) and the Hough space after having made the voting system (right image) showing the six points corresponding to the straight lines on the left image. These six highlighted points are the most voted cells.

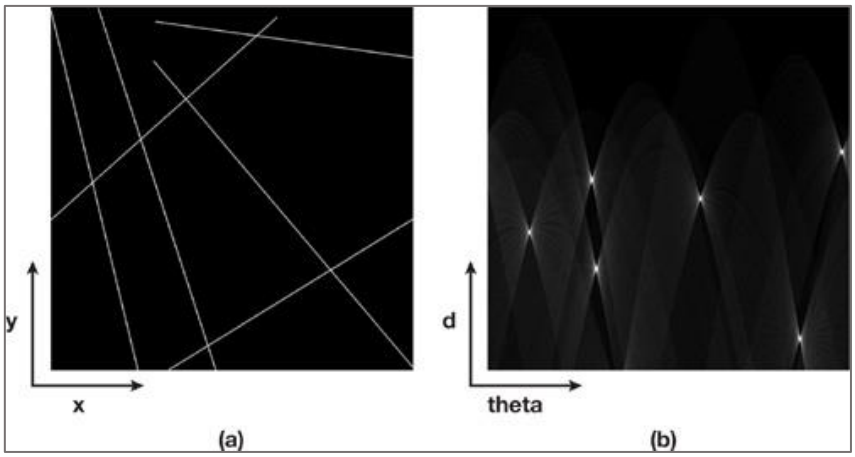

Figure 3.12. a) A set of six lines represented in an XY plane.b) Hough space in which the six points corresponding to the most voted cells are highlighted [42].

3.2.5.2. RANSAC Algorithm

RANSAC (RANdom SAmple Consensus) is an algorithm introduced by Martin L. Fischler and Robert C. Bolles in 1981. It is an iterative algorithm used to estimate the parameters of a mathematical model from a data set containing outliers [45].

It is a nondeterministic algorithm, in the sense that it produces a correct result only with a given probability; to increase this probability, the number of iterations must be increased.

The basic assumption for the operation is that the data consists of a set of inliers, i.e., data whose distribution can be characterized by the parameter set of a model, and a set of outliers, being data not represented by that model. Outliers can come, for example, from extreme noise values, erroneous measurements, or incorrect assumptions about the interpretation of the data. This algorithm assumes that, given a dataset containing outliers, there is a procedure that can estimate the parameters of a model thus, maximising the number of inliers, and optimally representing the data.

The operation of the RANSAC algorithm will be briefly explained using a set of points in 2D as a starting point [46]. The first step is the selection of a random sample of the minimum size necessary to fit the model, as can be seen in Figure 3.13.

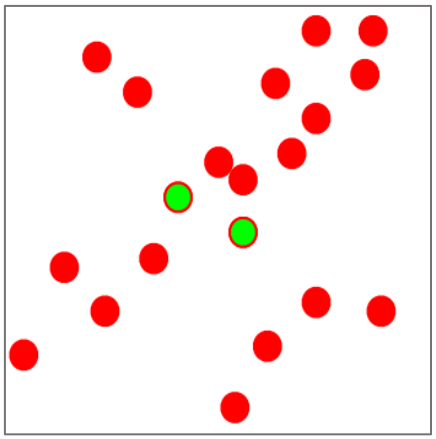

Figure 3.13. Selection of a random sample of the minimum size necessary to fit the RANSAC model [46].

Figure 3.14 shows the next step, which is the computation of a possible model from the sample set.

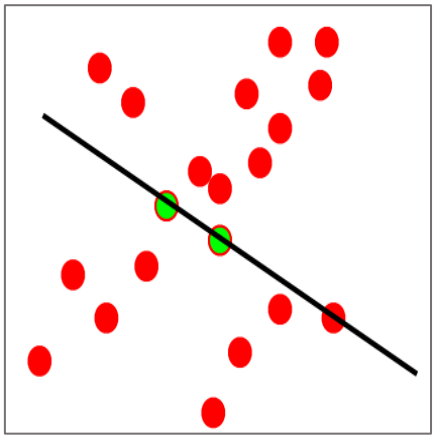

Figure 3.14. Putative model from the sample set [46].

The last step, shown in Figure 3.15, is the calculation of the set of model inliers from the whole data set.

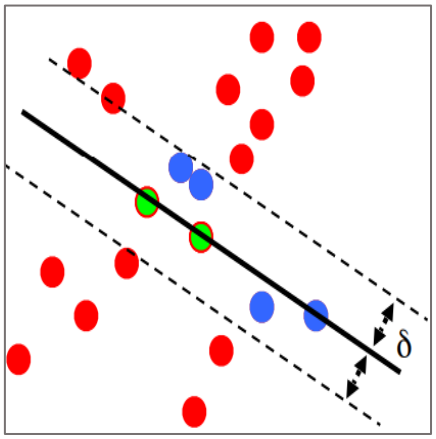

Figure 3.15. Calculation of the set of model inliers from the whole data set [46].

These steps are repeated iteratively until the model with the highest number of inliers is found, which for this data set is shown in Figure 3.16.

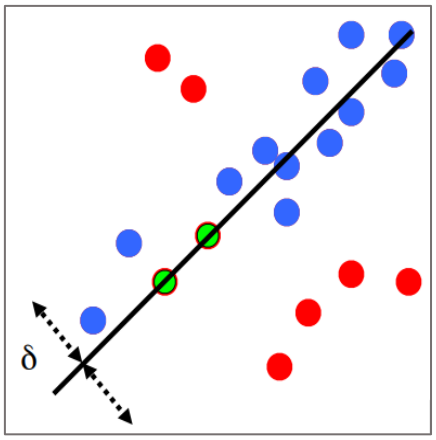

Figure 3.16. Model found containing the highest number of inliers [46].

3.2.5.3. Comparison between the two algorithms

The Hough Transform offers the following advantages. Firstly, all points are processed independently, therefor it presents robustness to the presence of outliers. It is also quite resilient to noise, which does not greatly affect the outcome. Moreover, it allows the detection of multiple elements in an image, and it has a moderate computational cost. However, in the presence of uniform noise, erroneous results can be obtained [46] [44].

On the other hand, the RANSAC algorithm is straightforward and simple to apply. It is very robust against outliers and is suitable for the definition of any parametric shape. However, it does not always guarantee convergence to global optima, which may result in failure of the algorithm [46] [44].

These algorithms have been studied since the case study will require the definition of a plane through a point cloud. After the analysis carried out and taking into account that the work is developed using an image of a wall, RANSAC has been considered to be the best algorithm. This selection was based on the simplicity of the processed point cloud, which, being a wall, does not present excessive outliers. This allows the definition of the plane to be carried out using a simple algorithm, namely RANSAC.

Also, since the PCL point processing library will be used, it provides an already integrated algorithm to apply RANSAC on the cloud. If Hough were decided to use with this library, it would be necessary to develop the algorithm beforehand, thus requiring more time.

3.2.6. Shape analysis

As a last operation, region-based description using geometric descriptors (concave and convex area) is going to be explained.

3.2.6.1. Retrieving the hull

A hull can be defined as the set of points that conform the outermost boundary of the cloud [36]. Two types of hulls can be calculated:

- Concave Hull. It is a polygon which embraces all points but normally takes less area than the convex hull [36]. An example of concave hull can be seen in the left image of Figure 3.17.
- Convex Hull. A part C of a vector space is convex if [47] "for every pair of points of C, the segment joining them is totally included in C; that is, a set is convex if it is possible to go from any point to any other point in a straight line, without leaving the set". The convex hull of a set of points is shown in the right image of Figure 3.17.

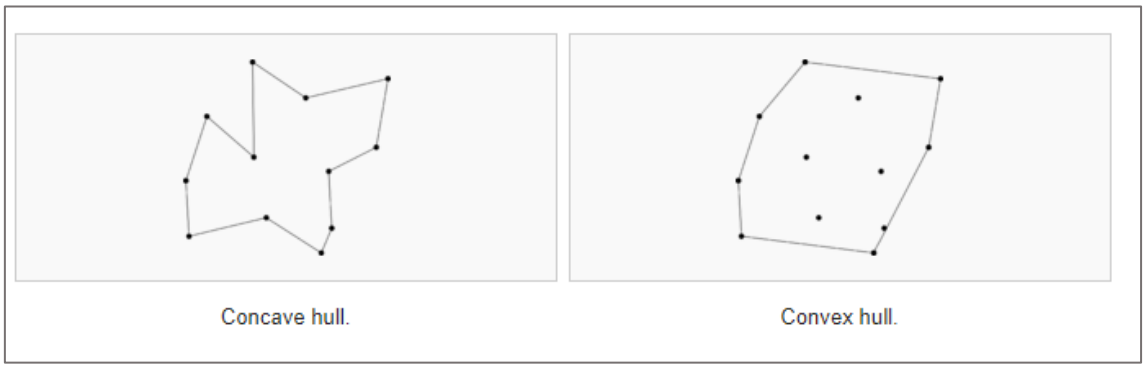

Figure 3.17. Concave and convex hull extraction from a set of points [36].

The creation of a convex or concave hull can be useful when you need to simplify the representation of a surface or want to extract its boundaries.

3.3. 3D Point cloud data processing libraries

The last part of this literature study will be based on the analysis of the different libraries currently available for point cloud processing. Since the selection of one of them will be necessary, after the analysis of the existing options, the reasons for the selection of the chosen library will be presented.

3.3.1. PCL

PCL (Point Cloud Library) is a C++ library focused on the processing of N-dimensional point clouds, developed by Willow Garage with the aim of performing processing with numerous techniques. This library has algorithms to, among others, apply filters, estimate functions, reconstruct surfaces, and fit and segment models. It is released under BSD license, that is, it is free for commercial and research use. Funding and support for this library is provided by major companies such as Nvidia, Google, Toyota, Trimble, Urban Robotics, Honda Research Institute and Sandia Intelligent Systems and Robotics [48].

PCL runs on various platforms such as Windows, Linux, MacOS, and Android. Through its tools, it offers the necessary potential to process and reconstruct a three-dimensional scene in a simple way. Figure 3.18 shows the PCL logo.

Figure 3.18. PCL (Point Cloud Library) logo [48].

3.3.1.1. Description

The PCL library is a synthesis of multiple algorithms and functions that allow working with images and point clouds in 2 or 3 dimensions. Structurally, this library is divided into a series of small libraries that can be compiled and used independently. This is one of the great advantages of PCL since two goals can be achieved in this way [48].

On the one hand, development is simplified because small pieces of code are easier to maintain than large libraries. On the other hand, it allows the use of the library on platforms with reduced specifications in terms of capacity or computing power. The following is an example of the division of PCL by means of a dependency network (Figure 3.19).

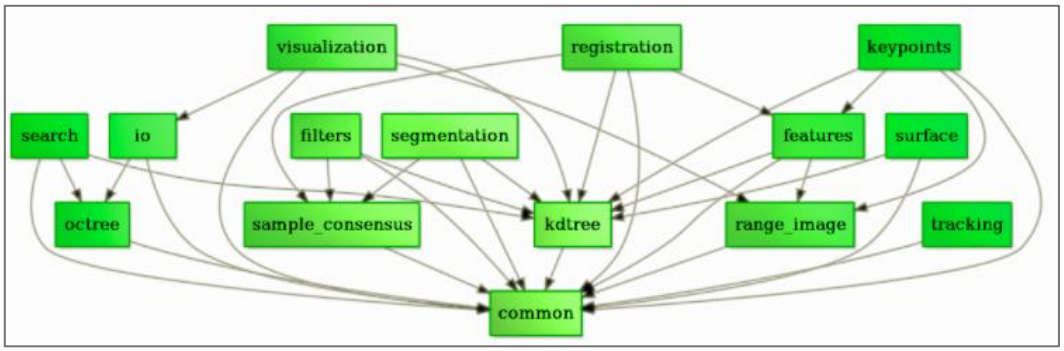

Figure 3.19. Dependency network PCL Division [48].

The main modules of the library are the following: filters, features, keypoints, registration, kdTree, octree, segmentation, sample consensus, surface, range image, I/O and visualisation.

3.3.2. Open3D

Open3D (Figure 3.20) is an open-source library that aids the development of software that deals with 3D data. Its interface contains a set of data structures and algorithms in both C++ and Python [49].

Data structures are available for three types of representations: point clouds, grids and RGB-D images. For each representation, a set of basic processing algorithms has been implemented, such as I/O, sampling, display and data conversion. Algorithms generally used in point cloud processing have also been included.

Figure 3.20. Open3D logo [49].

Open3D consists of 9 modules:

- Geometry, it implements three geometric representations: point cloud, triangle mesh, and image.
- Camera, its objects can be visualized in the 3D scene.
- Odometry, it allows tracking and alignment of RGB-D images.
- Registration, it provides implementations of multiple surface registration methods.
- Integration, it contains volumetric integration.
- I/O, reading and writing 3D data files.
- Visualisation, allows rotation, translation, and scaling via mouse operations.
- Utility provides support functions such as the file system.
- It provides Open3D Python tutorials.

3.3.3. PDAL

PDAL (Point Data Abstraction Library), Figure 3.21, is an open-source C/C++ library containing applications for translating and processing point cloud data. Although many of the library's tools have their origin in LiDAR, it is not only limited to the processing of LiDAR data [50].

Figure 3.21. PDAL (Point Data Abstraction Library) logo [50].

PDAL allows you to compose operations on point clouds into pipelines of stages. To write these pipelines a declarative JSON syntax can be used, or they can be built using the available API.

The foundation of PDAL is the concatenation of a set of components, each of which will provide a specific functionality. These components allow for reuse, compounding and separation. This library considers point cloud processing operations as a pipeline composed of a set of stages. To perform a given operation, instead of writing a single specialised program, it can be defined as a sequence of steps or operations.

PDAL allows users to apply several algorithms on data without having to worry about data formatting issue. It has a number of applications that allow users to coordinate and build point cloud processing workflows. Some of the tasks that users can perform are:

- Printing information about a dataset.
- Translation of data from one point cloud format to another.
- Application of exploitation algorithms. These include noise removal and reprojection from one coordinate system to another.
- Merge or split data.

3.3.4. Library selection

After the analysis of the different libraries, a decision had to be made on the selection of the one to be used for point cloud processing in the case study. Initially, the advantages and disadvantages of using each of the libraries will be presented.

The main attraction of PCL library is its extensive list of functionalities as well as its powerful processing capability. It has multiple tools for working with point clouds and three-dimensional models. It is developed in C++ a language often used to develop software. PCL has extensive documentation on its website. It has multiple tutorials to work with that help to handle teach the basic concepts of each module, including example code and various compilation possibilities [35].

Open3D is a complete library with many available features. It allows installation on different platforms and easy compilation of the source code. It makes use of a code review mechanism to keep the code clean and consistent styled. However, it is relatively new, therefor it is still under development. It uses Python as the main interface, providing most tutorials in this language [49].

PDAL can work with any point cloud storage format. It is an open source-source project with all its activities available online. By using a content-abstracted API, it allows users to apply algorithms to the data, which frees them from worrying about data formatting issues. It also has an easy-to-use command line. Its website states that [50] “developers get the freedom to access (…) the most complete set of point cloud format drivers in the industry”. Nevertheless, it is intended especially for processing large files. PDAL does not provide a friendly GUI interface, its users must have the confidence to work autonomously among the options of filters, readers and writers. Furthermore, PDAL is considered to be a complementary to PCL rather than a substitute, as PCL is more focused on algorithm development, robotic and computer vision, and real-time laser scanner processing.

The first thing to consider when selecting the library is the objective of this project in terms of point cloud processing. The objectives sought are the determination of a plane through the initial point cloud, its translation and a process that allows to select certain points to carry out the scanning. Seen in a general way, it is a simple process, therefore no very complex tools are required. Also, it is important to note that my programming

skills are in C++ and that this study only analysis and processes a small point cloud. Given that work in this field has been started without any previous experience, the most straightforward library needs to be selected.

Following this analysis, the library selected for the development of the process is PCL. It is considered to be an extraordinary tool for the realisation of software projects related to point clouds. It simplifies the work to be done, allowing more time to be spent on the development of the application, instead of having to program the mathematical bases and algorithms necessary to work with point clouds.

4. PRACTICAL DEVELOPMENT

The following chapter presents the case study that has been proposed for the development of the thesis. Once the operation and working environment of the robot arm is known, the next step is to work with the Intel Realsense LiDAR camera L515, which, together with a Kromek probe, will oversee scanning and detecting radioactively contaminated spots on walls. As explained in the introduction, due to time limitations, work with the probe will not be carried out in this thesis.

The next step is to work with the Intel Realsense LiDAR camera L515 to identify surface, following this identification the surface can be scanned for contaminated spots using the Kromek probe.

Figure 4.1 shows the steps that will be taken throughout the case study.

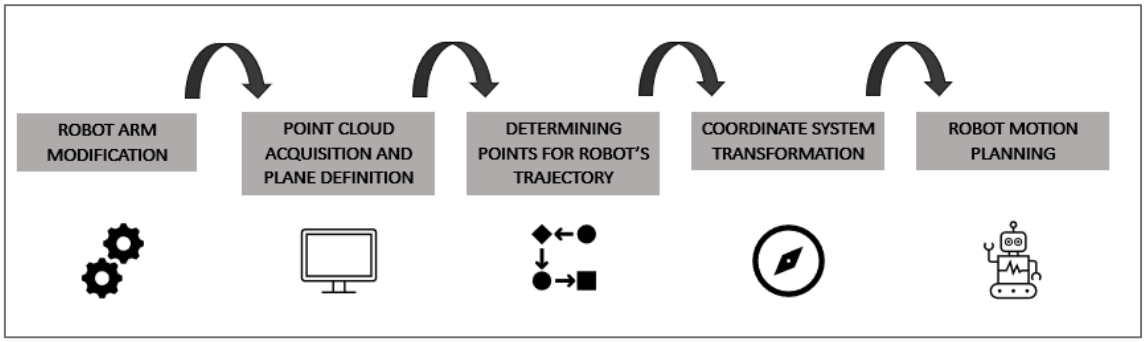

Figure 4.1. Steps taken in the case study.

4.1. Robot arm modification

The first step that needs to be carried out is the modification of the predefined structure of the robotic arm. The L515 camera and probe need to be attached to the robotic arm. For this, a 3D printed mount is created. It should be noted that this mount was designed by an engineer prior to the development of the thesis.

As a result, it is also necessary to modify the existing URDF file of the robot, as the gripper is to be removed and the camera and probe mount are to be positioned in its place. The URDF files have been explained in section 2.1.1.2 of the thesis.

The aim is to set a STL mesh file to one of the links of the robot arm, this link will be the 3D printed model in which the camera and the probe will be mounted.

Attached below is a series of perspectives of the model (Figures 4.2 and 4.3). The circular surface is where the Intel Realsense LiDAR camera L515 will be located and the Kromek probe will be positioned on the adjacent surface.

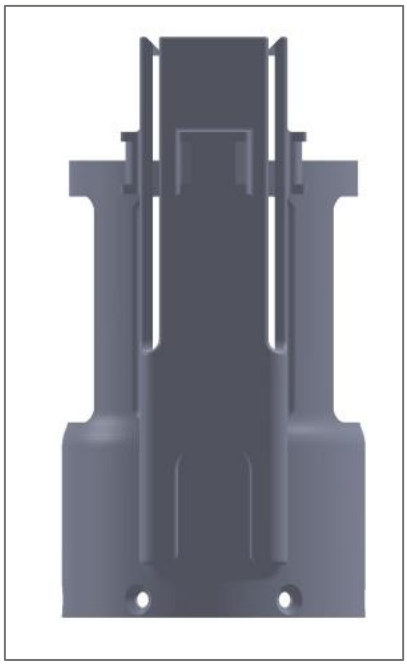

Figure 4.2. View of the area where the probe would be inserted in the 3D model of the mount.

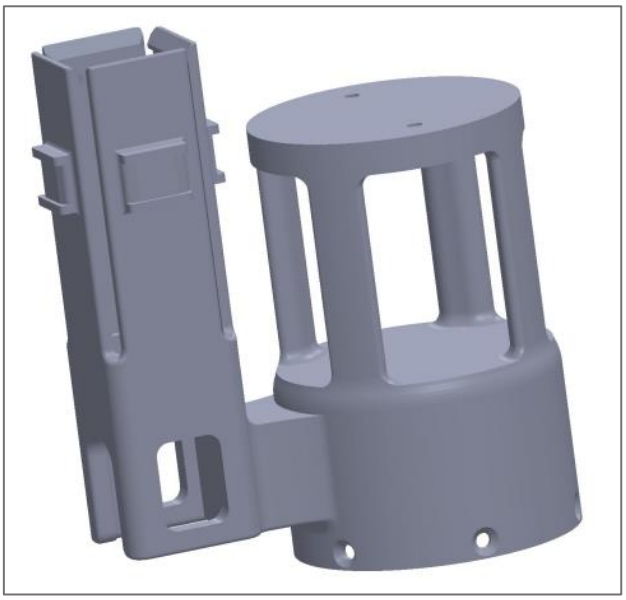

Figure 4.3. Overview of the 3D model developed to incorporate the Intel Realsense LiDAR camera L515 (circular surface) and Kromek probe (rectangular surface) into the Kinova robot arm.

Figure 4.4 shows an image of the mount placed on the real robot. As explained above, the original gripper has been removed and replaced by the 3D mount.

Figure 4.4. 3D printed model included in the robot arm located on the ARCHER robot.

Once the URDF file was modified, the final model of the ARCHER robot could be visualised using rviz, as can be seen below (Figure 4.5).

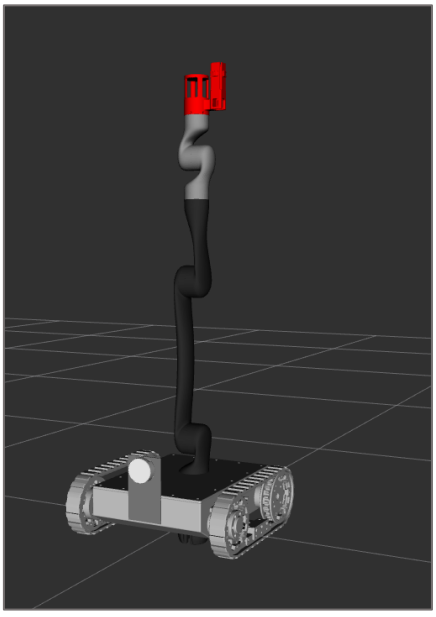

Figure 4.5. ARCHER robot model visualised using RViz.

4.2. Point cloud acquisition and plane definition

The next objective of the thesis is to obtain a point cloud of a wall with the LiDAR camera and define a plane through this point cloud.

To simplify the task, the image will be taken without connecting the camera to the robot, a step that will be left for a later stage once the correct functioning of the designed algorithm has been verified. However, due to time limitation it has not been possible to proceed with this step in this thesis.

4.2.1. Point cloud acquisition

The first step to start working with the camera is to install the Intel RealSense SDK (Software Development Kit) 2.0 [51]. This will make it possible to work with the Intel Realsense Viewer, through which the desired point cloud can be captured.

The SDK can be obtained through the official Intel Realsense website. Figure 4.6 shows the Realsense Viewer with the L500 Depth and the RGB camera connected.

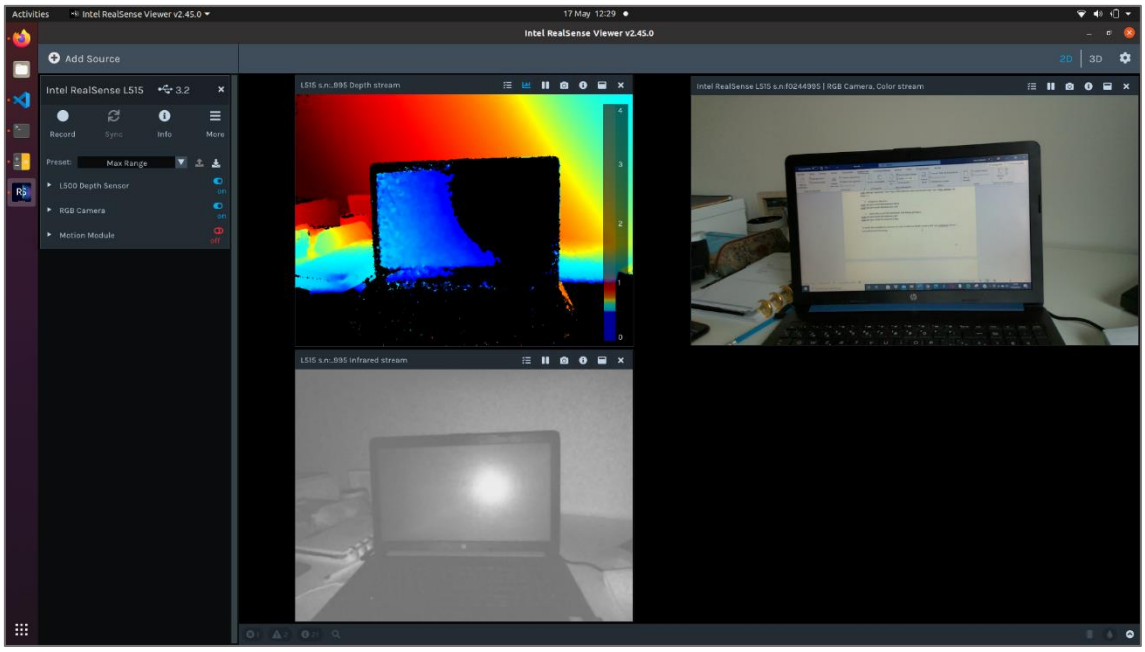

Figure 4.6. Intel Realsense Viewer L500 Depth Sensor and RGB Camera connected.

With the Realsense camera connected to the computer, using a USB cable, both the "L500 Depth Sensor" and the "RGB Camera" need to be switched, as shown in Figure 4.7. [52].

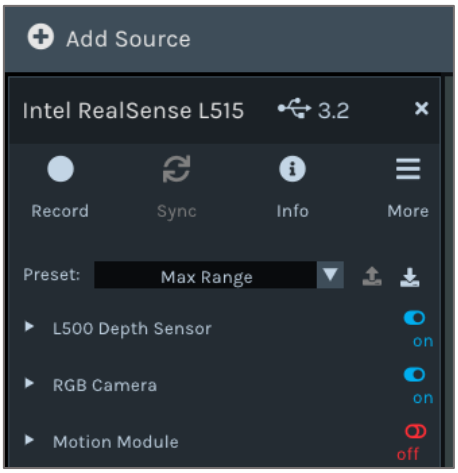

Figure 4.7. L500 Depth Sensor and RGB Camera activated using the Intel Realsense Viewer.

The depth view is color-coded to show the depth. Blue is closer to the camera and red is further away. As for Figure 4.6, the computer is closer to the camera while the wall is further.

Now, the 3D view on the top right corner of the screen must be activated, leading to visualise the point cloud seen in Figure 4.8.

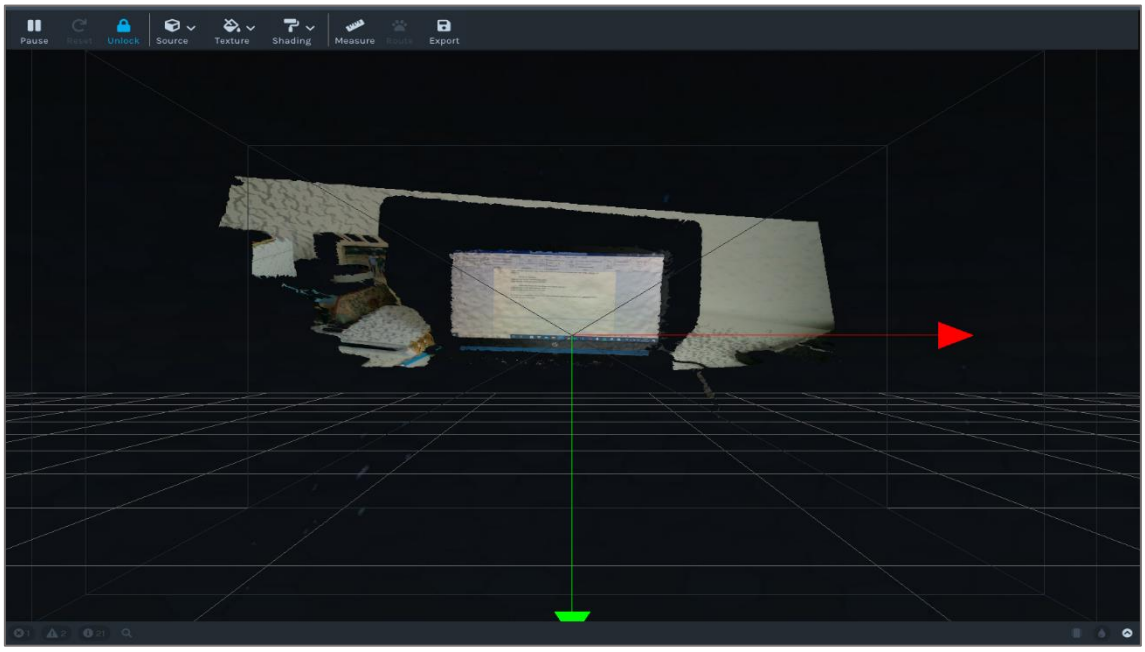

Figure 4.8. 3D View using the Realsense Viewer.

To generate a coloured 3D point cloud, the depth data and colour information are combined. Different perspectives of the object can be seen by dragging the mouse in the 3D view. Selecting the save icon ("Export 3D Model to 3rd-party-aplication"), in the top right corner, allows saving the point cloud in PLY format, which is a simple format for storing captured 3D data, as it has been previously explained in section 3.1 of the literature study.

For the first tests, instead of taking the image with the robot's built-in camera, the image was taken with the stand-alone camera, i.e. simply connected to the computer via a USB cable. To facilitate the process, an image of a random flat and smooth wall was taken. It is with the point cloud obtained from this image that all following steps will be explained.

As concluded in section 3.3.4, the obtained point cloud will be processed using the PCL library. The code will be programmed in C++11.

4.2.2. Code description for plane definition

The first thing to do is defining the point clouds that will be used throughout the code (Figure 4.9). At the time of definition, they are empty objects to store the point clouds.

```
// Objects for storing the point clouds.
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud1(new pcl::PointCloud<pcl::PointXYZ>);
pcl::PointCloud<pcl::PointXYZ>::Ptr plane(new pcl::PointCloud<pcl::PointXYZ>);
pcl::PointCloud<pcl::PointXYZ>::Ptr concaveHull(new pcl::PointCloud<pcl::PointXYZ>);
pcl::PointCloud<pcl::PointXYZ>::Ptr translated(new pcl::PointCloud<pcl::PointXYZ>);
```

Figure 4.9. Definition of objects for storing point clouds.

The type of file obtained when storing a point cloud with Realsense-viewer is PLY, however, to be able to work in PCL it needs to be modified to PCD format. For this, the code showed in Figure 4.10 is used. In this example, the point cloud obtained from the flat, smooth wall is stored in one of the previously defined clouds (*cloud*).

```
// Read a PLY file and convert it into a PCD file
pcl::PCLPointCloud2 clod;
pcl::PLYReader reader;
reader.read("wall6.ply", clod);
pcl::PCDWriter writer;
writer.writeASCII("wall6.pcd", clod);
pcl::io::loadPCDFile("wall6.pcd", *cloud);
```

Figure 4.10. Conversion of a PLY file into a PCD file and store PCD file in a point cloud.

At this stage, a plane of the obtained point cloud needs to be defined. To do this, it will be necessary to use an object recognition technique. In section 3.2.5, it was decided to use RANSAC to perform this recognition.

The PCL library has algorithms to obtain a planar model; based on the RANSAC algorithm.The *pcl:SACSegmentation <pcl::PointXYZ>* object is created and the model (*pcl::SAC_RANSAC*) and method type (*pcl::SACMODEL_PLANE*) are set. This is also where the "distance threshold" is specified, which determines how close a point must be to the model to be considered an inlier.

The indexes to the inliers are defined and looked for by the algorithm. The new points (inliers) are copied to a new cloud (*plane*). Figure 4.11 shows the implementation of the RANSAC algorithm using the resources available in the PCL library.

```
// Get the plane model
pcl::ModelCoefficients::Ptr coefficients(new pcl::ModelCoefficients);
pcl::SACSegmentation<pcl::PointXYZ> segmentation;
segmentation.setInputCloud(cloud);
segmentation.setModelType(pcl::SACMODEL_PLANE);
segmentation.setMethodType(pcl::SAC_RANSAC);
segmentation.setDistanceThreshold(0.01);
segmentation.setOptimizeCoefficients(true);
pcl::PointIndices::Ptr inlierIndices(new pcl::PointIndices);
segmentation.segment(*inlierIndices, *coefficients);

if (inlierIndices->indices.size() == 0)
  std::cout << "Could not find a plane in the scene." << std::endl;
else
{
  // Copy the points of the plane to a new cloud.
  pcl::ExtractIndices<pcl::PointXYZ> extract;
  extract.setInputCloud(cloud);
  extract.setIndices(inlierIndices);
  extract.filter(*plane);
```

Figure 4.11. RANSAC algorithm for plane fitting in the obtained point cloud.

Figures 4.12 and 4.13 visualise images of the plane obtained from the point cloud provided. The coordinate system shown in the figure is that of the camera. Where z is the blue axis (depth), y is the green axis (height), and x is the red axis (width).

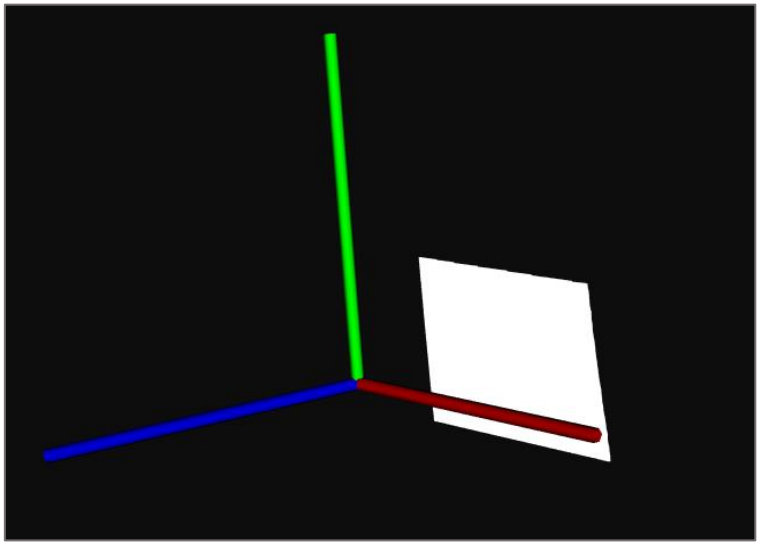

Figure 4.12. Visualisation of the plane obtained in the point cloud after using the RANSAC algorithm from a frontal perspective.

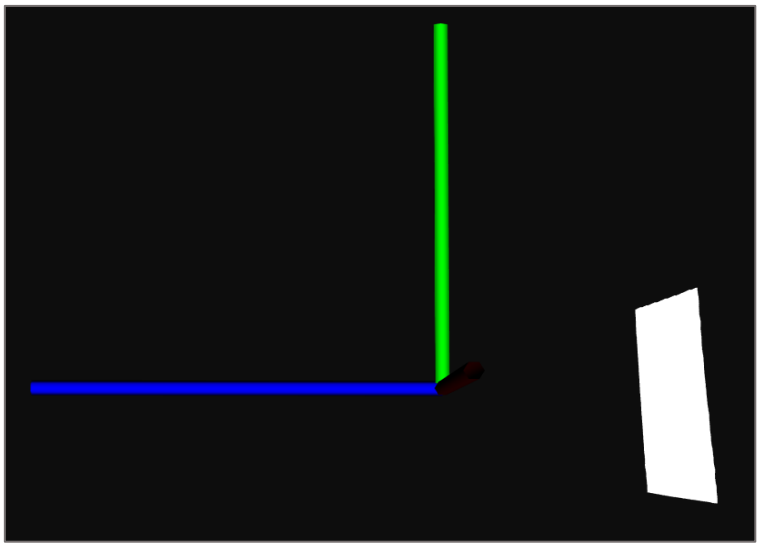

Figure 4.13.Visualisation of the plane obtained in the point cloud after using the RANSAC algorithm from a lateral perspective.

4.3. Determining points for robot's trajectory

The next objective of the thesis is the definition of a series of points that are separated a fixed distance from the previously defined plane. These points will serve to define the scanning trajectory that the robot's end effector must follow.

4.3.1. Code Description

The first thing to be done is displacing the previously defined plane. This step is necessary because, if the path is planned with the points from the original cloud, the robot arm would collide against the wall when performing the scan. The plane will be moved forward to select the scan points in this new plane.

To do this step it is necessary to perform a translation operation on the plane which has been defined through the point cloud. According to the camera coordinate system, given that z is the depth, the plane will be moved forward a defined distance in the z axis. Since the image has been taken at approximately 55 cm from the wall, it will be moved forward by 30 cm, so the robot arm trajectory would be in a plane located 25 cm from the camera. This measurement is now set randomly and should be carefully defined once the image is taken from the robot.

To perform this step, a translation matrix is defined, and the transformation executed by storing the data in a new point cloud (*translated*), as can be seen in Figure 4.14.

```
//Move foward the point cloud so the robot doesn't collide with the wall
Eigen::Affine3f transform = translation_matrix (0.0,0.0,0.3);
// Executing the transformation
pcl::transformPointCloud (*plane, *translated, transform);
```

Figure 4.14. Translation of the plane a certain distance from the wall and storage of the new set of points in a different point cloud.

Figure 4.15 shows the transferred cloud (red plane).

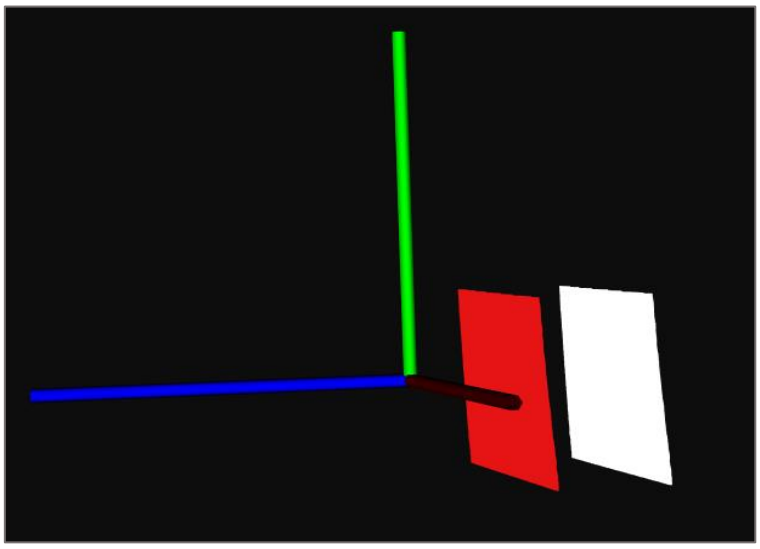

Figure 4.15. Initial plane (white) and transformed plane (red), which has been shifted forward using a matrix transformation are displayed in the camera frame.

The next step is the extraction of a series of points in this new plane through which the robot would be able to perform a scanning trajectory, and thus determine the radiological contamination.

After analysing different algorithms among those studied in chapter 3, it has been decided to use the concave hull shape analysis technique, explained in section 3.2.6 which will extract the contour of points in the translated plane. As a first option, downsampling (section 3.2.3.1) was considered, however the position of the points obtained was random which made it complicated to define an organised trajectory through them.

The object for retrieving the concave hull (*hull*) is defined, then it is applied to the input cloud (*translated*), the resolution for the hull is set and these new points are saved inside a new cloud (*concaveHull*), as can be seen in Figure 4.16.

```
// Object for retrieving the concave hull.
pcl::ConcaveHull<pcl::PointXYZ> hull;
hull.setInputCloud(translated);
// Set alpha, which is the maximum length from a vertex to the center of the voronoi cell
// (the smaller, the greater the resolution of the hull).
hull.setAlpha(0.1);
hull.reconstruct(*concaveHull);
```

Figure 4.16. Performing the Concave Hull algorithm to obtain the contour of the plane.

Figures 4.17 and 4.18 show the result of the extracted contour.

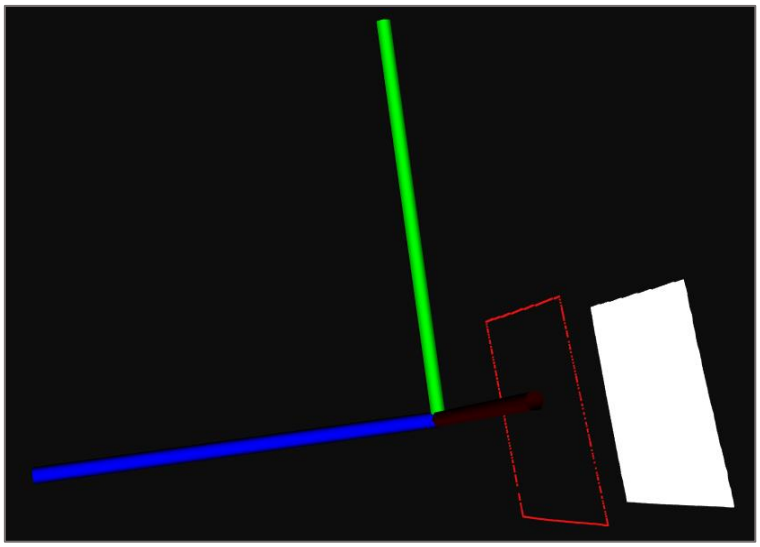

Figure 4.17.Lateral visualisation of the contour obtained using the Concave Hull algorithm.

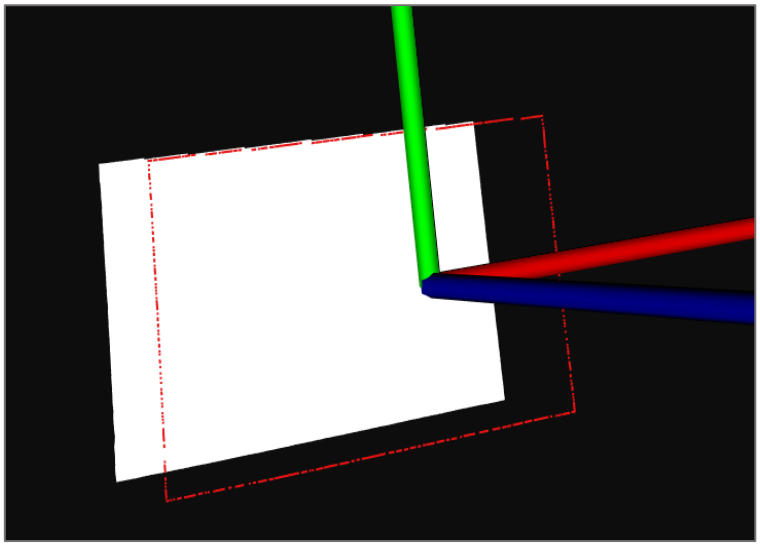

Figure 4.18.Frontal visualisation of the contour obtained using the Concave Hull algorithm.

4.3.2. Trajectory point extraction

The objective is to define a series of points within the obtained point cloud so that the robot can trace a path through them and perform the scanning of the wall. As indicated above, the initial plane is moved a certain distance from the wall so that the robot does not collide when performing its path.

So far, the contour of the point cloud, has been obtained through the Concave Hull algorithm. A set of points from this contour will be further extracted to define the trajectory. Since it is a very simplified model, flat and rectangular surface, whose z is constant (depth) and only the x and y coordinates vary.

To obtain a set of points, the following algorithm has been proposed. It should be noted that this is an approximate algorithm since no great precision is required when selecting

the points. The main idea is that the scanning process should cover as much of the surface as possible.

1. Obtaining the point with largest and smallest y-coordinate in absolute value. These would be the ones marked in Figure 4.19 (represented on a large scale), blue (largest value) and orange (smallest value). The reason for using the absolute value is the following: it avoids always selecting positive numbers as the largest values and negative numbers as the smallest values. However, it should be noted that it is also possible to define the largest positive value and the smallest negative value and to operate with them, always considering the signs.
   The minimum value, given that the operation is performed with the contour of a point cloud, is not exactly 0, but very close to it. Therefore, the algorithm could be defined assuming this value as 0.

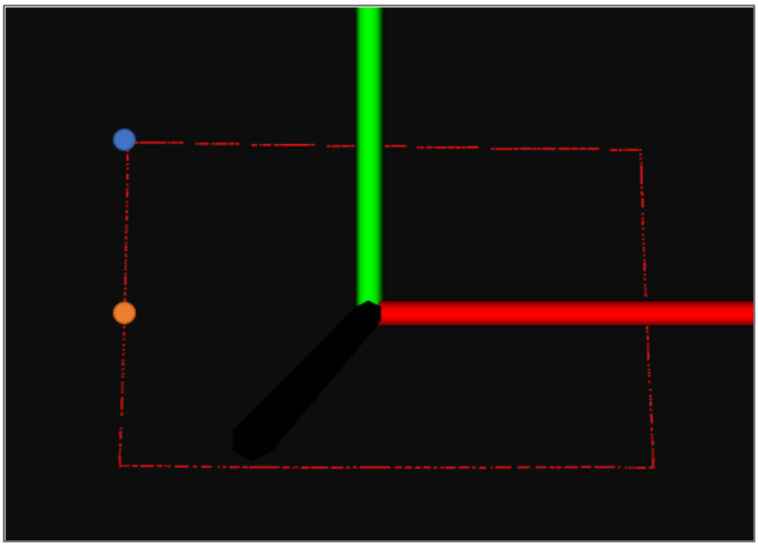

Figure 4.19. Points selected out of all existing points on the contour, being blue point (largest value) and orange point (shortest value) y-coordinates in absolute value.

2. Subtraction of these values to obtain the height of the plane (they are multiplied by two to obtain the total height, since otherwise only half would be obtained).
3. Since the x-value will be the same for all the selected points (sometimes negative and sometimes positive), it is necessary to extract the one with the highest value in the point cloud. In this way, since it is an approximate and not an exact algorithm, it is ensured that the largest possible area is scanned.
4. Define a radius of coverage of the measuring probe.
5. Divide the total width by the radius of the probe, so that several segments are defined.

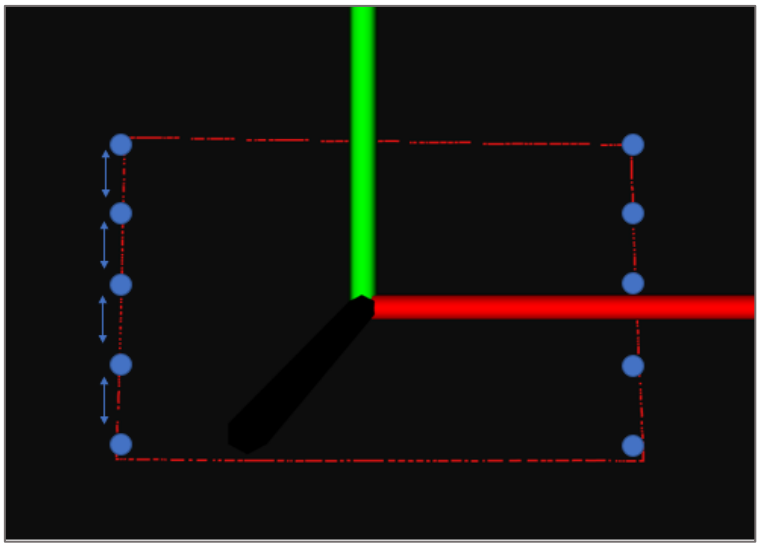

Figure 4.20. Set of contour points defined after performing the selection algorithm.

6. Fill a new point cloud with the following parameters: the cloud width (calculated in point 5), the largest x and y, the fixed z and the radius. For this purpose, the use of the positive and negative value of the previously defined x-coordinate has been alternated. In this way, a trajectory of points is obtained from side to side, as shown in Figure 4.20. If only one value of x were used, for example, the positive one, only the points on the right-hand side could be obtained.

Figure 4.20 shows the number of points that have been obtained so that the robot can do the scanning. It should be noted that this image, and Figure 4.19, shows roughly how the algorithm is executed, but it does not work with real measurements nor points.

4.3.2.1. Code Description

To execute this part of the code in a clearer way, it has been structured in functions, in charge of calculating the respective maximum and minimum values of the coordinates. Since all the functions are very similar, only the one that obtains the maximum value of y will be explained.

```
pcl::PointXYZ largest_y_value (pcl::PointCloud<pcl::PointXYZ>::Ptr cloud)
{
    pcl::PointXYZ first = cloud->points[0];
    for(int i = 0; i < cloud->width; i++)
    {

    pcl::PointXYZ v = cloud->points[i];
        // Find the largest y
        if(abs(first._PointXYZ::data[1]) < abs(v._PointXYZ::data[1]))
            first._PointXYZ::data[1] = v._PointXYZ::data[1];
    }

  return first;
}
```

Figure 4.21. Function to calculate the largest value of the y-coordinate given a point cloud.

Through an iterative loop, all the y-coordinate elements of the point cloud are compared in pairs. In a variable (*first*) is stored the greater of the two points compared, and finally, the largest (in absolute value) of the cloud will be obtained. This part of the code can be seen in Figure 4.21.

The function has the point cloud to be analysed as input, i.e. the point scan contour separated a fixed distance from the wall (*concaveHull*) and returns as output a point XYZ, which fulfils as characteristic that it is the one with the largest Y- coordinate of the whole cloud.

Figures 4.22, 4.23 and 4.24 are used to define the desired values. It is also necessary to set a depth value for the points, i.e. the z-coordinate. Since z is considered to be a constant value, as the distance from the camera to the scan contour remains fixed, the distance provided by one of the three points obtained will be used (they should all have the same z-coordinate).

```
//Set maximum value in the y axes

pcl::PointXYZ max_y = largest_y_value(concaveHull);

float l_x=max_y._PointXYZ::data[ 0 ];
float largest_y=max_y._PointXYZ::data[ 1 ];
float l_z=max_y._PointXYZ::data[ 2 ];
```

Figure 4.22. Obtaining the point with the highest y-coordinate of the contour.

```
//Set minimum value in the y axes

pcl::PointXYZ min_y = shortest_y_value(concaveHull);

float s_x=min_y._PointXYZ::data[ 0 ];
float shortest_y=min_y._PointXYZ::data[ 1 ];
float s_z=min_y._PointXYZ::data[ 2 ];
```

Figure 4.23. Obtaining the point with the shortest y-coordinate of the contour.

```
//Set maximum value in the x axes

pcl::PointXYZ max_x = largest_x_value(concaveHull);

float largest_x=max_x._PointXYZ::data[ 0 ];
```

Figure 4.24. Obtaining the point with the largest x-coordinate of the contour.

A scanning radius of the probe is defined, in this case 9 cm has been marked as a random value, and with this value the number of segments into which the height of the cloud is divided is calculated (*num_segments*). With this last value, the size of the new point cloud

(*width_cloud*) in which the selected points will be stored is defined. Figure 4.25 shows the code programmed to obtain these values.

A function has been defined to fill the new point cloud with the defined points.

```
//Define the radius of coverage of the probe (random value 9cm) and state the number of points

float radius=0.09;
int num_segments=(abs(largest_y)-abs(shortest_y))/radius;
cout << "Number of segments = " << num_segments<< endl;
int width_cloud = (num_segments*2-2)*2;
```

Figure 4.25. Definition of the number of segments into which the height of the plane is to be divided.

```
Longest element y = (-0.382071,-0.288829,-0.242688)
Shortest element y = (-0.382071,-0.000186767,-0.242688)
Longest element x = -0.402847
Number of segments = 3
Saved 8 data points to procces
    -0.402847 0.288829 -0.242688
    0.402847 0.288829 -0.242688
    -0.402847 0.108829 -0.242688
    0.402847 0.108829 -0.242688
    -0.402847 -0.0711713 -0.242688
    0.402847 -0.0711713 -0.242688
    -0.402847 -0.251171 -0.242688
    0.402847 -0.251171 -0.242688
```

Figure 4.26. Displayed result with the highest and lowest value of the y and x coordinate of the contour and the set of points to be scanned.

Figure 4.26 shows the points that have been selected in the point cloud out of the contour. These new points will be stored in a new point cloud, so that it now moves to working exclusively with them.

4.4. Coordinate system transformation

The next step is to proceed with the transformation between the camera frame and the robot's end effector frame to be able to work with the robot.

4.4.1. Execution of calculations

The coordinates currently available are referenced to the camera frame, however, to be able to work with the robot arm it is necessary to reference them to the end effector's coordinate system. For more complex systems, a hand-eye calibration would be required. However, since the dimensions of the mount connecting the robot to the camera are known, simple mathematical operations can be performed to proceed with the point conversion.

The first thing to consider is the location of the camera coordinate system. For this purpose, its datasheet will be analysed.

The depth start point or the ground zero reference can be described as the starting point or plane where depth = 0 (z=0). For the L515 camera, this point is referenced from the front of the camera cover glass, as can be seen in Figures 4.27 and 4.28.

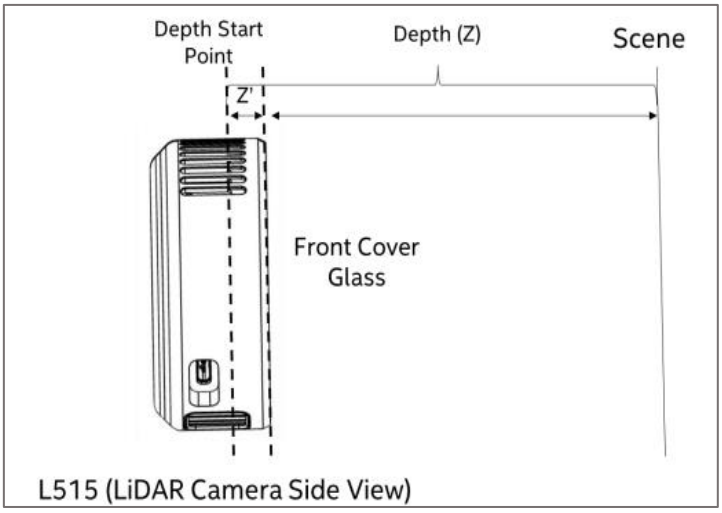

Figure 4.27. Depth Start Point location from the front cover glass in the Realsense camera L515 [29].

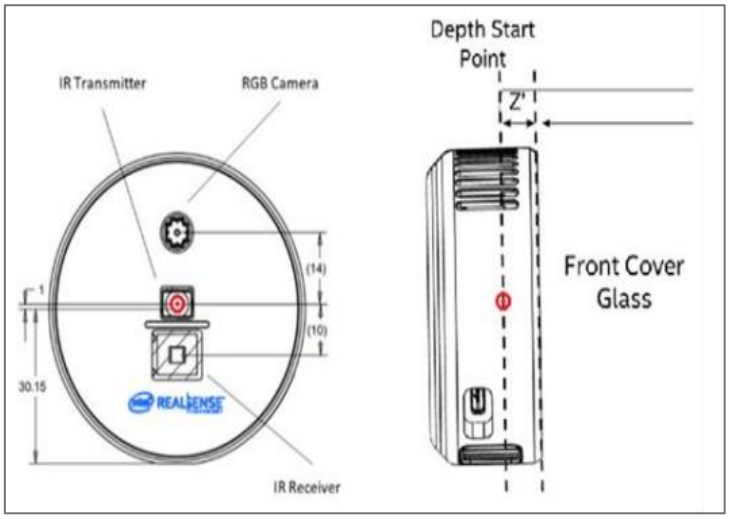

Figure 4.28. Camera frame origin location (left picture) in the front cover glass of the Realsense camera L515 [29].

As indicated in the datasheet, the depth start point is located at Z'=4.5 mm, with respect to the front cover glass. In order to know the distance between the coordinate's origin (depth start point) and the base of the camera (opposite to the cover glass), it is necessary to know the total width of the camera, a measurement that can also be obtained from the datasheet.

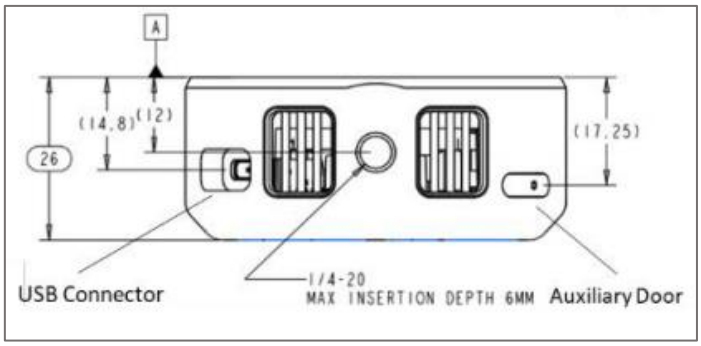

Figure 4.29. Width of the Intel Realsense camera L515.

As can be seen in the Figure 4.29, the total width of the camera is 26 mm. Therefore, the distance we are looking for is as calculated in equation 4.1.

$$Z'' = 2.6 - 0.45 = 2.15\ cm \quad (4.1)$$

We now proceed to analyse the CAD model that has been mounted on the robot arm. After measuring its length using the graphics program Revit from AutoDesk, it has been determined to be 9.6 cm, as shown in Figure 4.30. Therefore, the distance between the camera frame origin and the end effector (f) is 11.75 cm, calculated in equation 4.2.

$$f = l + Z'' = 9.6 + 2.15 = 11.75\ cm \quad (4.2)$$

This can be clearly seen in the sketch shown in Figure 4.32.

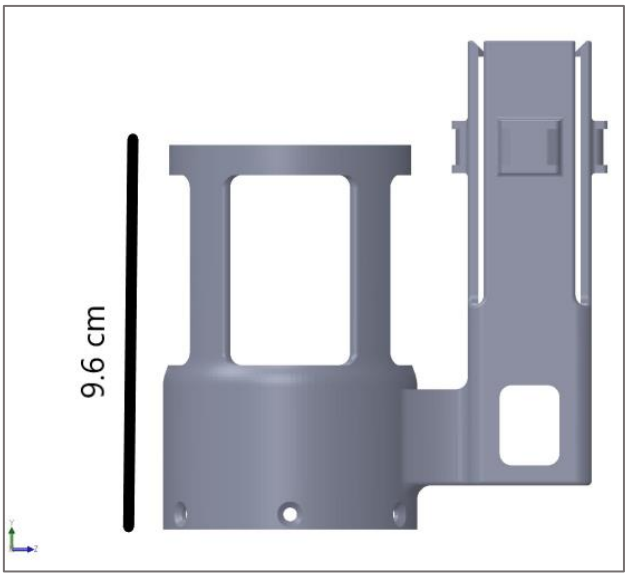

Figure 4.30. Measurement of the length of the LiDAR camera holder.

So far, the distance between the origin of coordinates of the end effector and the camera (f=11.75 cm) is known. However, besides translation, rotation of the camera frame to the end effector frame is also necessary.

To determine the orientation of the end effector, rviz has been used. Using TF, the position and orientation of the end effector coordinate system has been obtained, as shown in Figure 4.31. Both rviz and TF have been explained in section 2.1.1.2.

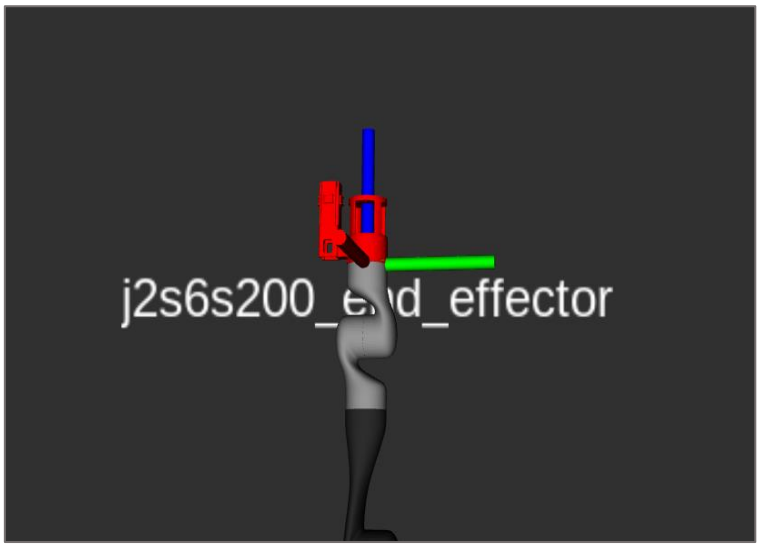

Figure 4.31. Visualisation of the end effector frame in the Kinova arm using RViz.

The orientation of the camera coordinate system could be determined from the previous processing of the point cloud.

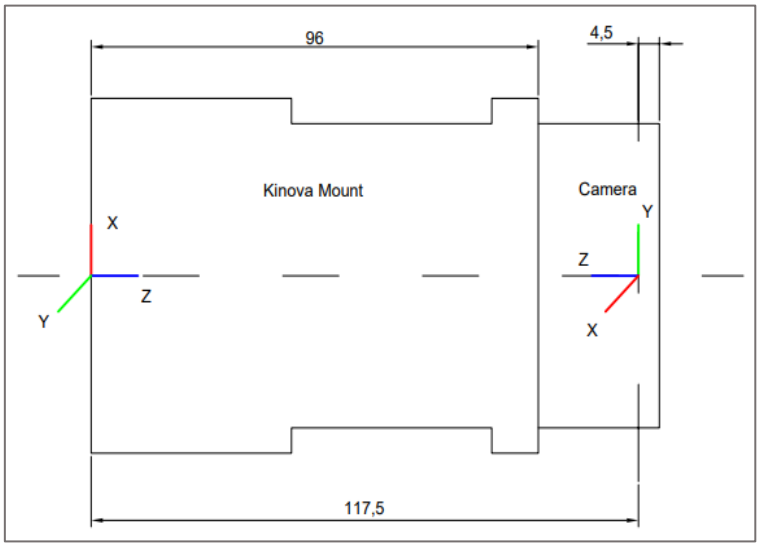

Figure 4.32. Schematic representation of the 3D printed mount together with the camera, showing the orientation and distance (in mm) of the coordinate systems of the end effector and the camera.

In the image above a side view of the Kinova mount and the camera, fitted together as they would be on the actual robot arm can be seen. The two frames are rotated in such way that their z-axis point to each other. As calculated before, the distance between both frames is 11.75 cm. The camera coordinate system must now be translated and rotated to be aligned with the end effector frame.

The steps to be followed to make the two systems coincide are as follows:

- Translation in the Z-axis (camera frame) of 11.75 cm, so that both coordinate origins are coinciding. Figure 4.33 shows the result of the translation.

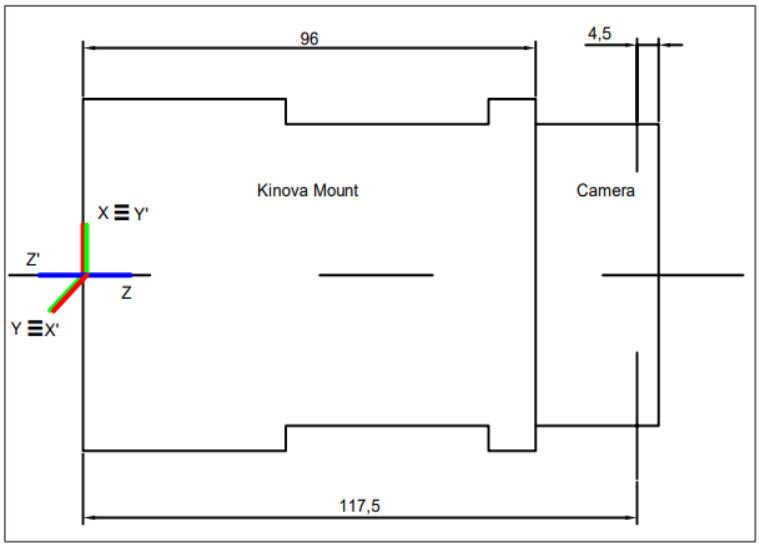

Figure 4.33. Camera and end effector frame after the camera frame being translated 11.75 cm along the Z camera axis.

- Rotation of 180º around the Y axis of the camera, this will make both Z axes coincide. Figure 4.34 represents both systems after doing the rotation, now the X and Y axes need to come together.

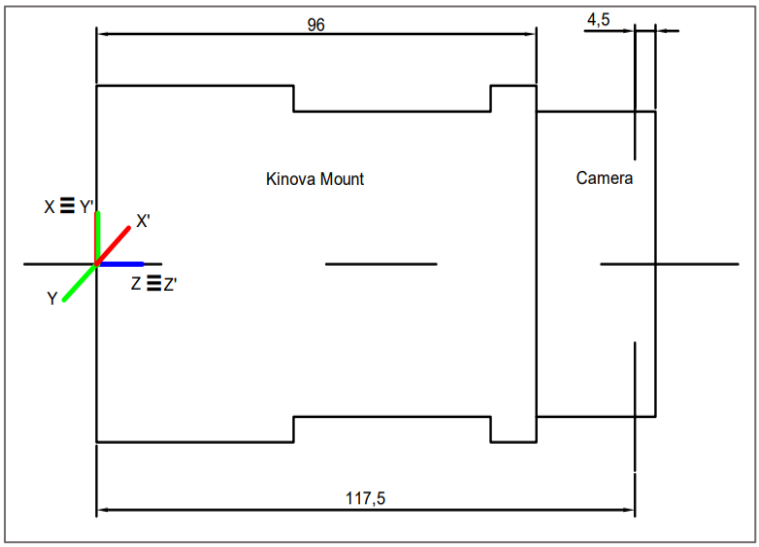

Figure 4.34. Camera and end effector frame after the camera frame being rotated 180º along the Y camera axis.

- In the latter coordinate system, after having rotated it 180º, a 90º rotation around the new Z axis is performed. Now both systems coincide, as depicted in Figure 4.35.

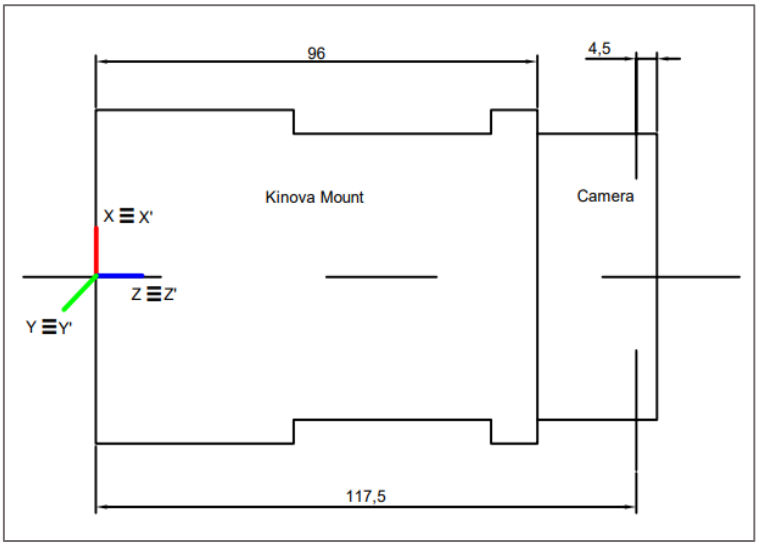

Figure 4.35. Camera and end effector frame after the camera frame being rotated 90º along the new Z camera axis.

4.4.2. Code Description

The attached code in Figure 4.36 shows how a translation matrix of 11.75 cm is defined along the Z axis, then this matrix is rotated by an angle theta of 180 degrees (π radians) and finally rotated by 90 degrees (π/2 radians) around the new Z axis.

This transformation is applied using the *pcl::transformPointCloud* class, which stores the transformed points in the new cloud (*cloud1)*.

```
//Change coordinate frame

// Translation 11.75 cm along Z axis
Eigen::Affine3f transform1 = translation_matrix (0.0,0.0,0.1175);
//Rotation 180 degrees along Y axis
float theta = M_PI; // The angle of rotation in radians (180 degrees)
transform1.rotate (Eigen::AngleAxisf (theta, Eigen::Vector3f::UnitY()));
//Rotation 90 degrees along Z axis
float theta2 = M_PI/2; // The angle of rotation in radians (90 degrees)
transform1.rotate (Eigen::AngleAxisf (theta2, Eigen::Vector3f::UnitZ()));

pcl::transformPointCloud (*newpc, *cloud1, transform1);
```

Figure 4.36. Change of the reference coordinate system from camera to end effector.

Now we have the points through which the robot will define its trajectory referred to the end effector frame. To be able to work with them in MoveIt!, which will be used to program the robot's path, it is necessary to store those points in a text file (.txt).

This will be done by storing the points in a matrix, with a number of rows equal to the number of points and a number of columns equal to 3, one for each XYZ coordinate. Subsequently, these points will be stored in a text file.

4.5. Robot motion planning

The last step of this thesis is to try to send the robot to the points that have been determined. For this, we will work with MoveIt!, which has been explained in section 2.1.2.

Initially, it is necessary to integrate our new robot with MoveIt!. To do this MoveIt SetUp Assistant will be used, so the robot's MoveIt! package can be created.

The MoveIt! Setup Assistant is a graphical user interface developed for configuring any robot to work with MoveIt!. Its main purpose is to generate a Semantic Robot Description Format (SRDF) file for the robot. It also generates another series of files needed to work with the MoveIt! pipeline [53].

Once the package is created, work with the the MoveIt! rviz Plugin can now begin. To do this, it is necessary to launch the following command:

roslaunch kinova_archer_v2_moveit_config demo.launch

This will launch rviz with the motion planning option available, as shown in Figure 4.37.

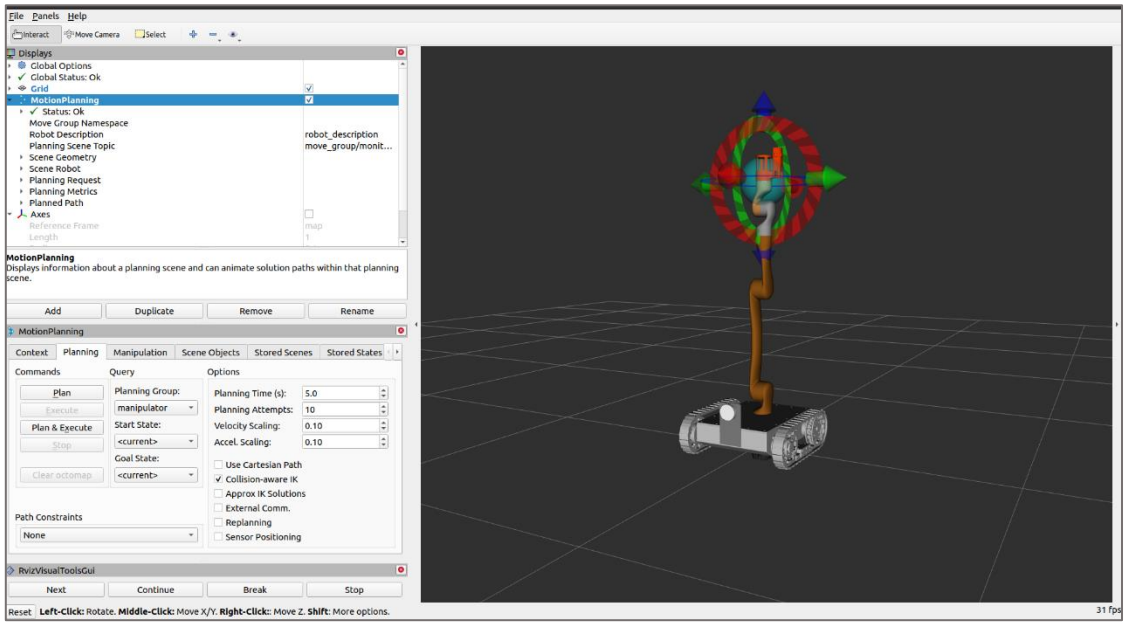

Figure 4.37. MoveIt! rviz pluging showing the ARCHER robot.

Given that the time available for the thesis has been limited, despite having worked on planning trajectories with the robot using the MoveIt! interface, it has not been possible to describe the trajectory through the points obtained in the cloud. This will be proposed as future work.

4.5.1. Move Group Interface

The MoveGroup class is the simplest user interface in MoveIt! It comes with easy to use functionalities for most operations, such as setting joint or pose goals, creating motion

plans, moving the robot, adding objects to the environment and attaching/detaching objects from the robot. As explained in section 2.1.2, this interface communicates over ROS topics, actions and services to the MoveGroup Node.

An example of the use of the Move Group C++ Interface, which will be implemented on a simulation of the robot arm, will be carried out. In this file, a Cartesian path is programmed, specifying a list of waypoints for the end-effector to go through.

To program the code, the help provided by the MoveIt! tutorials has been followed. The purpose of this is to have a clear idea of how the robot is supposed to move once the points have been obtained. Since a simulation was used , it has not been possible to rotate the joint constituted by the 3D mount, but it is to be made clear that the trajectory displayed should have the 3D mount facing the screen, as if the scan was actually being performed.

To run the trajectory in simulation it is necessary to launch the following commands:

roslaunch kinova_archer_v2_moveit_config demo.launch

roslaunch moveit_tutorials move_group_interface_tutorial.launch

In the digital version of the document, you can visualise the trajectory that has been programmed by clicking on the following image (Figure 4.38).

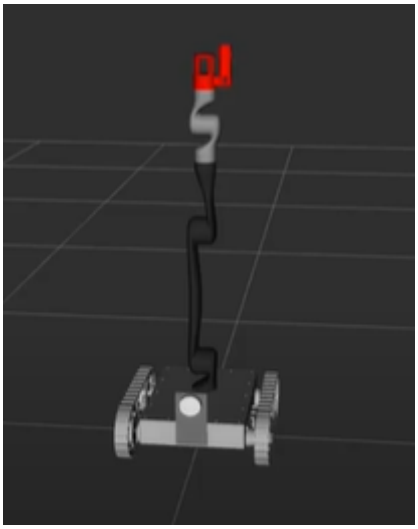

Figure 4.38. Image with associated link to access the video of the programmed trajectory.

5. CONCLUSION

5.1. Conclusions

In this bachelor thesis, the analysis of the structure of the Kinova robot arm was proposed, as well as a case study on point cloud processing to determine a path for the robot arm to follow.

Starting with the robot arm analysis, it has been found to have a ROS interface for robot control that communicates with the DSP inside the base of the robot. It is simple and intuitive to use and allows the robot to be controlled in cartesian and joint space in a variety of ways.

On the other hand, using a camera that makes use of LiDAR technology, a point cloud of a wall has been obtained. Using PCL and its multiple algorithms, it has been possible to extract a set of points which can be used as a path for the robot arm to follow. PCL was selected after an analysis of the different techniques that exist for processing point clouds has been carried out.

Finally, the ROS MoveIt! application was used to simulate the programming of a trajectory similar to the one that the robot would take when scanning the wall. However, given the limited time available, it has not been possible to describe this trajectory with the points obtained after cloud processing.

5.2. Possible future work

Within the framework of this thesis, there are a few points that could be proposed for future work.

The first step can be the modification of the robot arm by connecting both the camera and the probe to the 3D mount. As indicated above, given that time was limited, it was not possible to proceed with the actual modification of the arm, so the image obtained with the camera was taken from a computer. To be able to work with real measurements, it would be necessary to physically connect the camera to the robot arm.

Another objective that can be proposed for future work is to send the robot to the points defined after processing the point cloud. As indicated in section 4.5.1, the trajectory planned in MoveIt! is made up of a set of points that have been predefined, but they are not real points taken by the camera. To do this, it is necessary to have previously connected the camera to the robot.

Finally, a future work is the scanning of the surface to be processed using the Kromek probe. The contamination values obtained could be stored and saved in a file for further treatment.

BIBLIOGRAPHY

[1] "ACRO - Research Group," [Online]. Available: https://iiw.kuleuven.be/onderzoek/acro. [Accessed 24 May 2021].

[2] R. Vanonckelen, "Decoupled path planning for mobile manipulators: exploratory research via simulation of a robot scanning a surface," 2020.

[3] "ROS," [Online]. Available: https://www.ros.org/. [Accessed 22 April 2021].

[4] E. Robotics, "The Robot Operative System (ROS): Powering the world's robots," [Online]. [Accessed 22 April 2021].

[5] B. G. W. D. S. M. Quigley, Programming Robots with ROS, O'Reilly Media, 2015.

[6] "What is ROS?," 2018-08-08, 8 August 2018. [Online]. [Accessed 22 April 2021].

[7] E. F. y. L. S. C. A. M. Romero, Learning ROS for Robotics Programming (Second Edition), 2015.

[8] "ROS Concepts," 2014-06-21, 21 June 2014. [Online]. Available: http://wiki.ros.org/ROS/Concepts. [Accessed 22 April 2021].

[9] F. J. B. Durán, "Programación de robot móvil con manipulador para el sector del comercio en entorno ROS," Seville, 2017.

[10] "AWS RoboMaker," [Online]. Available: https://docs.aws.amazon.com/robomaker/latest/dg/what-is-robomaker.html. [Accessed 22 April 2021].

[11] I. Sucan, "ROS URDF," 11 January 2019. [Online]. Available: http://wiki.ros.org/urdf. [Accessed 22 April 2021].

[12] "ROS urdf link," 1 May 2021. [Online]. Available: http://wiki.ros.org/urdf/XML/link. [Accessed 3 May 2021].

[13] "ROS urdf joint," 5 November 2018. [Online]. Available: http://wiki.ros.org/urdf/XML/joint. [Accessed 3 May 2021].

[14] E. M.-E. W. M. Tully Foote, "ROS tf," 2 October 2017. [Online]. Available: http://wiki.ros.org/tf. [Accessed 22 April 2021].

[15] “MoveIt! ROS,” [Online]. Available: https://moveit.ros.org/. [Accessed 26 April 2021].

[16] “MoveIt Concepts,” [Online]. [Accessed 26 April 2021].

[17] “ROS Topics,” 20 February 2019. [Online]. Available: http://wiki.ros.org/Topics. [Accessed 26 April 2021].

[18] “ROS Services,” 18 July 2019. [Online]. Available: http://wiki.ros.org/Services. [Accessed 26 April 2021].

[19] V. P. M. A. Eitan Marder-Eppstein, “ROS actionlib,” 30 October 2018. [Online]. Available: http://wiki.ros.org/actionlib. [Accessed 2021 April 2021].

[20] B. G. &. W. D. S. Morgan Quigley, Programming robots with ROS., O’Reilly Media, 2013.

[21] “Parameter Server,” 8 November 2018. [Online]. Available: http://wiki.ros.org/Parameter%20Server. [Accessed 26 April 2021].

[22] M. Tutorials, “https://ros-planning.github.io/moveit_tutorials/,” [Online]. [Accessed 14 June 2021].

[23] KINOVA, “User Guide. KINOVA Gen2 Ultra lightweight robot.,” 2019.

[24] “Kinova-ROS,” [Online]. Available: https://github.com/Kinovarobotics/kinova-ros. [Accessed 2 May 2021].

[25] “Joint Position Controller,” 2011-04-19, 19 April 2011. [Online]. Available: http://wiki.ros.org/robot_mechanism_controllers/JointPositionController#:~:text=The%20Joint%20Position%20Controller%20commands,command%20to%20the%20current%20position.. [Accessed 26 April 2021].

[26] J. C. Fraile, “Cinemática Inversa de un Robot Manipulador,” 2020.

[27] Wikipedia, “LiDAR,” 15 May 2021. [Online]. Available: https://en.wikipedia.org/wiki/Lidar. [Accessed 25 May 2021].

[28] “What Is Solid State LiDAR and Is It Faster, Cheaper, Better?,” 18 June 2018. [Online]. Available: https://www.allaboutcircuits.com/news/solid-state-lidar-faster-cheaper-better/. [Accessed 10 June 2021].

[29] Intel, “LiDAR Camera L515 Datasheet,” 2021.

[30] “What are point clouds?,” [Online]. Available: https://www.dronegenuity.com/point-clouds/. [Accessed 27 May 2021].

[31] “What is point cloud modelling?,” [Online]. Available: https://www.takeoffpros.com/2020/07/14/what-is-point-cloud-modeling/. [Accessed 27 May 2021].

[32] Y. T. Z. M. Qian Wang, “Computational Methods of Acquisition and Processing of 3D Point Cloud Data for Construction Applications”.

[33] “La nube de puntos ¿Qué es?,” [Online]. Available: https://bit.ly/3cG8GYm . [Accessed 2 June 2021].

[34] A. S. J. C. K. S. M. S. R. J.C. Fernandez, “An Overview of Lidar Point Cloud Processing Software,” Civil and Coastal Engineering Department, University of Florida, 2007.

[35] R. A. Cano, “TÉCNICAS Y HERRAMIENTAS DE PROCESAMIENTO DE NUBES DE PUNTOS TRIDIMENSIONALES,” Madrid, 2015.

[36] “PCL/OpenNI tutorial 3: Cloud processing,” 1 November 2015. [Online]. Available: https://robotica.unileon.es/index.php?title=PCL/OpenNI_tutorial_3:_Cloud_processing_(advanced)#Retrieving_the_hull. [Accessed 18 May 2021].

[37] “Region Growing,” 14 January 2021. [Online]. Available: https://en.wikipedia.org/wiki/Region_growing. [Accessed 27 May 2021].

[38] J. Gomez, «Segmentacion basada en regiones,» Universidad de Valladolid.

[39] A. Golovinskiy y T. Funkhouser, «Min-Cut Based Segmentation of Point Clouds.,» Princeton University.

[40] “PCL/OpenNI tutorial 2: Cloud processing (basic),” 18 November 2015. [Online]. Available: https://robotica.unileon.es/index.php?title=PCL/OpenNI_tutorial_2:_Cloud_processing_(basic)#Voxel_grid. [Accessed 28 May 2021].

[41] “Removing outliers using a StatisticalOutlierRemoval filter,” [Online]. Available: https://pcl.readthedocs.io/projects/tutorials/en/latest/statistical_outlier.html. [Accessed 14 June 2021].

[42] “Hough Transform,” 19 March 2020. [Online]. Available: https://es.wikipedia.org/wiki/Transformada_de_Hough. [Accessed 27 May 2021].

[43] “Transformada clásica de Hough,” [Online]. Available: https://programmerclick.com/article/37371278385/. [Accessed 27 May 2021].

[44] J. Gómez, “Reconocimiento de objetos,” Universidad de Valladolid.

[45] “RANSAC,” [Online]. Available: https://kripkit.com/ransac/. [Accessed 18 May 2021].

[46] “Fitting and Matching,” [Online]. Available: https://n9.cl/opsauf. [Accessed 11 June 2021].

[47] Wikipedia, “Convexividad,” 13 May 2021. [Online]. Available: https://es.wikipedia.org/wiki/Convexidad. [Accessed 6 June 2021].

[48] “PCL Documentation,” [Online]. Available: https://pointclouds.org/about/. [Accessed 17 May 2021].

[49] J. P. V. K. Qian-Yi Zhou, “Open3D: A Modern Library for 3D Data Processing,” 2018.

[50] “What is PDAL?,” 12 April 2021. [Online]. Available: https://pdal.io/about.html#. [Accessed 28 May 2021].

[51] “Intel Realsense Depth Camera,” [Online]. Available: https://www.intelrealsense.com/get-started-depth-camera/. [Accessed 17 May 2021].

[52] “Capturing 3D Point Cloud with Intel Realsense,” [Online]. Available: https://www.andreasjakl.com/capturing-3d-point-cloud-intel-realsense-converting-mesh-meshlab/. [Accessed 17 May 2021].

[53] “MoveIt SetUp Assistant,” [Online]. Available: https://ros-planning.github.io/moveit_tutorials/doc/setup_assistant/setup_assistant_tutorial.html. [Accessed 24 May 2021].