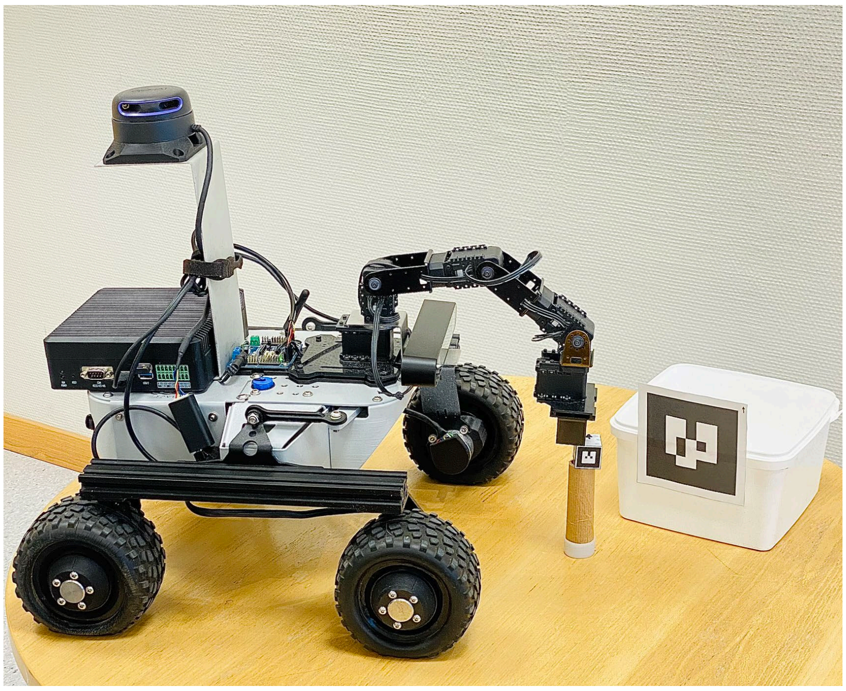
University of
South-Eastern Norway

www.usn.no

FMH606 Master Thesis 2024
Industrial IT and Automation

Computer vision-guided autonomous grasping system using Leo Rover with robotic arm.



Nasir Ali

Faculty of Technology, Natural Sciences and Maritime Sciences
Campus Porsgrunn

University of South-Eastern Norway

www.usn.no

Course: FMH606 Master's Thesis, 2024

Title: Computer vision-guided autonomous grasping system using Leo Rover with robotic arm

Number of pages: 104

Keywords: Leo Rover, Lidar sensor, ZED2 stereo camera, Pincher robotic arm, autonomous navigation using SLAM, real-time object detection, grasping system, simulation

Student: Nasir Ali

Supervisor: Ru Yan, Nils-Olav Skeie, Roshan Sharma

External partner: Applied Modeling and Control (AMOC) research group at USN.

Summary:

The master’s thesis project explores Leo Rover, equipped with a Lidar sensor, stereo camera, and robotic arm, available at University of South-Easten Norway. Leo Rover is a semi-autonomous mobile robot used for research and development purposes using the robot operating system (ROS).

The project aims to develop a computer vision-guided autonomous grasping system, where the rover detects nearby AR-tags (fiducial markers) based objects, determines the most suitable grasping approaches using its robotic arm, plan and execute Motion Planning (*MoveIt*) to pick-up and place small objects. The rover also performs autonomous navigation, including simultaneous localization and mapping (SLAM). Various methods are used including utilization of OpenCV tool for real-time object detection, configuring and integration of rover’s ROS network, testing multiple ROS packages, evaluating rover and its arm’s performance in both simulated and physical environment.

The thesis report provides valuable insights into the capabilities and limitations of the rover and its robotic arm. Consequently, a comprehensive risk assessment is carried out to identify potential safety concerns and ethical factors. This research serves as a foundational setup for future work for autonomous navigation and advanced grasping technologies in mobile robotics.

Preface

This master's thesis "Computer vision-guided autonomous grasping system using Leo rover and robotic arm", conducted by final year student in the course "Industrial IT and Automation (IIA)" at University of South-Easten Norway, Porsgrunn campus, during the spring term of 2024. It serves as a mandatory part of master's degree to fulfill the course requirements.

The target audience includes individuals with an interest or background in mobile robotics, sensor technologies, autonomous navigation and grasping system are beneficial for a comprehensive understanding of this master thesis. It has been challenging and time consuming while working with this thesis, however it offered an opportunity to learn more about robot operating systems. In this thesis, I utilized my knowledge and skills which was obtained throughout the courses "Control for robotic", "Software engineering" and "Introduction to autonomy" to achieve the project's tasks. With obtained results, I am satisfied to present my work to complete my journey of this thesis.

I would also like to express my gratitude to the research group (AMOC) and IT staff at USN, for providing all Leo rover and necessary hardware resources for completing success journey for master thesis. A special thanks to my supervisors, Ru Yan, Nils-Olav for continuous guidance and feedback. Their support and mentorship significantly enriched my knowledge and skills throughout the entire master thesis.

Overall, this journey reflects my commitment to achieve academic excellence and hands-on practical experience within industrial IT and automation.

Porsgrunn, 15.05.2023

Nasir Ali

Contents

Nomenclature

AC	–	Alternative Current
AD	–	Automated Driving
ADAS	–	Advanced Driver Assistance System
API	–	Application Programming Interface
ARS	–	Autonomous Robotic System
AVP	–	Automated Valet Parking
CNN	–	Convolution Neural Networks
CM	–	Centimeter
CSI	–	Camera Serial Interface
CPU	–	Centeral Procssing Unint
COCO	–	Microsoft Common Objects in Context
CUDA	–	Computer Unified Device Architecture
DC	–	Direct Current
DL	–	Deep learning
D-CNN	–	Deep Convolutional Neural Networks
DOF	–	Degree-of-freedom
EKF	–	Extended Kalman Filter
FK	–	Forward Kinematics
FDUCD	–	Fully Dressed User Case Diagram
FURPS+	–	Functionality Usability Reliability Performance Supportability
FTDI-USB	–	Future Technology Devices International – USB Cable
GIPO	–	General Purpose Input/Output
GPS	–	Global Positioning System
GPU	–	Graphincal Procssing Unit
HLF	–	High-Level Fusion
IK	–	Inverse Kinematics
IKFast	–	Inverse Kinematics Fast
ITRI	–	Industrial Technology Research Institute
IMU	–	Inertial Measurement Unit
IR	–	Infrared
JSON	–	JavaScript Object Notation
KITTI	–	Karlsruhe Institute of Technology and Toyota Technological Institute
LED	–	Light-emitting Diode
LeoOS	–	Leo Operating System
LLF	–	Low-level Fusion
Lidar	–	Light detection and ranging
mAh	–	Milliampere-hour

MSE –	Mean Square Error
MSDF –	Multi-Agent Service Function Chain Migration Framework
MPx –	Mega Pixel
ML –	Machine Learning
MLF –	Mid-level Fusion
MIT –	Massachusetts Institute of Technology
PCM –	Phase Change Material
PV-RCNN –	PointVoxel- Region-based Convolution Neural Network
PWR –	Power
ROS –	Robot Operating System
RPi –	Raspberry Pi
RGB –	Red Green and Blue
RGB-D –	RGB with Depth
Rviz –	ROS Visualization
SDK –	Software Development Kit
SAV –	Soft Actor-Critic
SSD –	System Sequence Diagram
SLAM –	Simultaneous Localization and Mapping
SMPS –	Switched Mode Power Supply
Sonar –	Sound Navigation and Ranging
SSD –	Single-Stage Object Detection
TCP –	Tool Center Point
TF –	Transform Tree
u.FL –	Ultra Miniature Coaxial Connector
UART –	Universal Asynchronous Receive/Transmitter
UI –	User Interface
UKF –	Unscented Kalman Filter
USB –	Universal Serial Bus
URDF –	Unified Robot Description Format
XML –	Extensible Markup Language
XML-RPC –	Extensible Markup Language - Remote Procedure Call
YAML –	Yet Another Markup Language
YOLO –	You Only Look Once

1 Introduction

This chapter provides an introduction related to the thesis, focusing on the development of computer vision-guided grasping system using Leo Rover and its robotic arm. It also includes several key elements. Firstly, it provides the background information regarding master's thesis and related research in mobile robotics. After that, it outlines objectives, including primary focus and goals of the achieved tasks. Finally, it discusses the employed methods, the scope of the project, contribution, software tools and an overview of the report structure.

1.1 Background

Nowadays, mobile robotics is one of the fastest-growing areas of scientific research. Mobile robotics is an industry focused on creating mobile robots that can move around in a physical environment. Mobile robots [1] are such kind of machines typically controlled by software that use various sensors and technologies to identify their surroundings. Due to their capabilities, these robots can replace humans in a variety of fields including exploration, remote handling of explosive materials, mining, construction, fire firefighting and rescue etc. It is apparent that mobile robots are becoming popular across a wide range of sectors because they are being used to assist in work processes to perform those tasks which are dangerous or impossible for human workers.

There are two main types of mobile robots such as autonomous and non-autonomous mobile robots. An autonomous mobile robot can explore its environments without using any external guidance, whereas a non-autonomous mobile robot can move around in its surrounding based on some type of instruction or guidance system [2]. Mobile robots equipped with robotic arms are autonomous machines, which allow them to navigate their surroundings and perform complex tasks. These kinds of mobile robots are becoming increasingly common in research and laboratories because they are versatile and perform repetitive tasks precisely in unpredictable environments. In distribution centres, they are especially useful for order picking, packaging, and shipping form one place to another place.

The University of South-Eastern Norway (USN) has a mobile robot Leo Rover which is equipped with robotic arm. It is physically available at Porsgrunn campus, accessible for on-campus students. The physical rover looks like shown Figure 1-1, when it received on the first day of master thesis project.

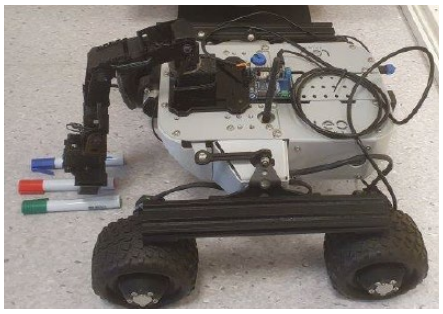


Figure 1-1: Leo Rover trying to pick up a red object.

1.2 Objectives

The primary objective of the project is to develop a computer vision-guided grasping system for the Leo Rover using its robotic arm. The system is combined with computer-vision to enable the rover to identify small objects within its nearby area, determine the optimized approach for grasping and retrieve the object using its robotic arm. Along with robotic arm, rover is modified by assembling Lidar sensor and ZED2 stereo camera to perform autonomous navigation and real-time object detection respectively. Key technologies also include installing ROS on Linux platform, setting-up necessary packages for rover and its electronics components, visualize rover in both simulated and physical environment, and perform motion planning.

1.3 Methods

There are various methods used to fulfil the project requirements, as outlined following stages:

- Finalizing the rover design:
    - It involves assembling and integration of Lidar sensor and stereo camera.
- Establishing the project environment:
    - It includes the installation of ROS on Linux platforms and configuration of necessary packages.
- Testing and evaluation:
    - Simultaneous localization and mapping (SLAM) in simple and complex environments.
    - Real-time object detection using fiducial marker system (AR-tags).
    - Implementation of motional planning using *MoveIt*
- Risk assessment using preliminary hazard analysis (PHA)

1.4 Scope

The master thesis project focusses on the Leo Rover, an open-source ROS-based mobile robot. The project has a specific timeframe to fulfil the tasks requirements which were set from 08.01.2024 to 15.05. 2024. The research includes testing and evaluation of the rover equipped with arm and its attached sensor performance in an actual and simulated environment. However, it can be extended to perform further exploration by using its features.

1.5 Contribution

The master thesis contribution fucuses on the integration of AR-tag (Fiducial marker system) based small objects and robotic arm manipulation techniques into the Leo Rover platform, to enhance the grasping capabilities. This integration involves writing Python code, conducting testing and evaluation on both simulated and physical environments. While previous research has explored pick-and-place tasks using AR-tags, this thesis addresses the important aspect of performing grasping action based on the detected tag's pose (position and orientation), where the rover is empowered to accurately locate and grasp objects within the environment.

1.6 Software tools

There are various software tools related websites are utilized throughout the thesis, including:

- Ubuntu 20.04 (Focal) as Linux operating system.
- ROS Noetic Ninjemys for controlling rover's functionalities.
- Creo Parametric for designing 3D designs.
- Micro for making system architectures diagrams.
- *MoveIt* for motion planning and manipulation using rover's arm.
- Rviz for simulation and visualization of rover and its connected devices.
- Python programming language.
- Visual studio code editor for writing Python code.
- ChatGPT for troubleshooting and research purposes.
- StarUML for drawing use case, classes etc.

1.7 Report Structure

The structure of thesis report as follows:

Chapter 2: Literature review describes the literature studies including computer-vision, sensor fusion in robotics, machine learning technique for object handling and robotic adaptive grasping.

Chapter 3: System architecture describes the specifications, software, and hardware structures of Leo Rover.

Chapter 4: Sensor integration describes assembling of rover's sensors, stereo camera, setup, and configuration of necessary ROS packages.

Chapter 5: Autonomous navigation describe how *leo_navigation* ROS package is incorporated into rover for autonomous navigation and SLAM (Simultaneously Localization and Mapping) including relationship of ROS *nodes* and *topics*.

Chapter 6: Real-time object recognition & detection describe the multiple AR-tags (Fiducial markers) real-time detections and recognition using both RPi and ZED2 stereo cameras.

Chapter 7: Optimized grasping position describe the implementation of motion planning and execution using *MeovIt*, preliminary testing for both simulated and physical rover' arm using Python script and presented optimized results for picking and placing AR-tags based small objects.

Chapter 8: Risk assessment and ethical considerations describes preliminary hazards analysis and ethical considerations related to rover and its arm.

Chapter 9: Discussion describes the how implications, limitations, challenges, and their solutions are addressed.

Chapter 10: Conclusion describes the summary of the thesis and future work for research development.

2 Literature review

This chapter provides a comprehensive overview and analysis of the existing research studies related to three important topics: computer-vision, sensor fusion technologies, machine learning for object handling and adaptive grasping system in robotics.

With the help of technological advancements in sensor fusion technology and obstacle detection algorithms, automated driving systems revolutionized more advanced transportation. It heavily relies on various types of sensors to perceive the environment surroundings, and the performance of multiple integrated technologies which directly impact safety and feasibility. Sensor capabilities and technical performance were evaluated throughout the article. It primarily focused on vision cameras and Lidar sensor, which are commonly used in autonomous vehicles. Automated vehicles or self-driving vehicles can perform all the functions of conventional vehicles, but they can also perceive the environment and navigate the target with minimal or zero human involvement. A precedence research report predicts that from 2020 to 2027, sales of autonomous vehicles which were embedded with multiple sensors had increased by 63.5%. Globally, these vehicles' market reached approximately 6500 units in 2019 [3].

Two types of sensors were analyzed including proprioceptive and exteroceptive sensor. These sensors allow the vehicle to perceive their surroundings using localization for path planning and decision-making capabilities. Most importantly to control vehicle motion based on information which was acquired by sensors. Primarily multiple vision cameras, Lidar, ultrasonic and IMU sensors were utilized to determine the absolute relative position of autonomous vehicle. The working principle of a camera lay on its lens which was used to detect the light emitted from its surroundings on a photosensitive surface to produce neat and clean images. It also had the capability to detect both static and moving obstacles with its field of view and deliver high-resolution images. In the case of road traffic when the vehicle was on the road, all these capabilities allowed the vehicle's perception system in a way, so it identified the road signs, traffic lights, road lanes and barriers properly. In autonomous vehicles, cameras were commonly used for detecting traffic signs, traffic light signals and road lanes but due to high dependence of camera performance in the environment, it often needed to be fused with other sensor data such as Lidar sensor which ensured the perception ability.

Lidar technologies has been used for development of ADAS (Advanced Driver Assistance System) and AD (Automated Driving) vehicles. It is a remote sensing device that operates on the principle of emitting infrared beams or laser light, bounce off target objects to detect them. These bounced reflected laser lights used to measure distance of objects from the Lidar sensor. In most autonomous systems, including industrial robots and self-driving cars, sensor fusion plays an important role in overcoming the detection uncertainties. Similarly, both camera and Radar sensor were also fused to provide high-quality images for detection purposes. Several factors are compared to analyze the sensors perform, Table 2-1.

Tabel 2-1: Comparison of common sensors used in self-driving cars [3].

Factors	Camera	LiDar	Radar	Fusion
Range	~	~	✓	✓
Resolution	✓	~	✕	✓
Distance Accuracy	~	✓	✓	✓
Velocity	~	✕	✓	✓
Colour Perception	✓	✕	✕	✓
Object detection	~	✓	✓	✓
Object classification	✓	~	✕	✓
Lang detection	✓	✕	✕	✓
Obstacle edge detection	✓	✓	✕	✓
Illumination conditions	✕	✓	✓	✓
Weather conditions	✕	~	✓	✓

The table provides a comprehensive comparison of the common sensors which were used in self-driving cars, including camera, Lidar, and Radar sensors, considered as technical characteristics as well as other external factors. There are three symbols "✓", "✕" and "~". According to "✓" represents the sensor operate competently under the given specific factor, "✕" indicates that the sensor does not operate well under the specific factor and "~" represent the sensor perform the tasks reasonably good.

There were three types of fusion methods used to combine the sensor data from various sensing modalities in Multi-Agent Service Function Chain Migration Framework (MSDF): low-level fusion (LLF), mid-level fusion (MLF) and high-level fusion (HLF). HLF method was compared to the other two methods, and it discarded lower-confidence classifications, whereas LLF required precise extrinsic calibration and temporary calibration. MLF fused multi-target features extracted from sensor data from a given environment and contextual information. In addition, sensor fusion techniques were extensively classified into classical and deep learning sensor fusion algorithms. Whereas classical algorithms utilized theories of uncertainty to fuse sensor data, while deep learning algorithms processed raw data and extracted features for intelligent tasks. Along with it, various other algorithms such as YOLO (You Only Look Once), SSD (Single-stage Object Detection), VoxelNet, and PointNet had proposed for object detection in autonomous vehicles.

Based on ARS (Autonomous Robotic System), autonomous robots are equipped with several sensors to detect sound, light, temperature, or pressure. Studies [4] have shown that autonomous vehicles can facilitate human life to provide safer transportation, but a simple mistake can cause tragic accidents. Therefore, the data from the sensors must be as accurate and reliable as possible for implementation. Camera and Lidar sensor were two of the most common electronics devices used in autonomous systems excessively, but they each have strengths and weaknesses.

For example, when the light intensity was too high/too low in camera images, pixels lost image processing tasks. Similarly, Lidar sensor was sensitive in bad weather conditions which lead to negative consequences for obstacle detection. However, when the information was given by a sensor in the form of fusion, then more positive results were obtained based on the dataset.

The study was carried out on a mini mobile robot [4] supplied by the Opezeka, used in the Massachusetts Institute of Technology (MIT)'s autonomous vehicle racing competition. The kit includes some of the same features which have been utilized on the Leo Rover model, such as ZED stereo camera, Lidar sensor and Nvidia Jetson developer kit. They have the capability to process computer vision algorithms. The camera was used to capture high-resolution images for image processing techniques. The result has shown, the raw data obtained from the camera and Lidar sensor were fused with the DL and CNN models for object detections. Furthermore, at the end of model representation, the fusion model was compared with the Lidar sensor and camera model, which gave better results than the individual sensor applications.

For the data collection, the data set to be used for training and testing operations was collected from the laboratory. The data set consisted of 5126 camera images and converted into grayscale Lidar point space data. The Rplidar A2 Lidar sensor on the car kit was used to detect obstacles up to 25 meters away, but the vehicle was not able to perform autonomous driving efficiently with this data set. To solve this problem, the pixel values o Lidar measurements at 15 cm and closer distances has changed to 255 as it was shown in Figure 2-1:

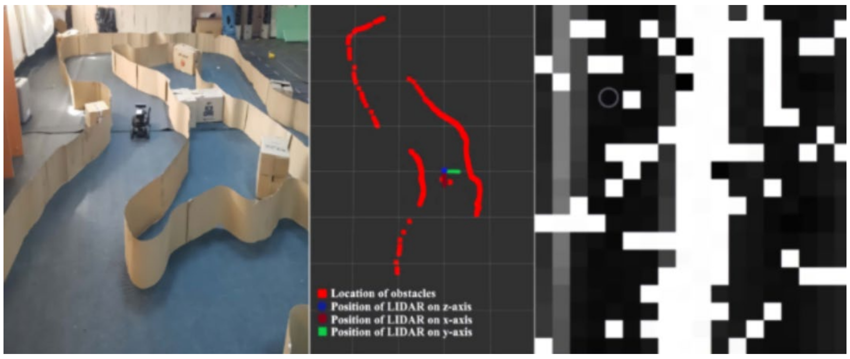

Location of obstacles
Position of LIDAR on z-axis
Position of LIDAR on x-axis
Position of LIDAR on y-axis

Figure 2-1: Grayscale transformation [4].

Using ML models, it was preferred to evaluate the performance of predicted values based on the learning process. To do this, MSE (Mean Square Error) state was employed to measures the regression curve from a series of points. When the MSE value was close to zero, the learning process considered as efficient, and the predictions produced near to real value as shown in Figure 2-2.

In the case of regression model, the accuracy metric looked different as compared to training and validation, that is why the sometime vehicle complete AVP (Automated Valet Parking) autonomously without any problems. Another evaluation criterion was changing in error functions during training and testing using camera, lidar sensor and combined data. Training and testing error functions were closed to each other if the process results were efficient. Overall results shows that performance become smoother when both electronic devices were fused.

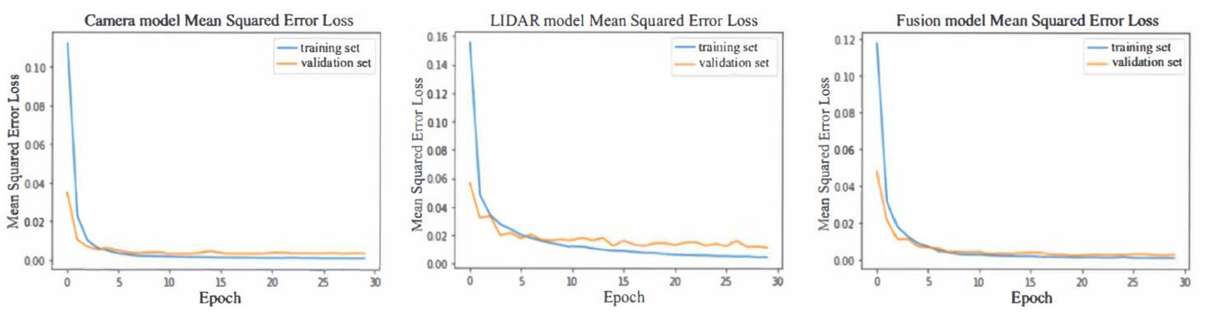


Figure 2-2: Loss graph of using camera, Lidar and fusion data respectively [4].

Unmanned vehicle system [5] was designed using multi-modal sensor fusion focuses to solve the problem of poor performance of positioning accuracy using low-cost sensors. System framework was analyzed to understand how SLAM algorithm used on RPi to sense the actual position of the unmanned vehicle. The Figure 2-3 shows that how navigation system was composed with GPS and IMU, fused with image data taken by the binocular camera T265 throughout Kalman filter to achieve the accurate position of the vehicle. In computer vision, binocular vision localization plays an important role for providing path planning to mobile robots. Throughout the results, the quality of path planning experienced not effective due to having binocular vision as it had low sensitivity to perceive weak texture features. Therefore, in the case of binocular vision cameras with missing scene features, the data input needed to be effective in compensating for the lack of spatial coordinate information of the camera to obtain optimized results.

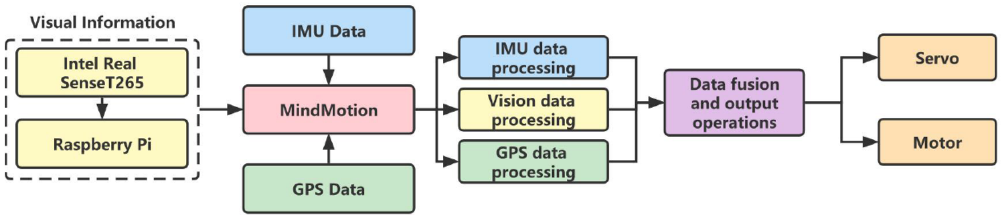


Figure 2-3: System framework for unmanned vehicle [5].

Several sensors were used in autonomous mobile robots to acquire information from its environment. The definition of sensor fusion given by Hall and Llinas [6]: by combining the data which is collected by multiple sensors with information from their associated databases, sensor fusion techniques enhance the performance of mobile robots and provide ability to make more precise decision that could be achieved using a single sensor. It is important to note that there were three fundamental ways mentioned how to combine sensor data i.e. competitive, complementary, and cooperative. Competitive types of sensor fusion were used for fault tolerance. The fault tolerance for a system requires exact information of how it fails. By configuring the sensor in this way, the risk of an incorrect indication was reduced. The reduction of noise by overlaying two camera's images was one example of a competitive method. In complementary type of sensor configuration, system ensures that the sensor was not dependent on each other but rather complement one another by providing different measurement. It helped to solve the incompleteness of sensor data and was particularly useful for localization.

An example of complementary, employing multiple cameras to observe different areas of mobile robot surrounding to build up a picture of the environment. In order to execute complementary method, the relevant information was available from two different sensors to create data that would not have been feasible from a single sensor alone. In stereoscopic vision, this cooperative method was used to generate 3D of sensor by combining two dimensional images from two cameras at slightly dissimilar viewpoints.

Object detections solution [7] using deep learning on 3D point cloud was discussed for the measurement of shape and depth of information of targeted objects in small and medium-size enterprises (SMEs). Three challenges have addressed including 1) detecting workstation for human with increased robustness in the SME environment; 2) Navigating and localizing the mobile manipulator in workstation precisely; 3) developing suitable tools for gripper to perform stable and precise manipulation for production tasks.

Technical details of the detections and localization of the target object were utilized using two methods including data acquisition and dataset establishment. Localization of the mobile robot was conducted using pre-organized maps which were built by SLAM. Due to generating localization error when the layout of nearby objects target were changes. It preferred to use precise localization techniques which were calculated manually using transformation matrix method from the targeted object to the robot based on detection results. At the end, the evaluation and performance were performed based on detection method and experimental results. Point cloud dataset was utilized which consist of five parts; printing machine, laser cutter, human, charging station, plug and sockets. To overcome the complexity for difference size, labeling techniques for training performance. KITTI dataset [7] referred to train the mobile robot where 4400 training samples were taken, which divided into training dataset (80% samples) and the evaluation dataset (20% samples). The main goal was to perform automatic plug-in charging tasks using a mobile manipulator. In addition, PV-RCNN was trained using dataset including these five parts.

Transformation was required when robotic arm needed to perform manipulation tasks. As shown in Figure 2-4, relative position was required to calculate by the transformation from targeted objects to the mobile robot.

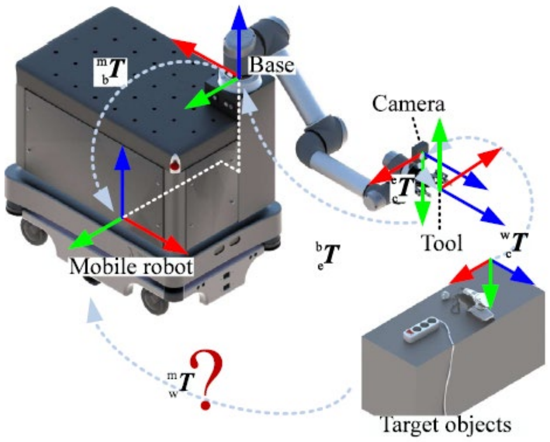


Figure 2-4: Coordinates and transformation matrices of the system [7].

Coordinate frames TCP (Tool Center Point) associated with mobile robots, the manipulator base, the manipulator, camera, and targeted object. All these frameworks were indicated by sub-indexes m, b, e, c, and w, respectively from the targeted object to the mobile robot [7];

$$ {}^{m}_{w}T = {}^{m}_{b}T . {}^{b}_{e}T . {}^{e}_{c}T . {}^{c}_{w}T \quad (1) $$

The equation presented as a co-related form with other transformation points which observed thought arrows symbols in Figure 2-4. Where the matrix ${}^{m}_{b}T$ represents last joint of the manipulator, and matrix ${}^{c}_{w}T$ denotes the transformation matrix from the targeted object he 3D camera. The matrix ${}^{b}_{e}T$ from the TCP to the base of the manipulator obtained by kinematics of the manipulator and matrix ${}^{e}_{c}T$ from the camera to the end-effector obtained by the hand-eye calibration method.

After the transformation, using machine learning method, training was performed on a dataset of 4000 samples and an evaluation dataset of 400 sample, model performance was evaluated by comparing overlap volume between predicted cuboid boxed and ground truth boxes. Where PV-RCNN training was involved with its network parameters to determine the impact of the number of cloud points and epochs on detection performance. Results demonstrated that the model achieved stable detection result with 1000 cloud points and epochs when it started to exceed 500. However, almost similar results have achieved for average overlap volumes of charging station, printing machine, humans, socket and plug presented in paper [7].

Subsequently, two-finger gripper was also used which directly mounted on the end-effector of robotic arm. It helped to grasp small items such as plugs and wood sheets post-cutting, but this gripper arm was extended by integrating vacuum module. This vacuum module allowed the robotic end-effector to handle various production components within framework such as needles, name tags and medals. However, there were some limitations when dealing with large, thin board-shaped components like wooden sheets or cardboard. To address this challenge, two solutions were proposed. One of them involves purchasing a tool changer, while the other one refers to leverages a module that gripper can directly grasp and charge. Consequently, the vacuum module has appropriate geometry and electrical connections to the on-board 12V vacuum pump and the 24C 3/2 valve as it can been seen in Figure 2-5.

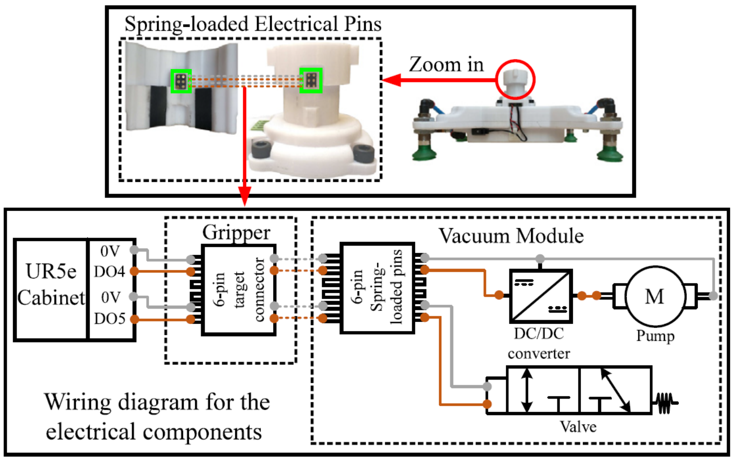


Figure 2-5: The vacuum module and the wiring diagram of the electrical components [7].

Results were analysed throughout experiment that shows the integration of SLAM mapping which enables the mobile manipulator to navigate efficiently on working stations, where both localization and calibration processes were executed. The precision achieved in calibrating the robot's position relative to the printing machine, facilitated by 2D camera calibration. In addition, the integration of real-time human detection ensures operational safety, with the mobile manipulator adapting its path accordingly. Finally, the application of the PV-RCNN model extends to the detection of charging stations, with subsequent point cloud to facilitate accurate plug and socket positioning for efficient charging.

Robotic grasping types are divided into three types: familiar, known, and unknown. Known objects means that it is included in training datasets where robotics can generate and execute grasping poses based on prior experience. In contrast, unknown objects and familiar objects have not been encountered previously, but familiar objects have some resemblance with training datasets. However, grasping known objects has been well-established in several industries. It is observed [8] that current research revolves around developing deep learning grasping models for unknown objects, with methodologies DCNN, RGB images, and depth images. Furthermore, a general grasping process was discussed for both offline generation and online grasping phases. In the offline phase, the training was performed on grasping different objects based followed by quality of each grasping process. After that, a grasping model was prepared based on the training data which was stored in a database for further development. On the other hand, in the online phase, object detection tasks were performed using vision-based techniques and mapped to the model database. Afterward, the learning database was used to generate a grasping pose and discard those objects that cannot be grasped.

In the industrial sector [9], vision-based pick-and-place tasks using industrial manipulator were used to grasp the objects. But in that case, the 3D object model needed to be known in advance. There were two ways for grasping objects. It either analyze the grasping shape of the object and try to find a proper way to grasp the object or could perform feature matching and shape recognition to find suitable position for the manipulator to perform grasping task including pick-and-place tasks. However, it had some limitations and a sensitive approach. For example, in that case, the grasping object was not known in advance, then their grasping approaches failed. With the help of machine learning algorithm particularly deep learning methods were referred to perform automatic object grasping tasks.

The object grasping technique combined with computer vision-based, particularly three tasks such as object detection/localization/recognition using a deep learning reinforcement learning algorithm, shown in Figure 2-6. A robotic pick-and-place system was developed, where YOLO detected objects based on interest of image, captured by stereo camera. SAC used to provide desired grasping location in the image point based on information of dept images. At the final stage, grasping point (2D-image plane) convert to 6D desired grasping pose in the Cartesian space (linear movement). By doing this, the robot manipulator grasped the object and placed it at the designated position. The whole process was based on a reward mechanism.

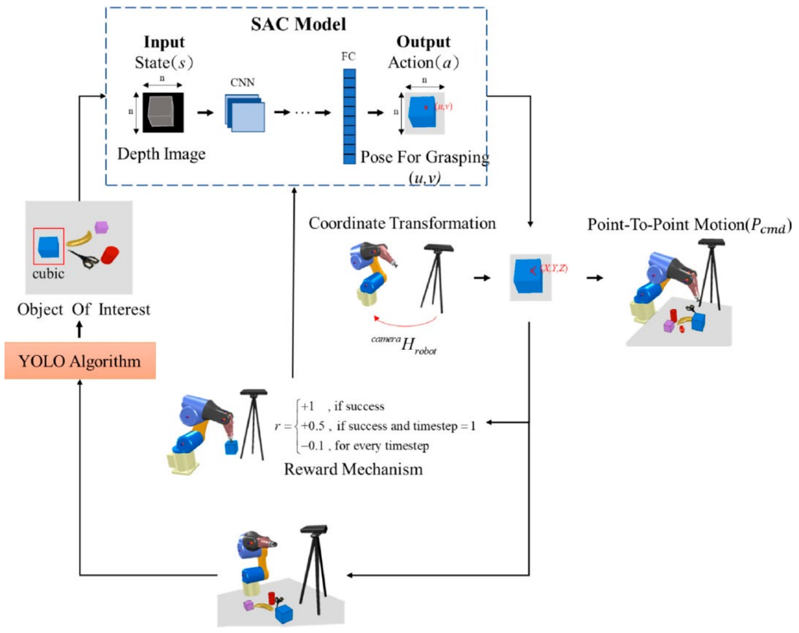

SAC Model
Input
State(s)
Output
Action(a)
FC
CNN
Depth Image
Pose For Grasping
(u,v)
Coordinate Transformation
Point-To-Point Motion(P_{cmd})
cubic
Object Of Interest
YOLO Algorithm
${}^{camera}H_{robot}$
$r = \begin{cases} +1 & \text{, if success} \\ +0.5 & \text{, if success and timestep} = 1 \\ -0.1 & \text{, for every timestep} \end{cases}$
Reward Mechanism

Figure 2-6: Schematic diagram of robotic pick-and-place based on computer vision [9].

Both real experimentation and simulator environments were employed to achieve the tasks. The primary purpose of the simulated environment was to facilitate the training and testing of deep neural networks. On another hand, the real experiment involved the utilization of an articulated robot manipulator produced by ITRI (Industrial Technology Research Institute), boasting a 6-degree-of-freedom (6-DOF) range of motion. Multiple AC servomotors were integrated at each joint of the robot manipulator, with a vacuum sucker (capable of handling a maximum payload of 3 kg) attached to the end-effector. Additionally, a Kinect v2 RGB-D camera was utilized for vision sensing tasks. This camera captured images of various objects, including apples, oranges, a banana, a cup, a box, and building blocks, with the YOLO algorithm employed for categorization. The experimental setup included two robust computers responsible for controlling the 6-DOF robot manipulator and vacuum sucker. To train the YOLO v3 model, the COCO dataset was utilized, with training conducted over 45,000 iterations until the loss function converged to 0.0391. Subsequently, the performance of the trained YOLOv3 model was evaluated by placing several objects randomly on a table, and the detection results were compared with those of the trained objects.

The OpenCV library method was also used to find the actual position of board including four corners. In this paper, robotic manipulator was particularly two approached used for object recognition: RGB and depth-based object recognition. Based on the requirement on board, a depth-based method was developed to detect objects. The major reason was that it allowed the manipulator's detection system to solve the problem even if objects fail to detect when object was in the same color background or due to changing light intensity. In results, it was concluded that the object recognition system could only work for 2D depth images effectively and it could not recognize 3D object information.

3 System architecture

This chapter provides detailed information about the specifications of rover, configuration of hardware and software architecture used throughout the thesis.

3.1 Specifications of Leo Rover

The Leo Rover is a small-sized, 4-wheeled, open-source robotic platform developed by Fiction-lab. It is mainly based on Raspberry Pi module which runs on Ubuntu Linux 20.04 with ROS (Robot Operating System) [10]. This rover has been modified to fulfil the requirements of a master thesis project. However, the Figure 3-1 shows assembled rover without attachment of sensors and electronic devices.

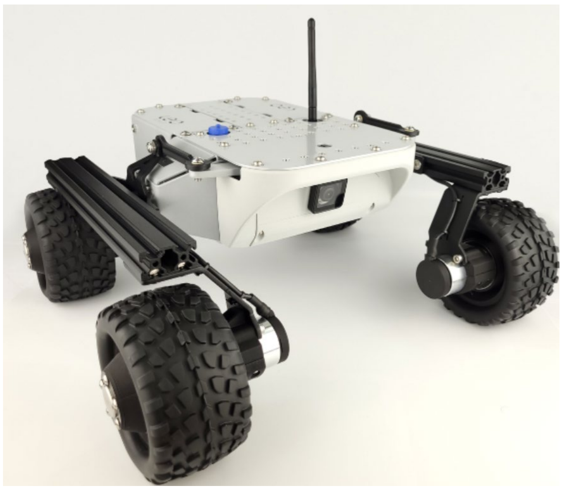


Figure 3-1: A 4-wheel drive-controlled Leo Rover without modifications [11].

A quick overview of the specification and technical drawing of Leo rover can be seen in Table 3-1 and Figure 3-2:

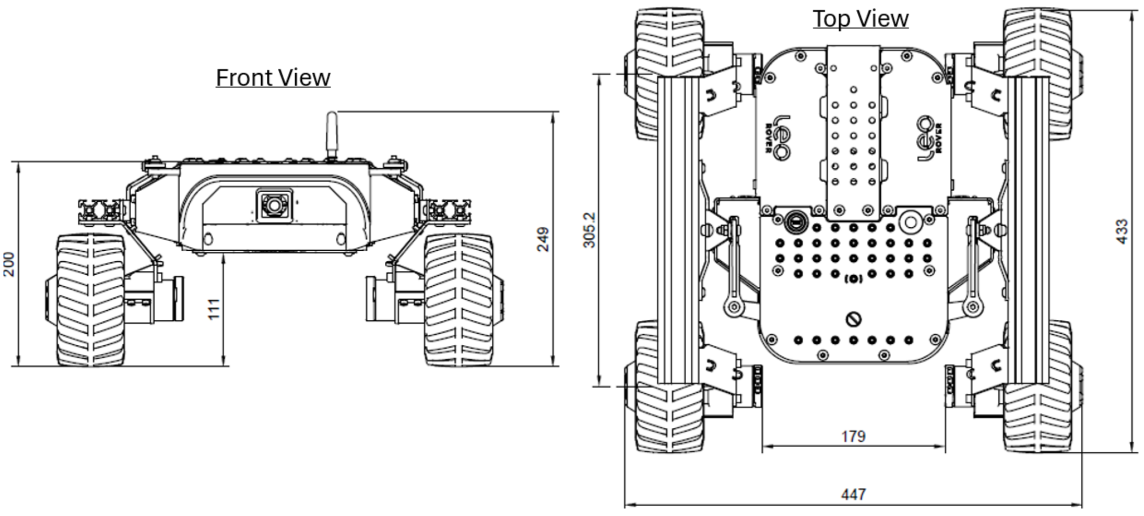


Figure 3-2:Technical drawing for Leo Rover [10].

Table 3-1: A quick overview of the Leo Rover specifications [10].

Leo Rover			
Size and performance		Weight	6.5 kg
		Dimension	447 mm (length) x 433 (width) x 249 mm (height)
		Max. linear speed	Approximately 0.4 meter/s
		Max. angular speed	Around 60 degree/s
		Running time	4 hours
		Connecting range	Up-to 100 meters (with live video streaming)
Payload		Payload capacity	Approximately 5 kg
		Upper mounting platform Dimensions	299 mm x 183 mm
		Hole grid	18 x 15 mm
		Holes	40 x Φ 7mm + 22 x Φ 5,5mm
Components	**Wheels**	Motors	4 x in-hub DC motor with 73.2:1 planetary gearbox and 12 CPR (Counts Per Revolution) encoder
		Wheel diameter	130 mm
		Tire material	Rubber with foam insert (non-pneumatic)
	Battery	Voltage	11.1 V DC
		Capacity	5000 mAh
		Type	Li-ion with internal PCM (Phase Change Material)
		Max. current	8A (total for the whole Rover)
	Camera	Camera resolution	5 MPx (Mega Pixels)
		Lens	Fisheye with 170 deg field of view
	Network	Primary modem	WiFi 2.4 GHz access point with external antenna
		Secondary modem	WiFi 2.4 GHZ + 5 GHz on internal RPi antennas for connectivity
Software		Operating system	*LeoOS* based on Ubuntu 20.04 + ROS
		UI (User Interface)	Ready-to-go UI located under '10.0.0.1' when using standard Leo Software Image
		Firmware	Open-source firmware
Electronics boards		Main computer	Raspberry Pi 4B
		Microcontroller	LeoCore2 board

In general, a robotic system is a complex system represented by multiple sub-systems, known as mechanical, actuation, sensory and control system [1]. The most essential component of a robotic system in Leo Rover is mechanical system, which endows with locomotion apparatus (four-wheeled) and manipulation apparatus (robotic arm including end-effectors). Actuation systems execute an action including both locomotion and manipulation to move the mechanical components of mobile robot, which is done through Bühler Gear DC and its drivers.

The sensory system is responsible for perception which acquires data from the internal status of mechanical system (i.e. identifying position of transducer using preceptive sensor) and external status of the environment (i.e. exteroceptive sensors and vision system cameras). Lastly, the control system works like a controller with coordinating and integrating the actions from sensors.

3.2 Hardware components for the rover

The Leo rover is powered by LeoCore (Core2-ROS) electronics board connected with GPIO input/output header of RPi via UART. The configuration of boards shown in Figure 3-3, demonstrates the utilized blocks (red color) in the rover. A 12V (5000 mAh) DC battery supplies power to both RPi and LeoCore boards. The primary function of LeoCore is to control the rover functionalities including four Bühler Gear servo motors, IMU sensor and facilitate communication with RPi board.

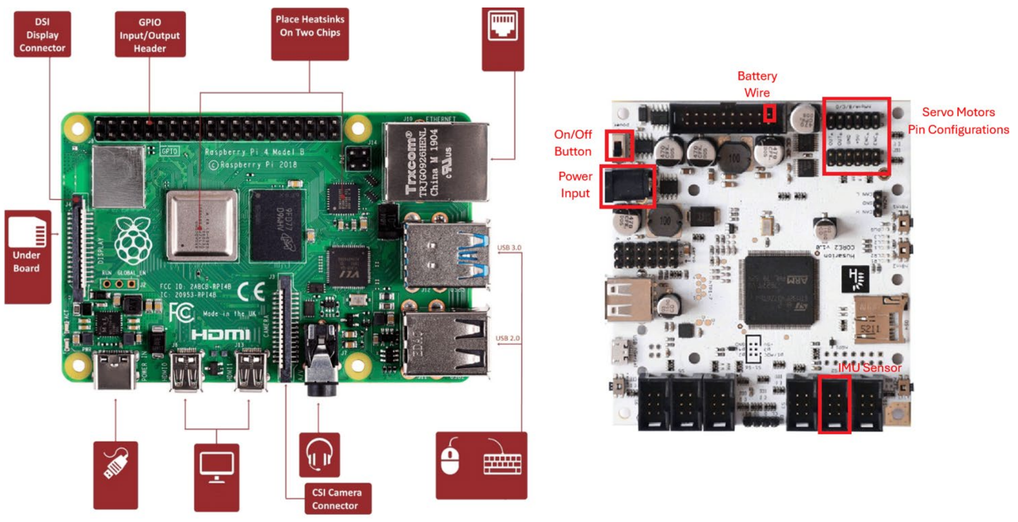


Figure 3-3: Raspberry Pi 4B module [12] and LeoCore2 boards [13].

To get the access point of rover's connectivity, a primary Wi-Fi modem operating at a frequency of 2.4 GHz (gigahertz) frequency and a secondary modem operating at both frequency of 2.4 GHz and 5 GHz on the internal RPi antenna are used. The external antenna is connected to the rover's RPi board via a u.FL connector, allowing to connect with internet and remote desktop access on a laptop, shown in Figure 3-4.

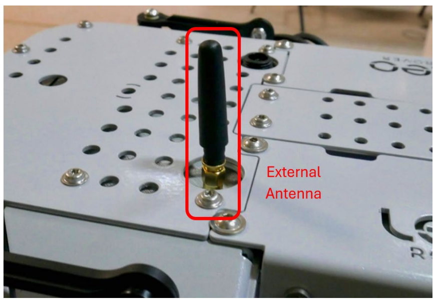


Figure 3-4: External antenna attached with rover's RPi board.

The remotely controlled PhantomX Pincher 4-degree-of-freedom robotic arm (shown in Figure 3-5) is employed to facilitate vision-based pick and place applications. Its primary function is to plan and execute grasping tasks to grab small objects near to the rover. Additionally, an Arbotix-M controller is used to control the five Dynamixel servo motors of Pincher arm, connected to RPi board via FTDI-USB cable. Before the assembling the Lidar sensor and stereo camera, the rover appears as shown in Figure 3-6. The small red circle on the arm indicates the Iss (1 till 5) of the servo motors.

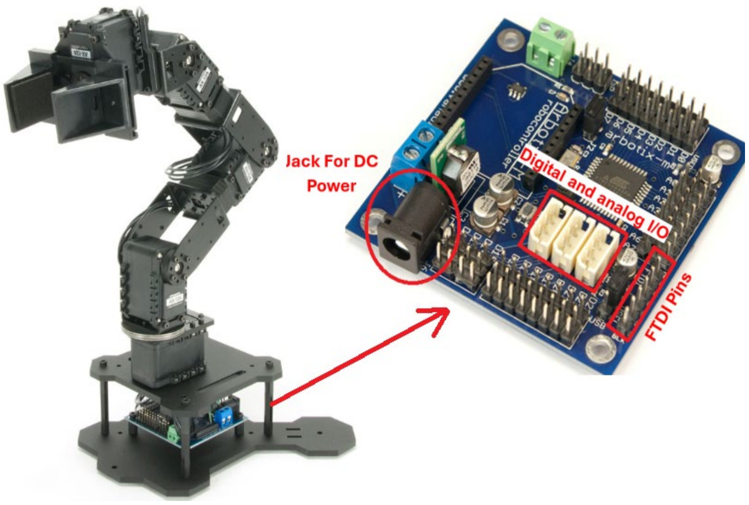


Figure 3-5: PhantomX Pincher arm and Arbotix-M controller [14].

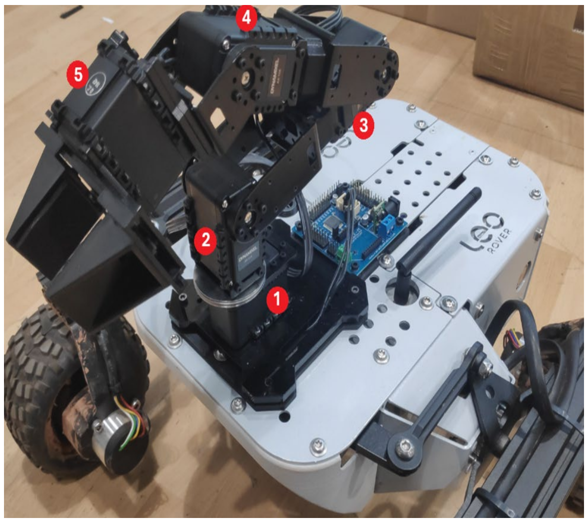
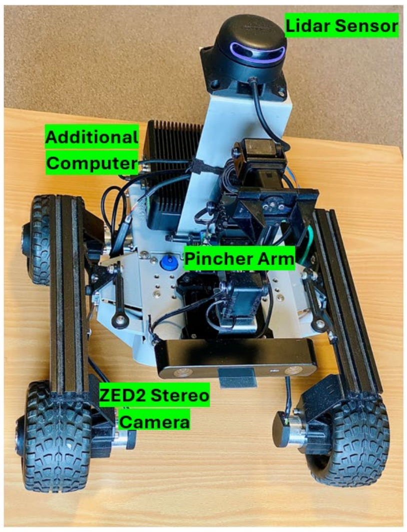


Figure 3-6: Pincher arm along with motors Ids equipped with rover.

To enhance the rover’s capabilities for autonomous navigation and computer vision, two most important electronic components are utilized along with Pincher arm: the Rplidar A2M12 Lidar sensor and ZED2 stereo camera along with additional computer Nvidia Xaviour NX reComputer Industrial J2012. The Figure 3-7 shows the upgraded version of rover after assembling and integrating with robotic-based system.



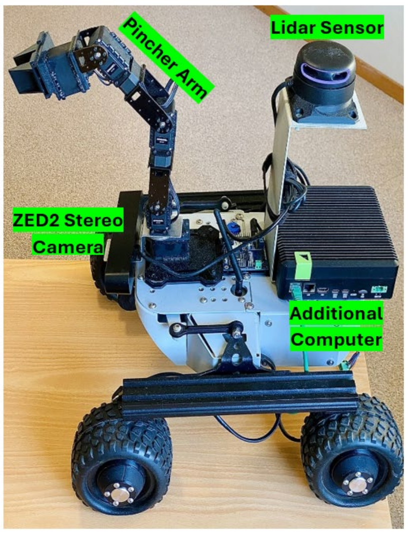


Figure 3-7: Enhanced Leo Rover: After assembling and integrating Lidar Sensor, ZED2 Camera along with additional computer.

3.3 Rover system architecture

A robotic system is a collection of sensors and actuators that interact with each other within the given environment. The primary purpose of the robotic system is to achieve a specific set of tasks. Along with these tasks, robotic system architecture is necessary for enabling the robot to achieve the goal without requiring a complex system. In general, the system architecture is defined by two parts including the structure and style [15]. The structure defines the way components are divided into a system and how they interact with each other. While the style refers to the computation concepts that define the implementation of the design.

The robots are designed with the combination of hardware and software architecture , where hardware is responsible for system assembling and software for system operations [16]. These components must be seamlessly integrated to enable a robotic system’s functions as expected.

3.3.1 Hardware architecture

The hardware structure of a mobile robot generally consists of four main important components: controller, sensors, actuator, and power system. The same concept is applied to Leo Rover to design its hardware architecture, illustrated in Figure 3-8.

The rover is mainly based on a microcontroller (RPi board) attached with another controller (LeoCore) via GPIO pins. These boards are empowered by 12V DC battery to provide the power supply including the boards itself and attached sensors. Sensors provide feedback about the robot environment, such as distance and relevant data. In this case, Lidar sensors is connected to the rover’s board via USB cable, to generate 2D maps for autonomous navigation using simultaneous localization and mapping (SLAM). Similarly, IMU sensor is connected to rover using IMU module (Grove – IMU 9DOF) to the LeoCore microcontroller board to enable the precise movements and positioning for the SLAM. Along with these sensors, two cameras including RPi 5-mega-pixel and ZED2 stereo camera are attached to the rover.

The ZED2 is a powerful AI camera that can capture images from two different lenses simultaneously, enabling depth perception capability and 3D mapping. It is linked with an additional computer, the Nvidia Jetson J2012 computer via Ethernet cable to the RPi board. While RPi camera is connected to RPi board via CSI connector. Moreover, to get rover's access point, external antenna is connected to the RPi board via USB cable. It also helps to connect with Wi-Fi network to access the internet facilities.

The final part is actuators which are responsible for the movement of rover and its robotic arm. This rover is designed for locomotion (4-wheeled), along with four DC gear motors, which are connected to LeoCore controller to perform linear and angular movements. On the other hand, the 4-DOF Pincher arm along with Arbotix-M controller is configured to the rover via USB cable to control the 5 Dynamixel servor motors, with specific Ids (1 till 5).

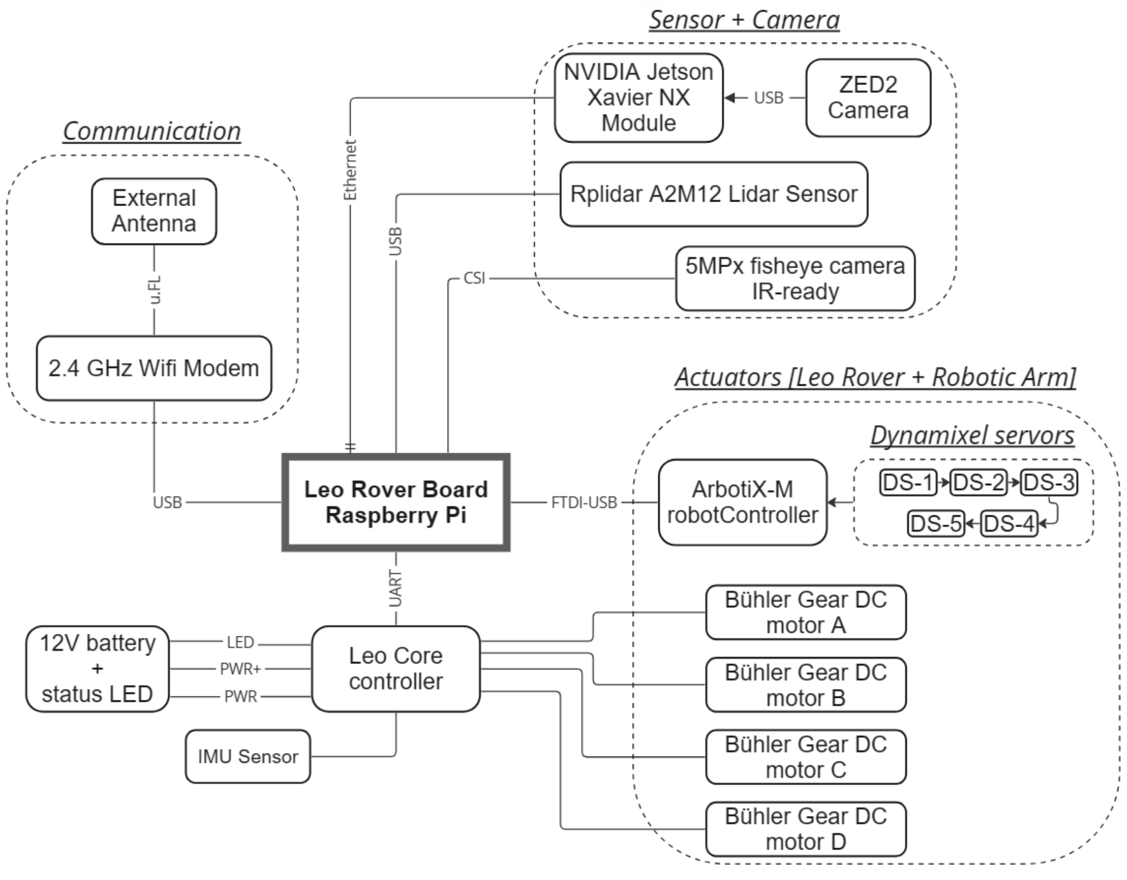


Figure 3-8: Hardware structure of Leo Rover [17].

Note: The figure is modified by Fiction-lab hardware structure, where the rounded rectangle shape represents the indication of connected devices and sensors, while dotted rounder rectangle shapes refers the specific sections.

3.3.2 Software architecture

Before making software architecture, it is required to make software analysis for the Leo Rover equipped with robotic arm, Lidar, IMU sensor, RPi and ZED2 stereo camera. The rover is based on a collective system including autonomous navigation, object detection and grasping system. In the whole system process, rover autonomously performs simultaneous localization and mapping (SLAM), detect, and recognize AR-tag Fiducial marker and perform pick-and-place activities within an environment.

There are various approaches to collect the system's requirements. Developing software architecture for the rover, the FURPS+ approach is employed to gather requirements for the rover's collective system. It is used to collect and classify requirements. It stands for functionality, usability, reliability, performance, and supportability, with the "+" sign indicating additional characteristics [18].

3.3.2.1 Functionality

The main functions of the system are as follows:

1. Autonomous navigation:
   - The rover should be able to navigate autonomously using SLAM.
   - It should plan the path and reach the targeted goal while avoiding obstacles.
2. Object recognition and detection:
   - The rover's RPi and ZED2 cameras should detect AR-tag markers.
   - It should also accurately identify, locate the detected objects, and provide its coordinates pose (position and orientation).
3. Object manipulation using *MoveIt*:
   - The rover's Pincher arm should be able to grasp AR-tag based objects.
   - It should be capable of picking up and placing grasped objects at specific locations.

3.3.2.2 Usability

Usability is the interaction between the user and system, as follows:

- Rover should be controlled by Web UI, accessed via web browser.
- It should also display a camera stream from the Web video server and output of the current battery voltage measurement.
- It should also include GUI of the controlled system to interact with the user to perform these tasks.

3.3.2.3 Reliability

Reliability covers a range of issues, ensuring that the availability of the system, implementation of error handling mechanisms, and enabling efficient recovery from failure, as follows:

- The system must be recoverable if error occurs and ensure the accuracy of sensor data such as Lidar and IMU sensor for reliable navigation.

3.3.2.4 Performance

Performance ensures speed, capacity, and utilization of resource, as follows:

- RPi board should ensure smooth operation for all tasks and timely response.
- If the sensors and electronic devices do not work properly as per the requirements, the system should be able to trigger indication to shut down or restart the system.

3.3.2.5 Supportability

Supportability refers to ease maintenance of the system, as follows:

- The system to be easily maintainable, allowing for updates its functionality in future.

3.3.2.6 "+"

"+" signs indicate additional characteristics of the system, as follows:

- System should implement security measurements to protect data transmitted by sensors and camera by the rover.
- It should ensure that only authorized users have access to interact with the rover.
- It should facilitate seamless integration with ROS for communication between its components.

3.3.3 Use case

A use case represents the main functions that the system performs. StarUML software is used to draw use case diagram, shown in Figure 3-9. Therea are five use cases which are being performed throughout the project. The actors (user interface and Leo Rover) are associated with a system using arrows. These arrows represent a conversation between the actor and the system components which are responsible for executing the use cases.

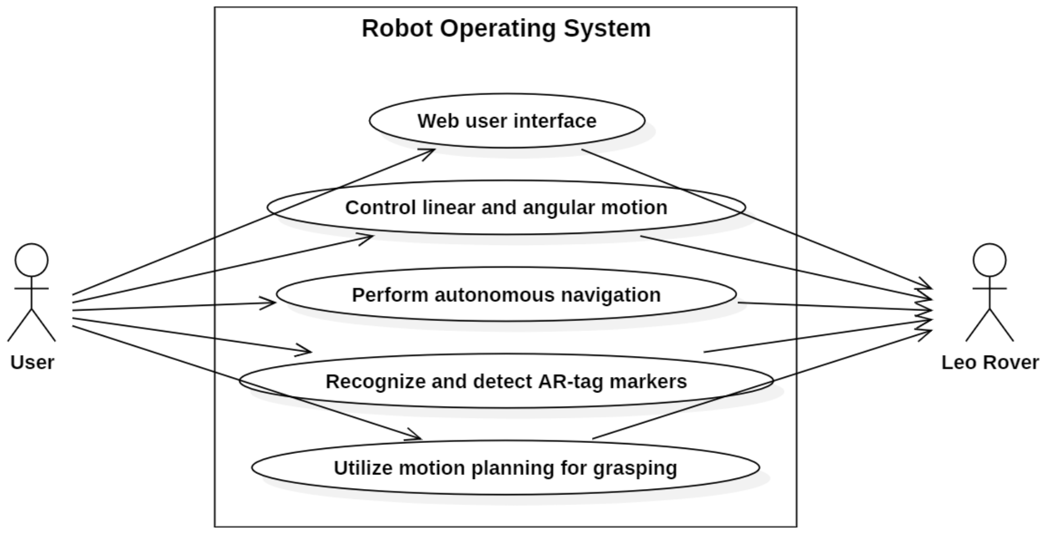

Robot Operating System
Web user interface
Control linear and angular motion
Perform autonomous navigation
Recognize and detect AR-tag markers
Utilize motion planning for grasping
User
Leo Rover

Figure 3-9: Use case for Leo Rover

3.3.4 Domain model

A domain model consists of a set of classes. Each class represents a specific entity within the domain model and shows the relationship between the entities. It also defines the attributes and operation of classes based on the requirement of the system. The purpose of creating a domain model is to establish a clear overview of system architecture, shown in Figure 3-10.

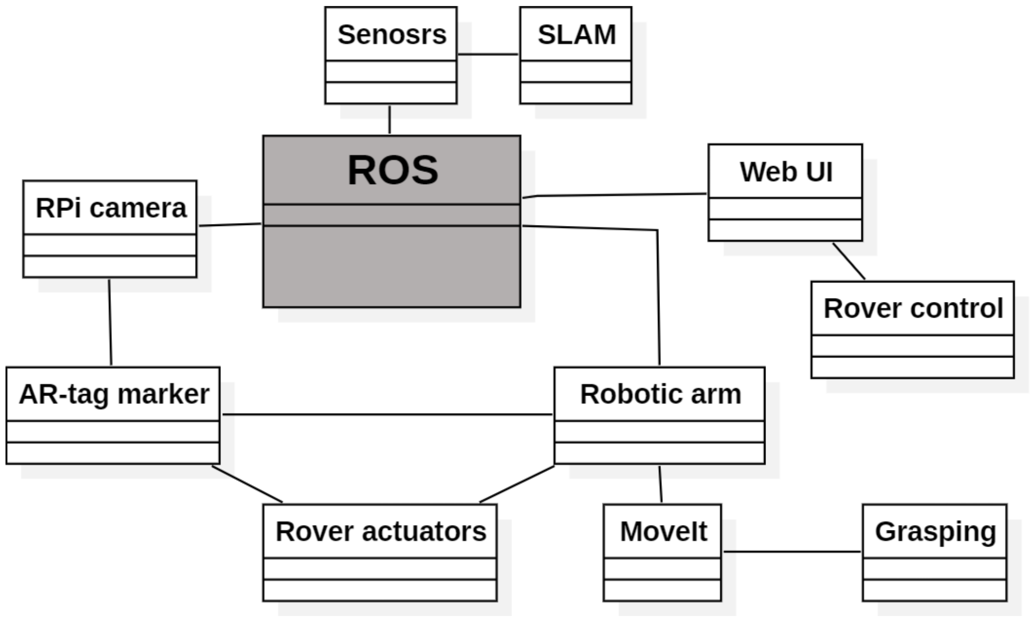


Figure 3-10: Domain model for Leo Rover

Note: The dark gray box shows the main operating system of rover and rest of the boxes show the representation of classes generated in StarUML software.

3.3.5 Use Case analysis

To describe the functionality and behavior of system requirements, a fully dressed use case document (FDUCD) is performed to identify the most important use case, “Control linear and angular motion”. The reason to prioritize is because it plays a core foundation to perform all related tasks. For example, it is essential for enabling precise movement, which is important autonomous navigation, object detection and robotic arm manipulation. Accurate movements ensure the rover planned path, avoid obstacles, and reach specific destinations to perform grasping activities.

The following points cover FDUCD, as follows:

1. **User case name:** Control Linear and Angular Motion
2. **Scope:** Web UI and Rviz software
3. **Level:** Monitoring rover’s movements while performing tasks
4. **Primary actor:** User
5. **Stakeholder and interest:** No stakeholder in this case
6. **Precondition:**

a) The rover should be power ON.
b) The Web UI or command console should be accessible.
c) The rover's sensors and motors are functioning correctly.
d) All ROS *nodes* and *topics* should be activated.

7. **Success guarantee:** The rover moves according to the specified linear and angular velocities without any error.
8. **Main success scenario:**
   a) User get the rover's access point to test the rover's movements including move forward, backward, or turn left, right via the web interface.
   b) The LeoCore controller sends the control signal to motors accurately.
   c) The system continuously monitors the rover's movements and provides feedback to the user.
   d) User stop or adjust movement speed (increase or decrease the speed of actuators) as based on feedback.
9. **Extensions:**
   a) If the command is invalid, the system displays an error message and prompts the user to re-enter the commands to the terminal.
   b) If there is any communication failure with the actuators, the system should send a message to the user to re-establish communication.
   c) The system automatically stops the rover when it collides with the boundaries.

10. **Special requirements:** The system must ensure the safety mechanism to stop the rover if it collides with boundaries/obstacle. Because it may damage the attached sensors which are costly to repair or replace it.
11. **Technology list:** Various types of input devices (RPi camera)
12. **Frequency of occurrence:** User needs to ensure RPi camera ROS *topic* list available.
13. **Miscellaneous:** The user can store the data for further analysis and debugging.

3.3.6 System sequence diagram

The system sequence diagram provide an overview of the system behaviour for selected use case. It based on main success scenarios from the use case analysis and used to represent the function requirement for the system, shown in Figure 3-11

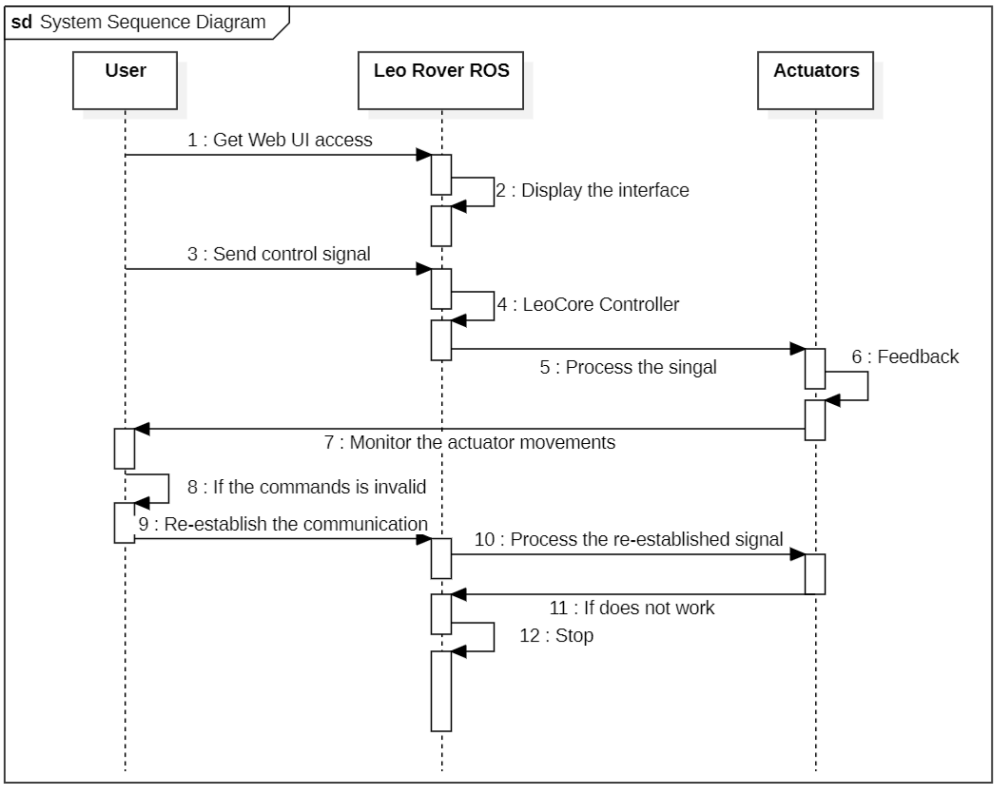

sd System Sequence Diagram
User
Leo Rover ROS
Actuators
1 : Get Web UI access
2 : Display the interface
3 : Send control signal
4 : LeoCore Controller
5 : Process the singal
6 : Feedback
7 : Monitor the actuator movements
8 : If the commands is invalid
9 : Re-establish the communication
10 : Process the re-established signal
11 : If does not work
12 : Stop

Figure 3-11: SSD for selected use case.

3.3.7 Development process:

To develop a software structure for the rover, Unified Process (UP) approach required to proceed software development. It consists on four phases, including, inception, eleboration, construction and transition phase [19]. As there are five use cases for the rover collective system. For each phase of iteration, only one use case will be selected to plan and collect the requirement, implement the analysis, desing and teste the deployed code until all use case are tested. Totale time for UP depends upon the resources, developer team and level of deteails required in each phase.

3.4 ROS *nodes* and *topics* for rover

To get better understanding for connected sensors for the Leo Rover, it is beneficial to familiarize some fundamental of robot operating system (ROS):

ROS is an open-source framework used for writing robotic software. It provides a set of libraries and tools to develop complex robot applications, including drivers, packages, algorithms, and communication protocols. For Leo Rover, ROS1 Noetic version is utilized to test the rover's functionalities. The biggest strength is that it provides an ability connect multiple ROS *nodes* together. The ROS *nodes* are pieces of blocks (a small software which can be written in Python/C++) contain *messages* (a set of information). These nodes share their information by using Publish/Subscribe protocols to ROS *topics*. *Topics* are like channels which exchange information between *nodes*. In other words, it allows *nodes* to share information without needing to know each other's identity. ROS *services* provide a way for *nodes* to request specific actions or computations from other *nodes*. ROS *messages* define the data structures (integers, floats, and arrays) for *nodes*, which describe the format and connection of information shared via *topics* and *services* and *actions* [20].

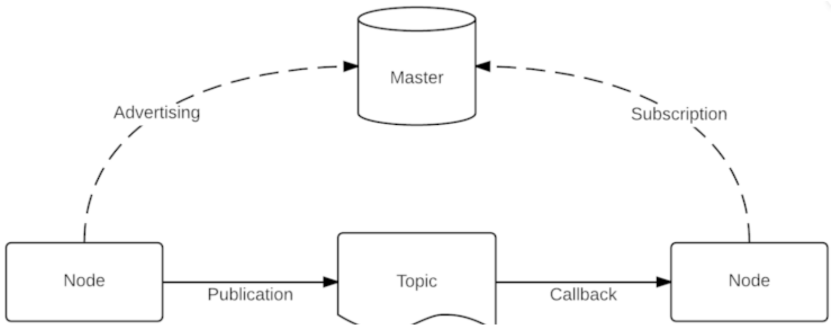


Figure 3-12: ROS-master-node-topic relationship [20].

For example, a Laser scan *message* might contain distance measurement from Rplidar A2M12 Lidar sensor. Everything is handled by the ROS *Master node*, which runs everywhere in the ROS network, and makes sure that all *nodes* are properly connected. In this master thesis project, ROS *master node* runs on the rover RPi board.

The structure for the ROS *nodes* and *topics* is modified using the ROS *rqt_graph*, which provides a convenient graphical display for rover's components and relationship [21]. It is commonly used in ROS environments for visualizing the connection between ROS *nodes* and *topics*, facilitating easier understanding, and debugging of the system's architecture.

The complete the structure, it is divided into two segments, shown in both Figure 3-13 and Figure 3-14. The first figure includes a diagram illustrating the ROS *nodes* and *topics* launched by Lidar sensor. Meanwhile, the second figure illustrates ROS *nodes* and *topics* launched by Pincher arm, RPi and ZED2 cameras.

The rover utilizes the Leo Operating System (*LeoOS*) based on ROS1 Noetic. The operating system mainly runs on two parts: *Firmware* and ROS *nodes* shown in both figures. Along with these two parts, the ROS core incorporates various other electronic equipment which are available with rover, such as IMU and Lidar sensor, Pincher arm, RPi and ZED2 camera.

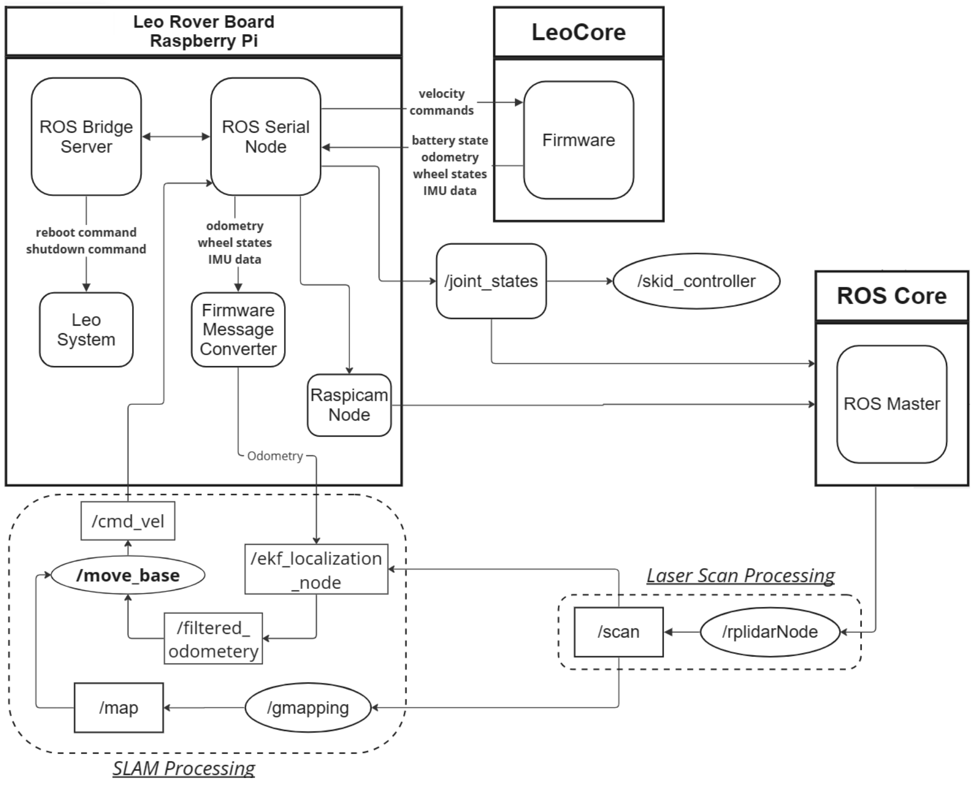


Figure 3-13: Software structure of Leo Rover (Segment-1) [22].

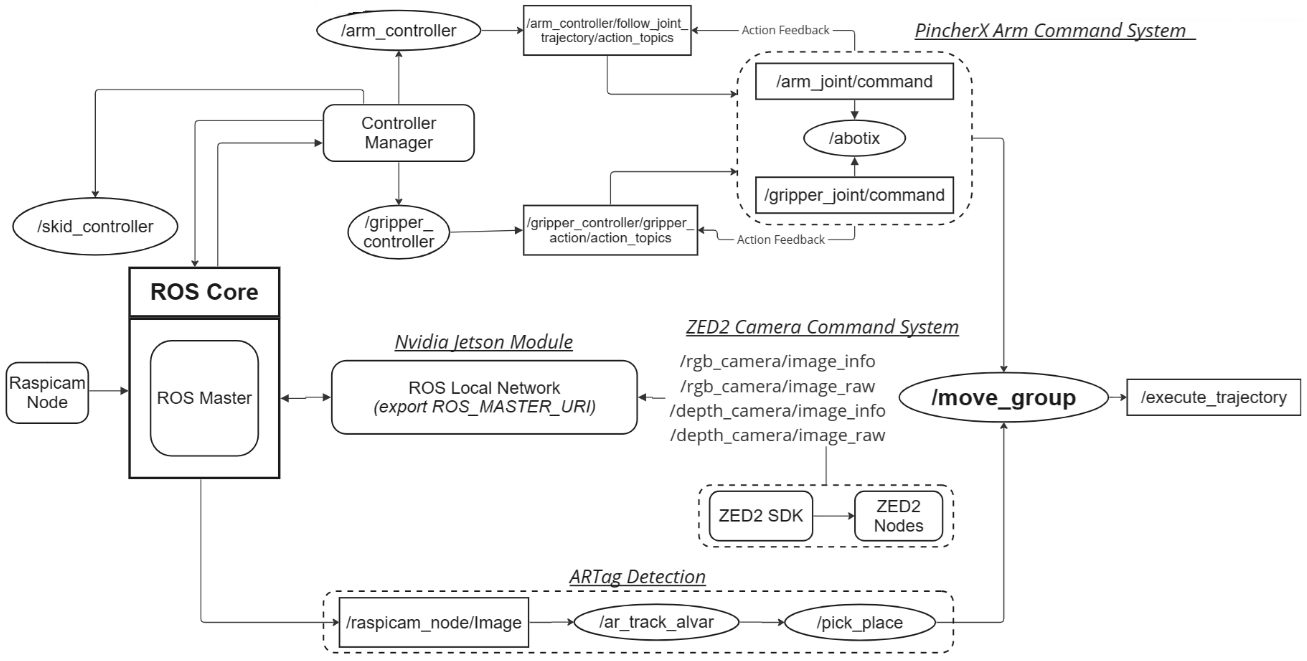


Figure 3-14: Software structure of Leo Rover (Segment-2) [22].

Note: The oval and rounded rectangle shapes represent the ROS *nodes*, while the rectangle with solid lines represents the ROS *topics*. The arrows between ovals, rounded and solid rectangles lines represent the Publish/Subscribe protocols. The dotted rounded rectangle shapes indicate the specific section.

In Figure 3-13, *Firmware* is the core program that runs on the processor of LeoCore (Core2 electronic board). It provides several functionalities to the RPi board through serial bus. The key feature of firmware based on *leo_firmware* package (GitHub) includes: differential drive controller (*cmd_vel* interface), battery voltage monitoring, wheel odometry calculation, wheel states monitoring (*joint_states* interface) and IMU support.

The differential driver controller is responsible for rover's movements (linear, angular) along with *cmd_vel* interface, which expresses velocity commands in free space into linear and angular movements, while wheel states continuously track wheel status along with *joint_states* interface, which holds data to describe the state of a set of torque-controlled joints. Moreover, IMU support utilizes data from the IMU sensor to measure the orientation, velocity, and gravitation force of rover. Lastly, wheel odometry calculates the rover position and orientation based on wheel rotations and movements.

When the RPi boots, a set of ROS *nodes* are initialized. These *nodes* allow different kinds of features via ROS *topics* and *services*. They are defined in *leo_bringup* package (GitHub). This package mainly consists of three elements: ROS *serial node*, *bridge server* and *Raspicam* (RPi camera) *node*. The ROS *serial node* establishes communication with *Firmware* of LeoCore (Core2 electronic board) and enables access to its functionalities through ROS *topics* and *services*. ROS *bridge server* generates a WebSocket which provides a JSON-API to ROS functionalities for non-ROS applications. It also facilitates Leo System *node* which provides both system shutdown and reboot operations through ROS *topics*.

Raspicam node publishes images captured by the RPi camera module to ROS image transport *topics*. These three elements incorporate with flexible framework known as ROS Core, which provides a set of tools and libraries via ROS *Master node* (a centralized XML-RPC server). It also enables communication between different parts electronics component (such as Lidar and IMU sensor, Pincher arm, RPi and ZED2 camera) and control manager of rover to perform autonomous navigation, grasping task and object detection.

Note: Testing procedure of hardware and software for attached devices are explained in appendix section 12.7.

4 Sensor integration

This chapter provides in-depth information about how Lidar sensor and ZED2 stereo camera are assembled with Leo Rover. The subsequent sections describe the ROS integration (necessary packages) with the rover and its attached electronics components. These packages offer base structure and enhance the rover's functionalities.

Sensor integration into rover is an important setup towards achieving autonomous navigations, objects detection and perception capabilities. To assemble the upgraded version of rover, little assistance is taken by USN engineers to ensure working conditions. The Figure 4-1 present a comparison of rover before and after modification. The left side shows the rover before the integration of Lidar Sensor and ZED2 camera along with additional computer, while the right side depicts the rover after assembling and integration. The Pincher arm has already been assembled; however, the arm connection and configuration are not integrated with rover's board on the first day when the rover is received.

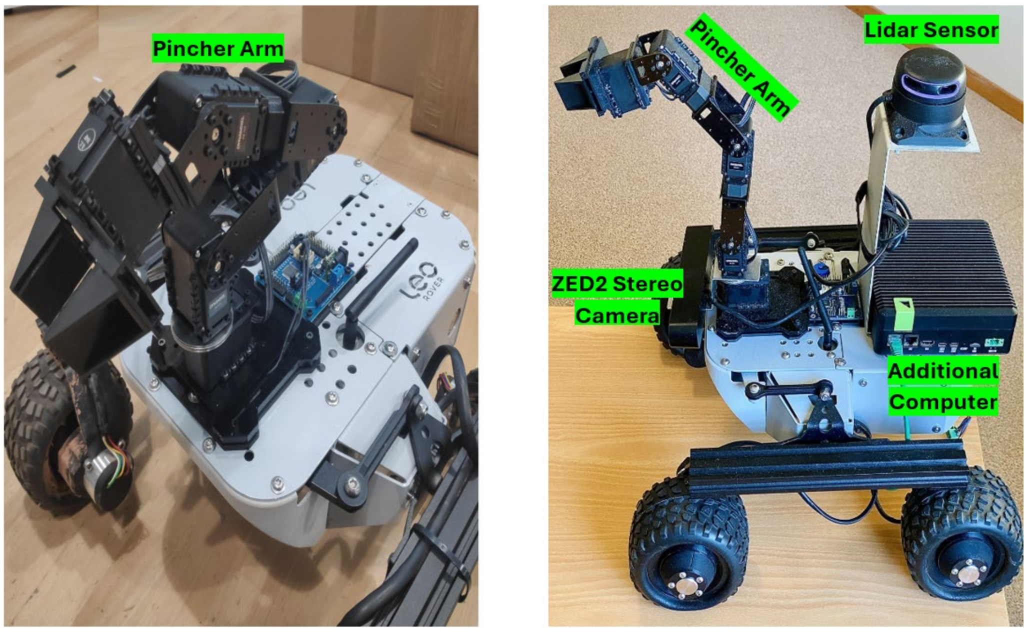


Figure 4-1:Comparing Leo Rover: Before and after assembling and integrating Lidar Sensor, ZED2 Camera, and Pincher Arm.

4.1 Assembly of Lidar sensor

Light detection and ranging is mechanisms used for generating a map for the environment, tracking the speed of vehicle, obstacle avoidance, robot navigation and in a variety of other applications. The Rplidar A2M12 is a two-dimensional 360-degree triangulation laser sensing scanner which is developed by Slamtech[21]. In the rover, Lidar sensor performs autonomous navigation using simultaneous localization and mapping technique.

The Figure 4-2 shows 3D printable adapter (developed by Fiction-lab) which allow to mount the Lidar sensor to the rover's mounting plate. A simple design (shown in Figure 4-3) using aluminum plate (4 mm) is prepared at USN workshop. It is positioned on the middle of rover's mounting plate which provides a wide field of view with minimal obstacle for the laser beam. The plate's dimensions are 21 cm in height and 9 cm in length for both the top and bottom sides. After preparing the design, it is attached to mounted plate of the rover, shown in Figure 4-3.

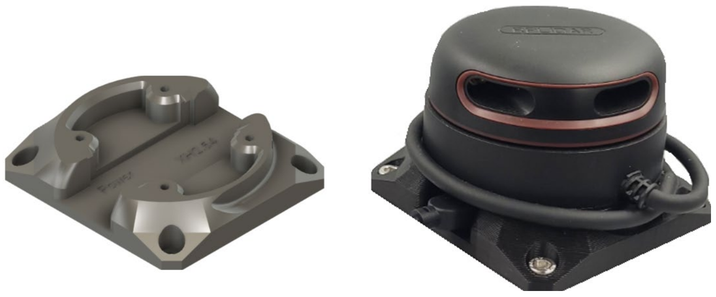


Figure 4-2: Rplidar A2 adapter [21].

A brushless motor in the board of Lidar sensor, the laser scanner operates smoothly which help to reduce the mechanical friction. By scanning its 360-degrees environment using a laser range scanner and generating an outline map, the device's core rotates in a clockwise direction. Furthermore, with its high rotation speed, it can take up to 8000 laser range samples per second, making it one of the fastest and most accurate Lidar scanner. With a range up to 12 meter, it performs two-dimensional 360-degree scanning operations and generates highly detailed point cloud data that can be used for mapping, localization, and navigation [21].

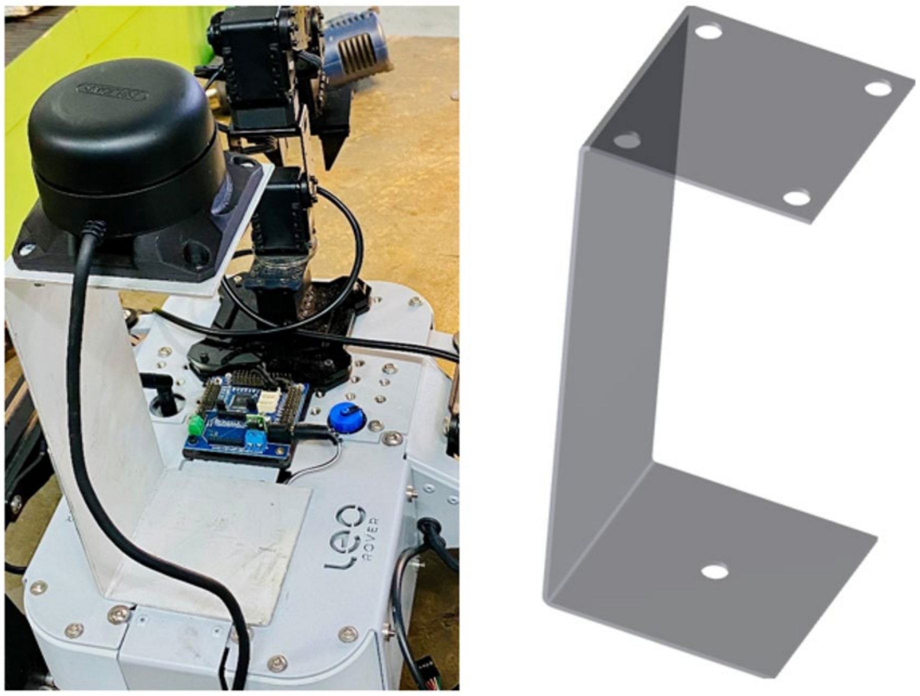


Figure 4-3: Aluminum frame for Lidar sensor attached with Leo Rover.

4.2 Assembly of ZED2 camera

The ZED2 stereo camera is capable of simultaneously capturing images from two lenses. As a result, this ZED2 can determine the distance between objects from the cameras based on the position [23]. This information can then be used in a variety of applications. For example, in Leo rover, stereo camera is used to detect AR-tags which based on small object (a small square box). To connect with rover, additional computer "Nvida Xaviour NX reComputer Industrial J2012". The reason to this additional computer is that the RPi board does not support CUDA operations and does not have capability to perform high level image processing.

Ethernet cable is used to connect additional computer to the Leo Rover, shown in Figure 4-4 and Figure 4-5, mounted at the top back side to the rover using mounting holes. ZED2 camera is attached with first servo motor of the Pincher arm using steel plate, shown in Figure 4-6. In addition, for supplying power to the additional computer, it is connected to rover's battery (power box) via electronic connector (shown in Figure 4-4), while the ZED2 camera is directly connected to additional computer via USB cable.

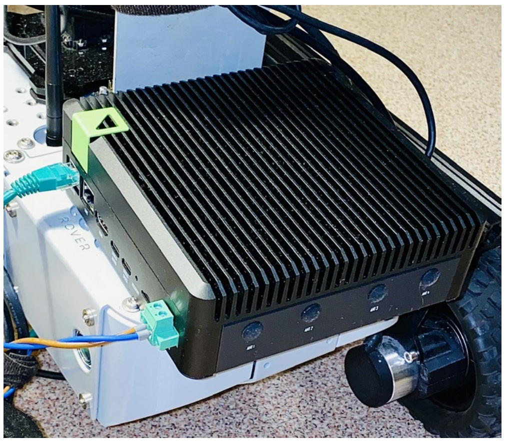


Figure 4-4: Addition computer connected with ethernet cable to the rover.

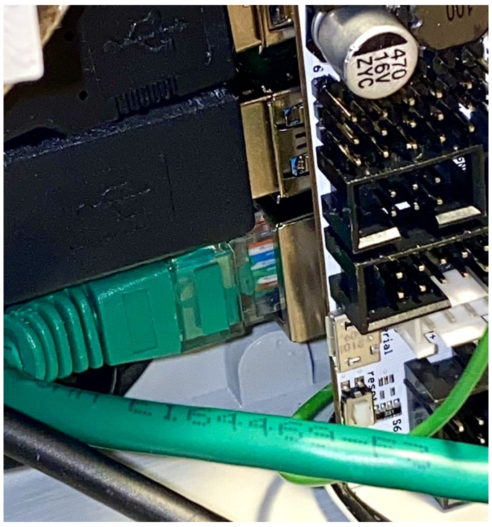


Figure 4-5: The same ethernet cable connected to the rover's RPi.

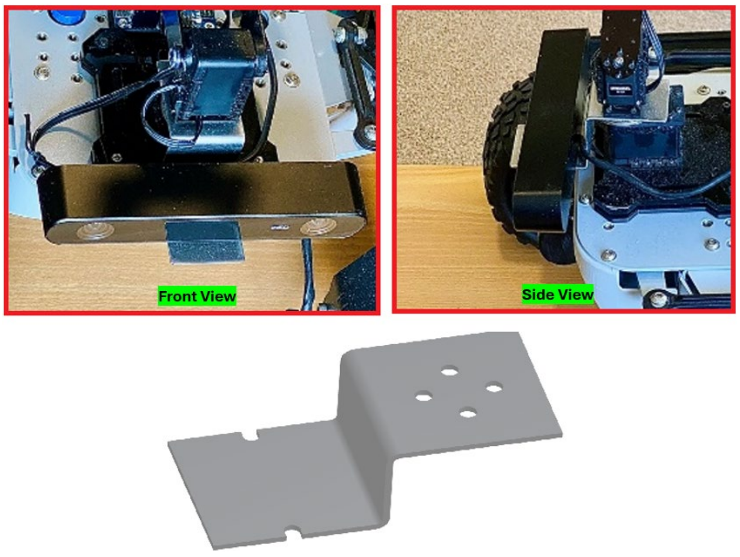


Figure 4-6: Design for ZED2 camera.

Note: The ZED2 design is a steel plate which is designed at USN workshop. The plant's dimension is 5 cm long from top and bottom side and have 3 cm bend. Small holes are punched to attached with servo mote and ZED2 camera.

4.3 Interfacing with ROS

ROS is a versatile framework for writing robot software. It contain several collection tools, libraries and conventions that aim to simplify the task of developing complex and robust robot behaviour, adaptable to various robotic platforms particularly available at – https://www.ros.org/ [24]. It allows to communicate with *nodes* running on Leo Rover and easier to interact ROS network within a robotic based environment. It also gives the possibility to write and run different processes (known as *rosnode*) which communicate with each other by sending and receiving messages through channels (known as *rostopic*) or by calling remote procedures (known as *rosservices*). In this project, the utilization of *LeoOS-1.2.0-2023-11-02-full.img.xz* (GitHub) is installed to facilitates ROS operation.

4.3.1 ROS commands

ROS provides some commands tools for inspecting the current networks of active *nodes* which are used to proceed Leo Rover sensors integrations. These tools include [25]:

1. *rosnode:* It is responsible for printing information about currently running available *nodes*, terminate them and testing its connectivity.
2. *rostopic:* It is responsible for listing and printing available *topics* information which are currently be used. It also publishes the *messages* to topics and find a type of published *messages*.
3. *rosservice:* It is responsible for listing and printing information of available *services* and call the *services* with specified arguments.
4. *rosmsg:* It displays the type of *messages*.

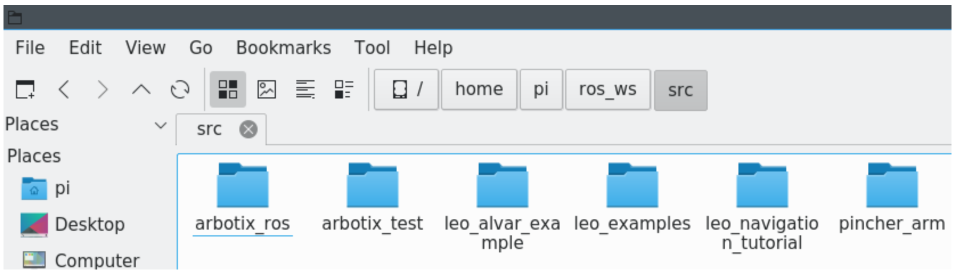
4.3.2 ROS packages

To achieve the task requirements of master thesis project, various ROS packages are utilized to develop and implement Leo Rover functionalities. These packages *include leo_description, leo_bringup, leo_viz, rplidar_ros, leo_navigation_tutorial, leo_examples (for ARTags), leo_alvar_example, arbotix_arm, arbotix_test, pincher_arm, and zed_wrapper.* These packages collectively contribute to the development of a comprehensive robotic system [11]. The installation process and corresponding GitHub links are provided explained in 12.5.7 Building necessary ROS packages.

- *leo_description:* It provides a detailed description of the rover's physical characteristics.
- *leo_bringup:* It initializes various *nodes* and parameters which are essential for bringing up the Leo robot system, i.e. servo motors, *serial nodes* (ports) and web video camera.
- *leo_viz:* It offers visualization tools including Rviz software for debugging and analysis.
- *leo_navigation_tutorial:* It uses many other available packages such as *robot_localization*, *move_base* and *twist_mux* in ROS to provide autonomous capabilities (discussed in 5.1). It facilitates the integration of navigation capabilities into the rover's control system for SLAM implementation.
- *leo_examples* and *leo_alvar_example*: These two packages are used for implementing marker-based localization using AR-tags and Avar libraries, respectively, (discussed in 6.2).
- *arbotix_arm, arbotix_test* and *pincher_arm:* These three packages enable control and manipulation of robotic arm using *MoveIt* (discussed in 7.3).
- *zed_wrapper:* It interfaces with the ZED stereo camera for perception tasks (discussed in appendix section 4.3.7).

4.3.3 ROS workspace

All packages are built using the *Catkin* build system, which compiles and manages dependencies, ensuring that software components interact seamlessly. It provides an overlay mechanism, allowing one workspace to extend another result space [25]. It also offers a *catkin_make* commands for building empty workspace. Having this advantage, ROS workspace *ros_ws* is created inside home directory on RPi operating system which perform Leo Rover functions. All available packages are listed in this directory as shown in Figure 4-7 and their installation process is discussed in appendix section 12.5.



arbotix_ros
arbotix_test
leo_alvar_exa mple
leo_examples
leo_navigatio n_tutorial
pincher_arm

Figure 4-7: List of installed packages at ROS workspace *ros_ws* .

In addition to *LeoOS*, ROS packages need to integrate with its functionalities which provides an easy mechanism for adding new functionalities without building additional package. The whole process of starting to depend on the following files [26]:

- */etc/ros/**robot.launch**:* It is a launch file that starts the robot's functionality. A launch file is an XML file that outlines a collection of nodes to be initiated along with specific parameters. It incorporates the launch files from the *"leo_bringup"* package which is responsible for initiating the base of the rover. For example, Lidar sensor, Pincher arm, RPi camera and Alvar packages are assigned in this file to run their ROS *nodes* simultaneously. The file script is attached in appendix section 12.5.7
- */etc/ros/**setup.bash**:* It is an environment setup file which initialize ROS *nodes* successfully. It sources the setup file from the designated ROS distribution typically located at *"/opt/ros/noetic/setup.bash"* and configure additional environment variable for ROS operations.
- */usr/bin/**leo-start**:* It is a script file which start the Leo Rover's functionality and source the */etc/ros/**setup.file*** and launches the */etc/ros/**robot.launch*** files.
- */usr/bin/**leo-stop**:* It is another script file which stops the currently running ***leo-start*** process from Leo Rover.

4.3.4 Establishing a network connection

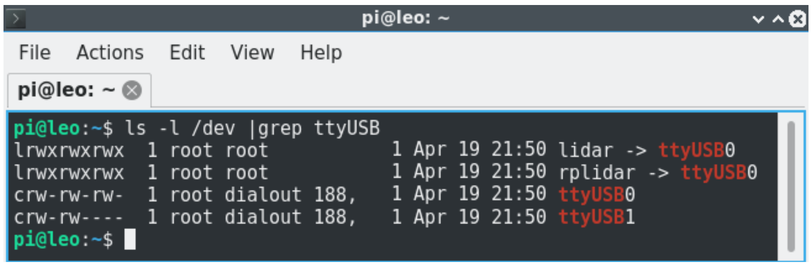
To establish a connection with the Leo Rover, connecting process begin by activating it using the main power button situated on the left side of the battery. Upon activation, the LED on the button starts blinking green, indicating the startup process. After approximately, 30 seconds, the LED stop blinking, signalling that the Leo Rover is now operationally ready for interaction.

Subsequently, the host device for Wi-Fi networks LeoRover-XXXX is available to connect the with the rover, where XXXX represent a unique identifier for rover's computer which has contain password.

After establishing a connection with the rover, it required to control the rover's interface. For remote desktop Remmina Remote Desktop Client software is installed on RPi operating systems to control rover's session. The whole process along with screenshots is discussed in appendix sections 12.5.3, 12.5.4, 12.5.5 and 12.5.6.

4.3.5 Rplidar A2M12 Lidar sensor

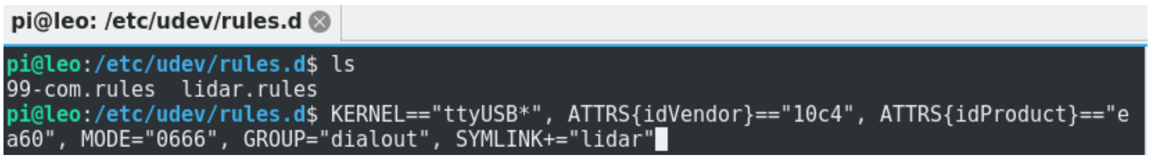
The first thing is needed to make sure that the device has the correct permission and is available at the fixed path on operating system. As it can be seen in Figure 4-8, there are two USBs (ttyUSB0 and ttyUSB1) connected to the RPi of Leo Rover. The Lidar sensor is connected to ttyUSB0, while robotic arm is connected to ttyUSB1.



```
pi@leo:~$ ls -l /dev |grep ttyUSB
lrwxrwxrwx  1 root root           1 Apr 19 21:50 lidar -> ttyUSB0
lrwxrwxrwx  1 root root           1 Apr 19 21:50 rplidar -> ttyUSB0
crw-rw-rw-  1 root dialout 188,   1 Apr 19 21:50 ttyUSB0
crw-rw----  1 root dialout 188,   1 Apr 19 21:50 ttyUSB1
pi@leo:~$
```

Figure 4-8: Available devices attached with RPi of Leo Rover.

Some specified rules are used for enabling Lidar sensor functionalities, After using the following command on RPi terminal, shown in Figure 4-9, the ***lidar.rules*** is appended to the directory path */etc/udev/**rules.d*** . This action ensures the availability of the sensor at the designated path ***/dev/lidar***.

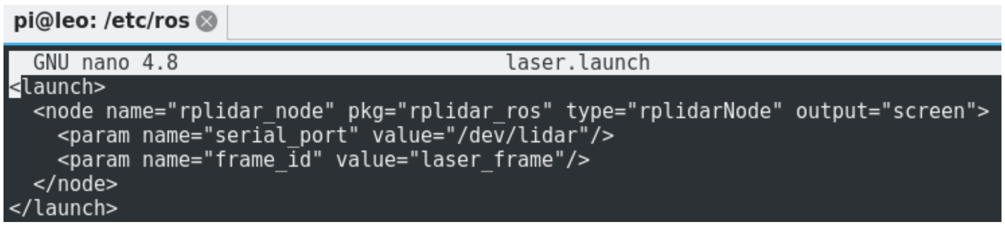


```
pi@leo:/etc/udev/rules.d$ ls
99-com.rules  lidar.rules
pi@leo:/etc/udev/rules.d$ KERNEL=="ttyUSB*", ATTRS{idVendor}=="10c4", ATTRS{idProduct}=="e
a60", MODE="0666", GROUP="dialout", SYMLINK+="lidar"
```

Figure 4-9: Identifying Lidar sensor rules [21].

To ensure the sensor's functionality is accessible with the ROS ecosystem, *rplidar-ros* package (GitHub) is installed to start the Rplidar A2M12 Lidar sensor. This is achieved by executing the following command *"sudo apt install ros-noetic-rplidar-ros"* on RPi terminal.

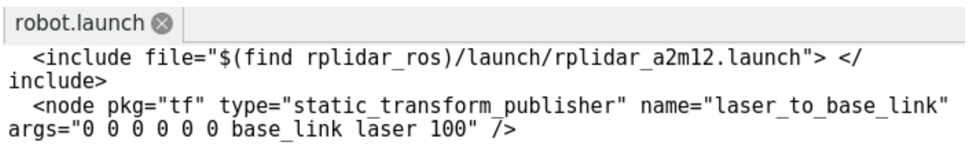
After identifying the Lidar rules and installing its package, additional launch file is added with following script to the path */etc/ros/**laser.launch,*** illustrated in Figure 4-10. This script is used to launch Lidar ROS *nodes*, which is responsible for interfacing with the sensor.



```
<launch>
  <node name="rplidar_node" pkg="rplidar_ros" type="rplidarNode" output="screen">
    <param name="serial_port" value="/dev/lidar"/>
    <param name="frame_id" value="laser_frame"/>
  </node>
</launch>
```

Figure 4-10: Creating a lunch file to start ROS *node* for Lidar sensor [21].

The following script shown in Figure 4-11, is added to the path */etc/ros/**robot.launch*** where the rover starts from boot to run related *rosnode* for Lidar sensor when ***leo-start*** command started.



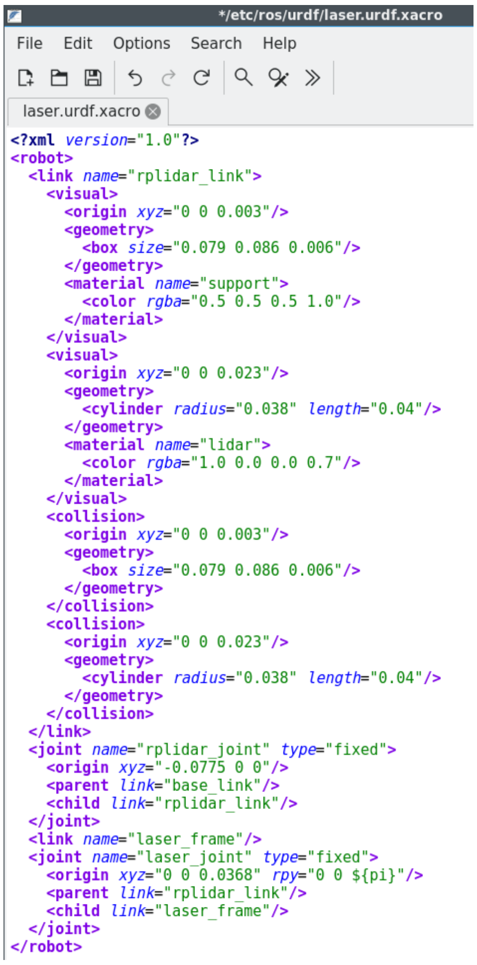
```
  <include file="$(find rplidar_ros)/launch/rplidar_a2m12.launch"> </
include>
  <node pkg="tf" type="static_transform_publisher" name="laser_to_base_link"
args="0 0 0 0 0 0 base_link laser 100" />
```

Figure 4-11: Including a Lidar sensor launch file into Leo Rover model.

It is important that the rover need to be aware, where the sensor is located and what space it occupies. To make it sure, following script is added to */etc/ros/urdf/**laser.urdf.xacro*** file with the path, illustrated in Figure 4-12. It is XML file that describe the Lidar sensor model using the Unified Robot Description Format (URDF) representation of lidar sensor in simulation. The URDF is commonly used in robotic to represent the robot geometries and physical properties [21]. It is important to note that necessary changes are made within the file, as the sensor is located at the top middle side of the rover.

The ***laser.urdf.xacro*** file composed with two links: rplidar_link and laser_frame. Within the file, <robot> tag encloses the entire sensor description. The first link represents the physical body of sensor including visual and collision properties, while the second link represents an abstract form associated with the laser scanner. However, the second link does not have any visual and collision properties.

The rplidar_joint is joint which connect the rplidar_link to the base_link at fix position. As it shown in Figure 4-12, Laser sensor is mounted at *"xyz"* coordinates with of (-0.0075, 0, 0) meters. Similarly, the laser_joint connects the rplidar_link to the laser_frame at fix position (like rplidar_joint). After making these changes, the Lidar sensor is attached to rover along with its robotic, shown in Figure 4-13. The rover model looks exactly same in simulation and real form.



```xml
<?xml version="1.0"?>
<robot>
  <link name="rplidar_link">
    <visual>
      <origin xyz="0 0 0.003"/>
      <geometry>
        <box size="0.079 0.086 0.006"/>
      </geometry>
      <material name="support">
        <color rgba="0.5 0.5 0.5 1.0"/>
      </material>
    </visual>
    <visual>
      <origin xyz="0 0 0.023"/>
      <geometry>
        <cylinder radius="0.038" length="0.04"/>
      </geometry>
      <material name="lidar">
        <color rgba="1.0 0.0 0.0 0.7"/>
      </material>
    </visual>
    <collision>
      <origin xyz="0 0 0.003"/>
      <geometry>
        <box size="0.079 0.086 0.006"/>
      </geometry>
    </collision>
    <collision>
      <origin xyz="0 0 0.023"/>
      <geometry>
        <cylinder radius="0.038" length="0.04"/>
      </geometry>
    </collision>
  </link>
  <joint name="rplidar_joint" type="fixed">
    <origin xyz="-0.0775 0 0"/>
    <parent link="base_link"/>
    <child link="rplidar_link"/>
  </joint>
  <link name="laser_frame"/>
  <joint name="laser_joint" type="fixed">
    <origin xyz="0 0 0.0368" rpy="0 0 ${pi}"/>
    <parent link="rplidar_link"/>
    <child link="laser_frame"/>
  </joint>
</robot>
```

Figure 4-12: Creating a URDF ***/laser.urdf.xacro*** file for Lidar sensor [21].

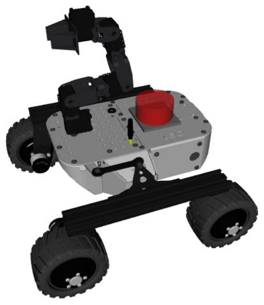
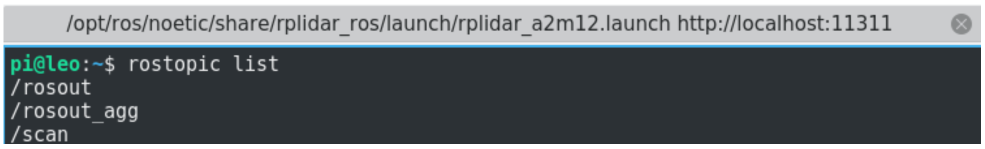


Figure 4-13: Position of Lidar sensor and robotic arm at Leo Rover.

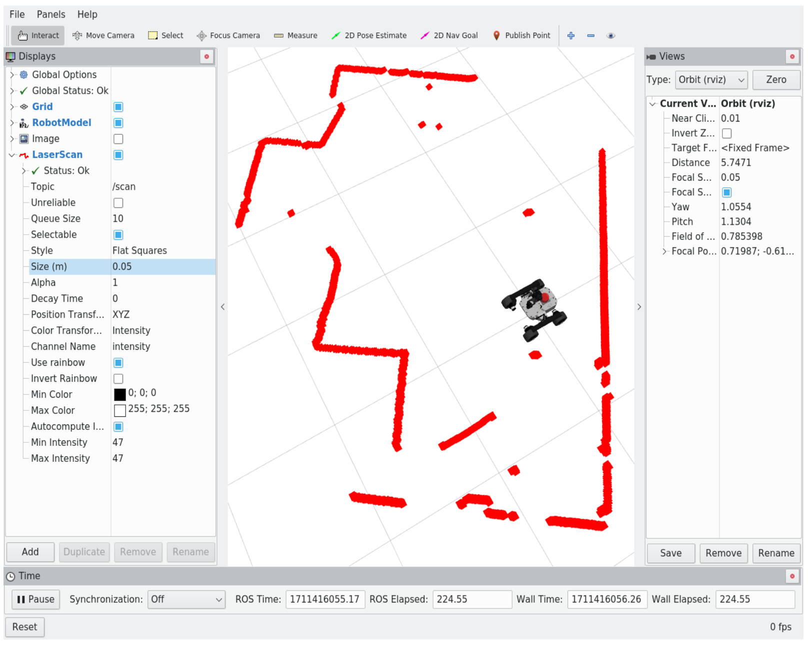
The robot should now publish the LaserScan message on the /scan *topic* as it available in *ROS topic list*, shown in Figure 4-14. It also starts rotating the Laser scanner when the rover is turn ON.



```
/opt/ros/noetic/share/rplidar_ros/launch/rplidar_a2m12.launch http://localhost:11311
pi@leo:~$ rostopic list
/rosout
/rosout_agg
/scan
```

Figure 4-14:ROS *topics list* for Lidar sensor after integration.

For the graphical representation of the rover and attached Lidar sensor, Rviz tools is utilized through *leo_viz* package, as depicted in Figure 4-15. It illustrates the RobotModel (Leo Rover) positioned at a specific location and visualizes the LaserScan using */scan topic*. The Laser data is shown with red dotted lines, each representing a scan point, with a size of 0.05 meters.



Displays

Global Options	
Global Status: Ok	
Grid	☑
RobotModel	☑
Image	☐
LaserScan	☑
Status: Ok	
Topic	/scan
Unreliable	☐
Queue Size	10
Selectable	☑
Style	Flat Squares
Size (m)	0.05
Alpha	1
Decay Time	0
Position Transf...	XYZ
Color Transfor...	Intensity
Channel Name	intensity
Use rainbow	☑
Invert Rainbow	☐
Min Color	■ 0; 0; 0
Max Color	□ 255; 255; 255
Autocompute I...	☑
Min Intensity	47
Max Intensity	47

Views

Type: Orbit (rviz) Zero

Current V...	Orbit (rviz)
Near Cli...	0.01
Invert Z...	☐
Target F...	<Fixed Frame>
Distance	5.7471
Focal S...	0.05
Focal S...	☑
Yaw	1.0554
Pitch	1.1304
Field of ...	0.785398
Focal Po...	0.71987; -0.61...

Time

Synchronization: Off ROS Time: 1711416055.17 ROS Elapsed: 224.55 Wall Time: 1711416056.26 Wall Elapsed: 224.55

0 fps

Figure 4-15: LaserScan data */scan topic* visualization in Rviz.

Note: On the left side of the figure, there is a tab named "Displays" which shows all available ROS *topic* while the Rplidar A2M12 Lidar sensor is running.

4.3.6 Phantom-X Pincher arm

Before installing *pincher_arm* ROS package, driver for Arbotix-M controller board is derived via arbotix_package (GitHub) in order to configure Dynamixel servo motors Ids. The FTDI-USB cable is connected to RPi computer make a proper communication channel with Arbotix controller to control the motors. In addition, 12 V SMPS (from power box) is supplied to the controller to start the controller. There are five server motors (AX-12A) in Pincher arm, which are consecutively connected. However, before connecting these motors together, all servo motors are individually tested using the method, shown in Figure 4-16, and specified them into five Ids from 1 till 5, shown in Figure 4-17.

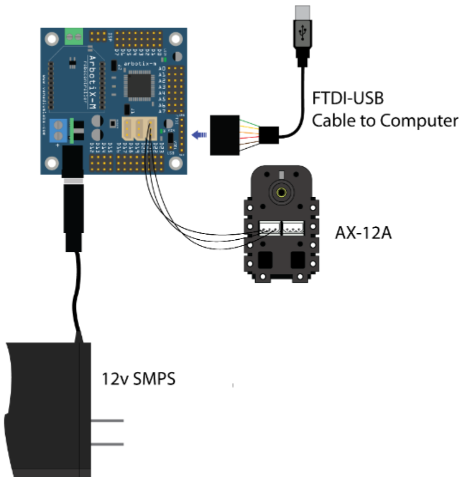
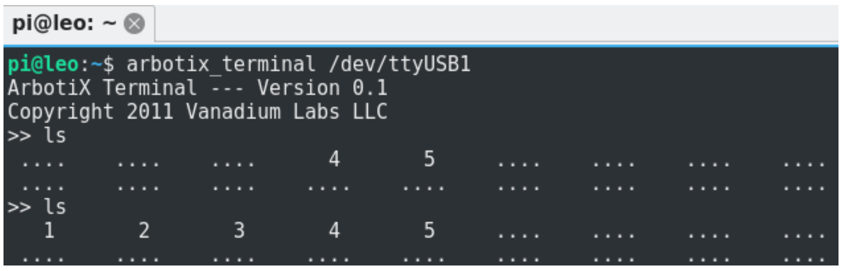


Figure 4-16: Testing individual servos motors [14].

To test the servo motors, *arbotix_arm* and *arobitx_test* packages are installed in ROS workspace *"ros_ws", shown in* Figure 4-7. As it can been in Figure 4-17, after assigning servo motors IDs, using command *"arbotix_termnial /dev/ttyUSB1"* on Rpi terminal, the available five motors are listed. The package installation process is discussed in appendix section 12.5.7.

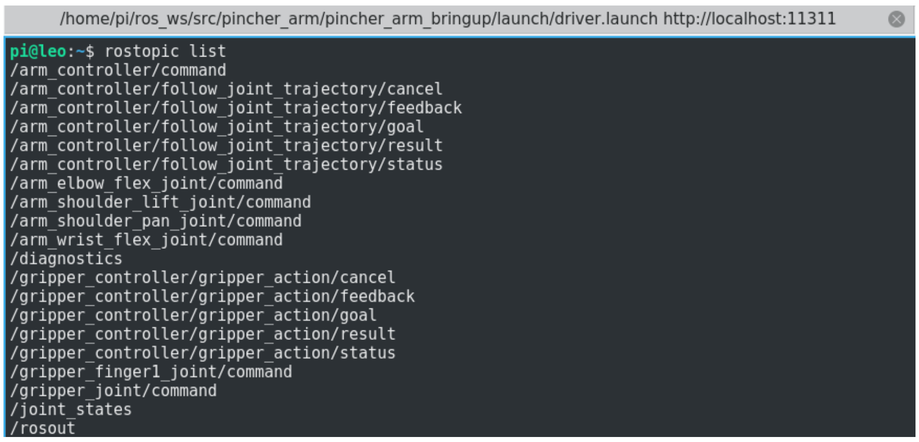


```
pi@leo: ~
pi@leo:~$ arbotix_terminal /dev/ttyUSB1
ArbotiX Terminal --- Version 0.1
Copyright 2011 Vanadium Labs LLC
>> ls
 ....    ....    ....      4       5    ....    ....    ....    ....
 ....    ....    ....    ....    ....    ....    ....    ....    ....
>> ls
   1       2       3       4       5    ....    ....    ....    ....
 ....    ....    ....    ....    ....    ....    ....    ....    ....
```

Figure 4-17: Testing the servo motors of robotic arm.

After testing all the servo motors, *pincher_arm* package is installed into the rover workspace *ros_ws* to utilized Pincher arm functionalities. It also include *Move_it* configuration, *pick and place* demo, *IKFast Kinematic*, *URDF* model integration with rover.

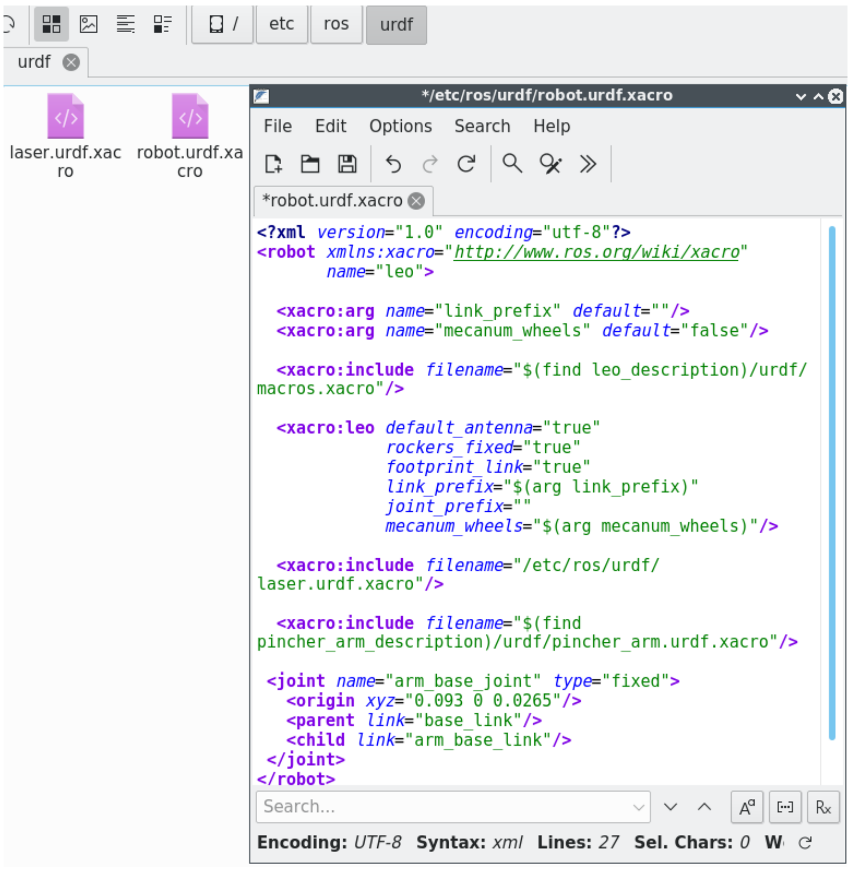
To test the arm, its package *"pincher_arm_bringup"* is launched to verify the ROS *topics* (shown in Figure 4-18) for performing further tasks.



```
/home/pi/ros_ws/src/pincher_arm/pincher_arm_bringup/launch/driver.launch http://localhost:11311
pi@leo:~$ rostopic list
/arm_controller/command
/arm_controller/follow_joint_trajectory/cancel
/arm_controller/follow_joint_trajectory/feedback
/arm_controller/follow_joint_trajectory/goal
/arm_controller/follow_joint_trajectory/result
/arm_controller/follow_joint_trajectory/status
/arm_elbow_flex_joint/command
/arm_shoulder_lift_joint/command
/arm_shoulder_pan_joint/command
/arm_wrist_flex_joint/command
/diagnostics
/gripper_controller/gripper_action/cancel
/gripper_controller/gripper_action/feedback
/gripper_controller/gripper_action/goal
/gripper_controller/gripper_action/result
/gripper_controller/gripper_action/status
/gripper_finger1_joint/command
/gripper_joint/command
/joint_states
/rosout
```

Figure 4-18: ROS *topic* list for robotic arm.

In the final stage, following script in ***robot.urdf.xacro*** (shown in Figure 4-19) is added to is added ***robot.launch*** files to the path *etc/ros/urdf/ and /etc/ros/* respectively. It is XML file that describe rover model using URDF in simulation. Within the file, Pincher arm URDF model is added along with Leo Rover model and Lidar sensor, where the arm_base_joint joint connect the arm_base_link (Pincher arm) to base_link (Leo Rover). The arm is mounted at "xyz" coordinates (0.0093, 0, 0.265) meters.

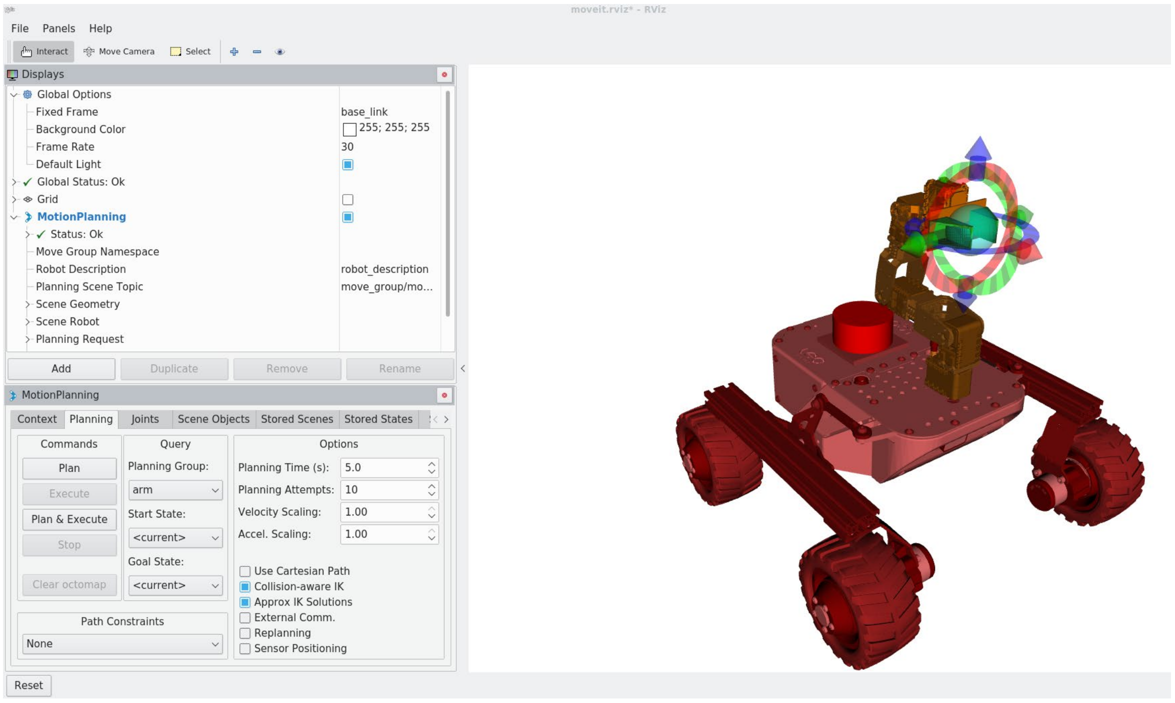


```
<?xml version="1.0" encoding="utf-8"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro"
       name="leo">

  <xacro:arg name="link_prefix" default=""/>
  <xacro:arg name="mecanum_wheels" default="false"/>

  <xacro:include filename="$(find leo_description)/urdf/
macros.xacro"/>

  <xacro:leo default_antenna="true"
             rockers_fixed="true"
             footprint_link="true"
             link_prefix="$(arg link_prefix)"
             joint_prefix=""
             mecanum_wheels="$(arg mecanum_wheels)"/>

  <xacro:include filename="/etc/ros/urdf/
laser.urdf.xacro"/>

  <xacro:include filename="$(find
pincher_arm_description)/urdf/pincher_arm.urdf.xacro"/>

 <joint name="arm_base_joint" type="fixed">
   <origin xyz="0.093 0 0.0265"/>
   <parent link="base_link"/>
   <child link="arm_base_link"/>
 </joint>
</robot>
```

Encoding: *UTF-8* Syntax: *xml* Lines: *27* Sel. Chars: *0* W

Figure 4-19: Robot's URDF description for Leo Rover including Lidar sensor and robotic arm.

To visualize the arm position at the rover, it can visualize in Rviz simulation tool, shown in Figure 4-20. It is situated at the same position which is defined in the ***robot.urdf.xacro*** file. After launching the *pincher_arm_bringup* launch file form it ROS package, *MotionPlanning* with *Move_it* configuration is tested and ready to perform grasping activities.



```
Global Options
  Fixed Frame            base_link
  Background Color       255; 255; 255
  Frame Rate             30
  Default Light          ✓
Global Status: Ok
Grid                     ☐
MotionPlanning           ✓
  Status: Ok
  Move Group Namespace
  Robot Description      robot_description
  Planning Scene Topic   move_group/mo...
  Scene Geometry
  Scene Robot
  Planning Request

Planning Group: arm
Start State: <current>
Goal State: <current>
Path Constraints: None
Planning Time (s): 5.0
Planning Attempts: 10
Velocity Scaling: 1.00
Accel. Scaling: 1.00
☐ Use Cartesian Path
☑ Collision-aware IK
☑ Approx IK Solutions
☐ External Comm.
☐ Replanning
☐ Sensor Positioning
```

Figure 4-20: Testing *Move_it* configuration in RViz.

Note: On the left side of the figure, there is a tabs named “Displays” and “MotionPlanning”. The Display tab shows all available ROS *topic* while the robotic arm is running. In the MotionPlanning tab, there are several other tabs which are responsible to test the arm’s functionalities.

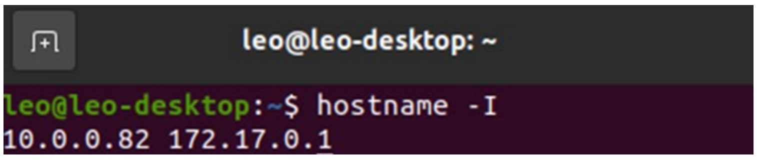
4.3.7 ZED2 stereo camera

The ZED2 stereo camera, developed by Stereolabs which offers stereo vision capabilities and depth sensing, makes it an ideal choice for increasing the rover’s perception abilities.

The initial step in the ROS integration process involved installing the ZED SDK on an additional computer. The primary purpose of SDK is to enable depth sensing capabilities. It allows to understand the 3D structure of the environment which is important to achieve related tasks such as object detection, augmented reality, 3D mapping for robotics [23].

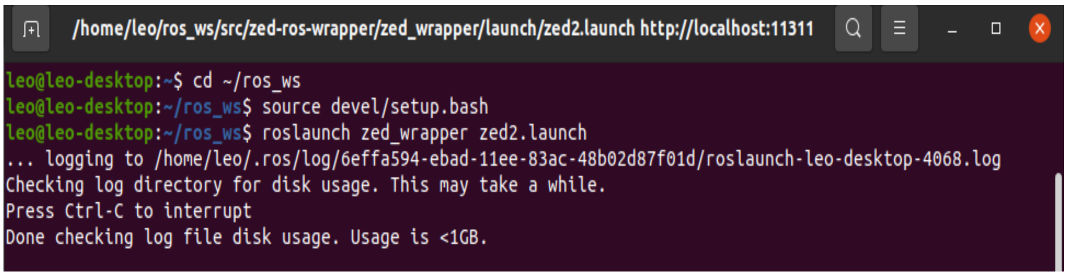
To interface the camera with ROS, the ZED ROS *wrapper_package* (GitHub) is utilized to facilitates the camera functionalities such as left and right rectified/unrectified images, depth maps, object detections and among others. Subsequently, all required dependencies are installed, and the ROS package is built properly on additional computer.

Prior to running the *nodes*, environment variables specifying the ROS *master*, the IP address (shown in Figure 4-21) of additional computer (10.0.0.82) is exported into the rover’s ROS network. Moreover, the additional computer is connected to the same network as the rover's network with access point *(LeoRover-18d4)*. This process ensured seamless communication with the ROS *master note* and connected devices. After launching the *zed2.launch* launch file from its *zed_wrapper* package, shown in Figure 4-22, the ROS *nodes* and *topic* list for ZED2 stereo camera are available for further implementations, shown in Figure 4-23.



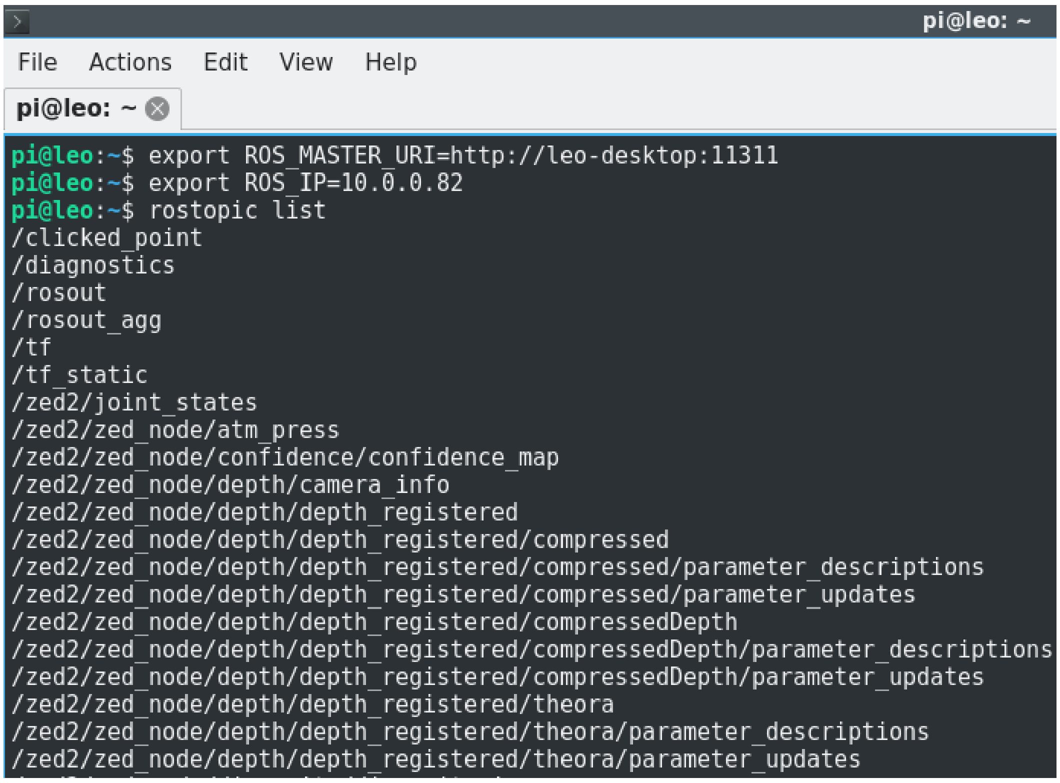
```
leo@leo-desktop:~$ hostname -I
10.0.0.82 172.17.0.1
```

Figure 4-21: IP address of additional computer (Xavior).


/home/leo/ros_ws/src/zed-ros-wrapper/zed_wrapper/launch/zed2.launch http://localhost:11311

```
leo@leo-desktop:~$ cd ~/ros_ws
leo@leo-desktop:~/ros_ws$ source devel/setup.bash
leo@leo-desktop:~/ros_ws$ roslaunch zed_wrapper zed2.launch
... logging to /home/leo/.ros/log/6effa594-ebad-11ee-83ac-48b02d87f01d/roslaunch-leo-desktop-4068.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
```

Figure 4-22: Launching ZED2 Camera on additional computer.



```
pi@leo:~$ export ROS_MASTER_URI=http://leo-desktop:11311
pi@leo:~$ export ROS_IP=10.0.0.82
pi@leo:~$ rostopic list
/clicked_point
/diagnostics
/rosout
/rosout_agg
/tf
/tf_static
/zed2/joint_states
/zed2/zed_node/atm_press
/zed2/zed_node/confidence/confidence_map
/zed2/zed_node/depth/camera_info
/zed2/zed_node/depth/depth_registered
/zed2/zed_node/depth/depth_registered/compressed
/zed2/zed_node/depth/depth_registered/compressed/parameter_descriptions
/zed2/zed_node/depth/depth_registered/compressed/parameter_updates
/zed2/zed_node/depth/depth_registered/compressedDepth
/zed2/zed_node/depth/depth_registered/compressedDepth/parameter_descriptions
/zed2/zed_node/depth/depth_registered/compressedDepth/parameter_updates
/zed2/zed_node/depth/depth_registered/theora
/zed2/zed_node/depth/depth_registered/theora/parameter_descriptions
/zed2/zed_node/depth/depth_registered/theora/parameter_updates
```

Figure 4-23: ZED2 Camera ROS node lists on Leo Rover.

Note: The appendix section 12.5.6 discussed how to connect ROS network to additional computer or laptop.

5 Autonomous navigation

This chapter describes the testing results of SLAM in both simulated and physical environment and highlights the rover's performance in given scanned map after utilizing *leo_navigation* ROS package.

SLAM is a fascinating technique which is utilized using Lidar sensor and it allows to generate a map of its environment while simultaneously determining its own position into the rover [21]. In this process, two sensors (Lisar and IMU sensors) work together to perform autonomous navigation, simultaneous localization and mapping.

5.1 Integration of *leo_navigation* package

The *leo_navigation* package (GitHub) is built within ROS workstation *ros_ws* of RPi board by installing all dependencies. The installation process is discussed in appendix section 12.5.7. The package consists of three main parts which describe its functionalities, including Odometry, SLAM and Navigation.

Odometry is necessary for an autonomous navigation system to estimate the exact position of the rover, which is done by using wheel-encoder and IMU sensor. After estimating the rover's position, SLAM uses both Lidar and IMU data measurement along with wheel-encoder to produce an occupied map of the terrain. While doing so, the estimated position needs to be corrected based on the loop closure detections. Once the whole terrain is mapped, the map is saved to a file for later use to track rover's pose against it. With an accurate enough pose estimation and occupied map, autonomous navigation can be performed.

When a navigation target is set on an occupied map, a path planning algorithm (A*) tries to find collision-free path to reach the desired goal. After that another algorithm (Trajectory tracking algorithm) sends velocity commands to the rover. It receives information about the desired path from the path planning algorithm (A*), as well as feedback from sensors to determine the rover's pose (current position and orientation). Based on this information, it calculates the appropriate velocity commands (linear and angular velocities) to steer the rover along the desired path while also avoiding obstacles detected in the environment.

SSN housing room is mapped to test SLAM's performance. Both Figure 5-1 and Figure 5-2 show the Rviz simulation results, where three launch files are launched including *odometry.launch, gmapping.launch* and *navigation.launch* together for utilizing the autonomous navigation and SLAM. The rover is placed at one position (Figure 5-1) on scanned map of the terrain and after providing a target goal using "2D Nav Goal" from the toolbar located at the top, the rover localize itself within the scanned map to drive autonomously to designated position successfully. As it can been seen in Figure 5-2, the rover is reached to targeted position where the green line shows the path which follows by path planning algorithm after being given navigation goal. In the left side of figure, The Displays tab show all available ROS *topics* such as RobotModel, Image, Odometry, Map, LaserScan, Path and Global Costmap are activated. In the camera, it can be visualized the position of rover is changed after the target is achieved.

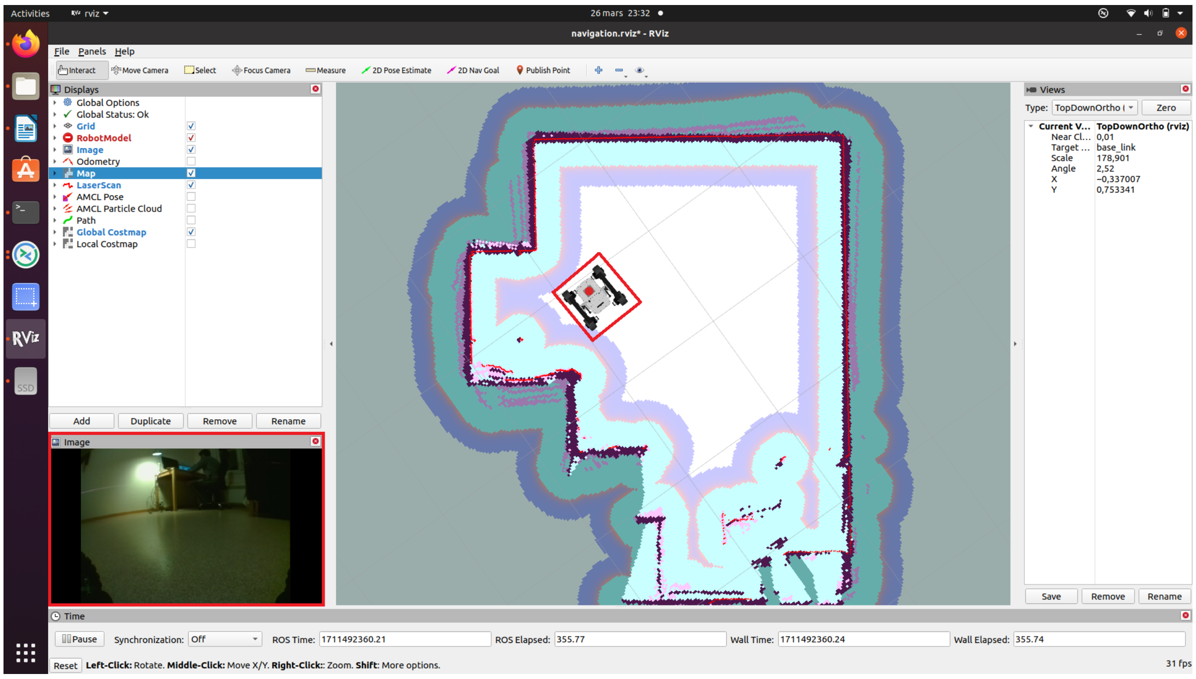


Figure 5-1: Testing SLAM (Part-1).

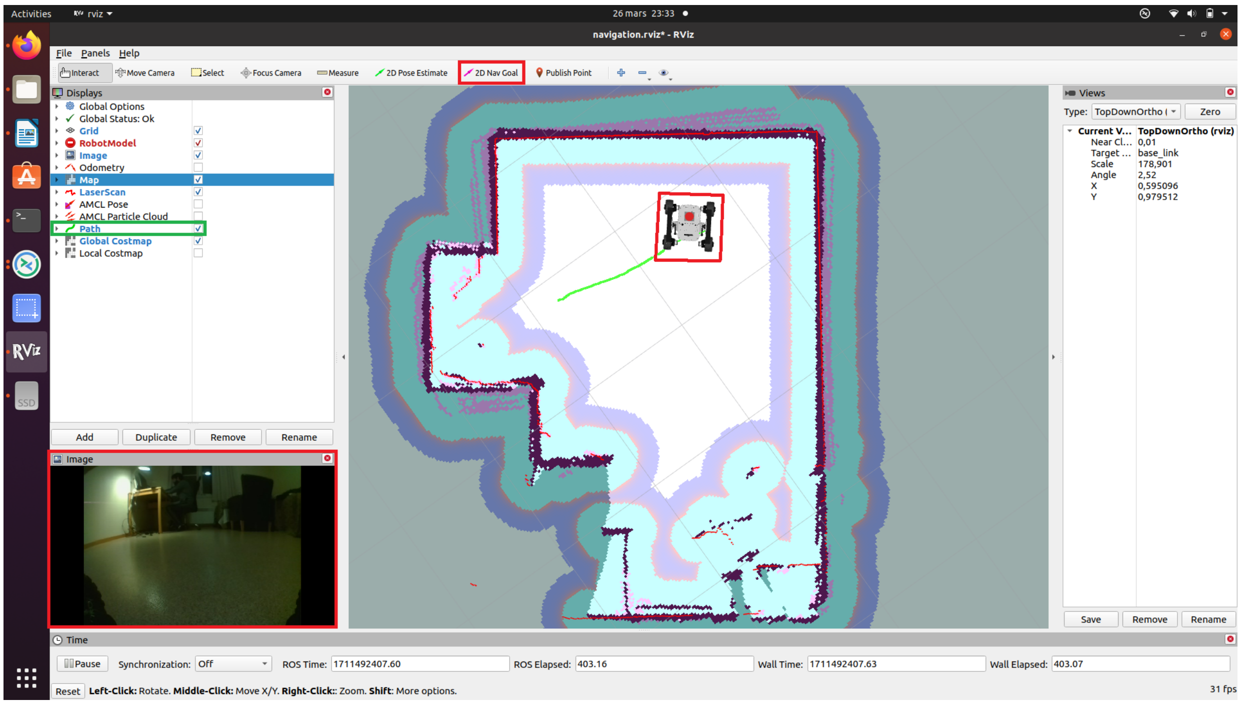


Figure 5-2: Testing SLAM (Part-2).

Note: On the left-side of the figure, available ROS *topics* are activated.

The Figure 5-3 shows a diagram generated by *rqt_graph* while *odometry, gmapping* and *navigation* launch files are actively running. The diagram illustrates *ROS nodes* and *topics* launched by A2M12 Lidar sensor launch rplidar_*a2m12.lauch* file.

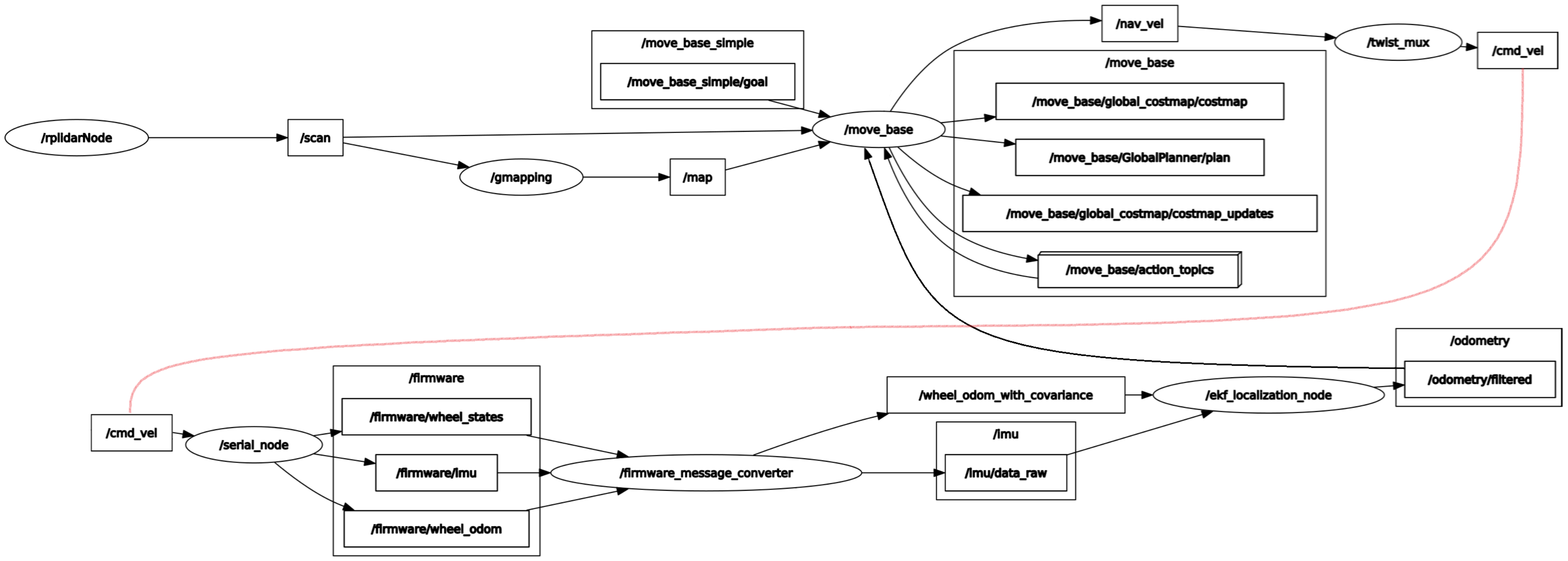


Figure 5-3: ROS nodes and topics launched by *rplidar_a2m12.launch* file.

Note: The oval shape represents the ROS *nodes*, while the rectangles with solid lines represent the ROS *topics*. The arrows between the oval and rectangles shapes represent the Publish/Subscribe protocols. The red line indicates the continuous section begin from ROS *topic /cmd_vel.* The bigger rectangle box indicates the groups.

These three sections 5.1.1, 5.1.2 and 5.1.3 describe how *ROS nodes* and *topics* of Lidar sensor working together along with the rover.

5.1.1 Odometry

To estimate the Leo Rover's position from the wheel encoders and IMU sensor measurements, additional robot_localization package (Link) is utilized, contain two state estimation nodes: the *ekf_localization_node* which implements Extended Kalaman Filter and the *ukf_localization_node* which implements Unscented Kalman Filter. In the localization package*, efk_localization_node* is preferred [21] because it has less computationally expensive than other one. In this graph, the state estimation node requires input topics (under */firmware* section), stamped with ROS messages which contain covariance matrices. That is the reason *leo_navigation* package provide */firmware_message_converter* node which include:

- For each *message* on */firmware/wheel_odom topic*, publishes a *message* on the */wheel_odom_with_covariance topic* which contains the same data but with an extension of covariance matrix.
- For each pair of */firmware/Imu* and */firmware/wheel_states topics*, publish a *message* on */Imu/data_raw topic* which combine the two *messages* with an extension of covariance matrices.
- Apart from these *topics*, the *ekf_localization_node* publish the data to */odometry/filter topic* and subscribe the previous both *topics* as common reference frame.

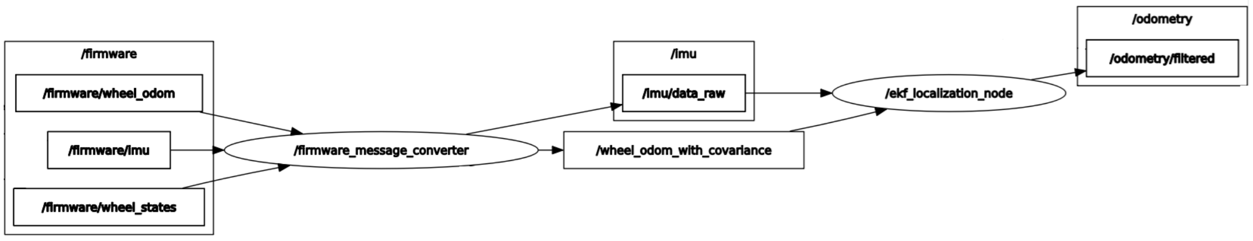


Figure 5-4: ROS *nodes* and *topics* launched by the *odometry.launch* file.

5.1.2 SLAM

There is one issue found related to SLAM which lies in the challenge of constructing an accurate map of the environment while simultaneously determining the rover's position within given circumstances [26]. To solve this issue, GMapping approach is used that utilizes range data from Lidar sensor (A2M12) and local odometry source. It consists of algorithms that has its own ROS wrapper node in the gmapping package (Link). After loading this package, rover is capable of generating maps, as illustrated in Figure 5-1 and Figure 5-2. The Figure 5-5 shows the diagram for the *gmapping.launch* launch file. This diagram is straightforward, illustrating that the */gmapping* node receive input from A2M12 Lidar sensor (*/scan topic*) and produces an occupied grip map of the terrain as an output (*/map topic*).

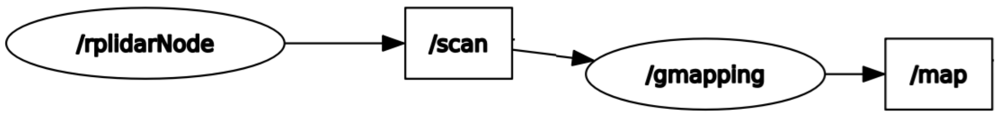


Figure 5-5: ROS nodes and topics launched by *gmapping.launch* file.

5.1.3 Navigation

As it demonstrated in section 5.1.2, the map is successfully generated. Consequently, the rover attains the capability to localize itself within its designated environment and autonomously navigate to predefined positions on the map. This functionality is achieved through the utilization of *move_base* (Link) and *twist_mux* (Link) packages. The *move_base* package gives a navigation goal and utilize path planning algorithm (A*) to reach at designated position via appropriate velocity commands. It also incorporates many other components which have their own ROS API to achieve the navigation capabilities. The ROS API is a list of the ROS *topics*, *services*, *action server* and *messages* that Leo Rover provide to give access to hardware, i.e. servo motors and Lidar sensor [26].

Finally, a *twist_mux* package performs multiplex several velocity commands, prioritizing one over the others. The Figure 5-6 shows the diagram for the *navigation.launch* file is launched. In the configuration, the */move_base* node's inputs include:

- Lidar laser scan data (*/scan topic*)
- Occupied map which is generated by GMapping (*/map topic*)
- Current position estimation from the Odometry (*/odometry/filtered topic*)
- The *move_base/action_topics* send/receive a navigation goal to track the execution status and cancelling operation.

After a navigation goal is being executed, the */move_base node* publishes velocity commands for the Leo Rover on the */nav_vel* topic. In a final process, the */twis_mux node* chooses */nav_vel topic* with velocity commands to forward to the */cmd_vel topic* to achieve the target goal.

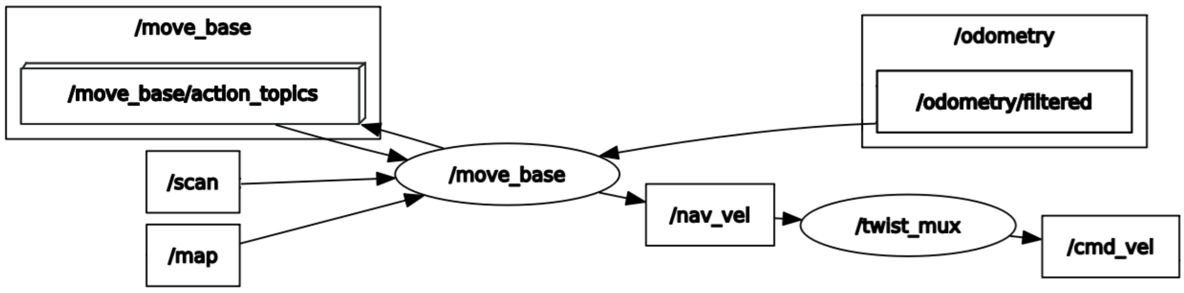


Figure 5-6: ROS nodes and topics by *navigation.launch* file.

6 Object recognition & detection

The chapter focuses on OpenCV library, based on Fiducial marker system which provides detection and recognition capabilities. It also presents both cameras (RPi camera and ZED2 stereo camera) testing results for detecting multiple AR-tags.

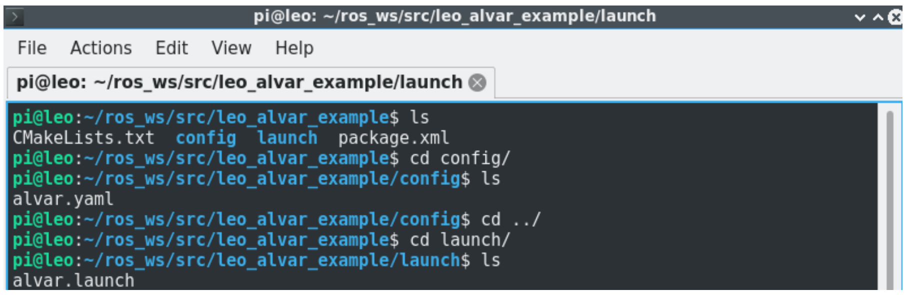
6.1 AR-tags

AR-tags inspired by ARToolKit Fiducial marker system which allows the camera to recognize and locate AR-tags [27]. In a robot's environment, AR-tags servers as a point of reference for detecting and tracking visual markers in real-time applications. It typically consists of a square shape, four corner points and black-and-white patterns with unique Id numbers. When the rover's camera captures an AR-tag, the system computes its pose (positional and orientation along with x, y, z and w axes) of the marker relative to the camera. This information is crucial for end-effector of robotic arm (Pincher arm) to reach specific pose.

6.2 Integration of Alvar package

To detect individual markers in the Leo Rover, *ar_track_alvar* package (GitHub) is installed in ROS workstation (*/ros_ws*) by installing all dependencies in order to test multiple tags. Alvar is a software library, developed by the VTT Technical Research Centre of Finland, is a versatile tool for creating virtual and augmented reality applications. It also offers high-level tools that allows to create AR experiences which is dependent on OpenCV 2.4.0 library [28].

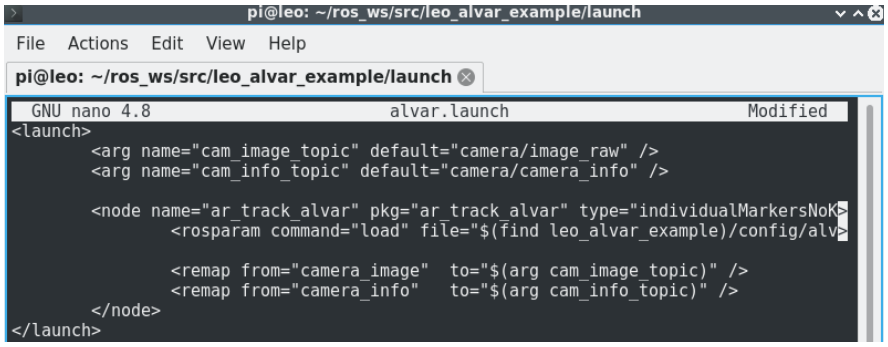
To utilize its functionalities, another package *leo_alvar_example* is employed which integrates with *ar_track_alvar* package. Along with this package, two additional directories (***launch*** and ***config***) are added inside the package to launch and test the Alvar files, as can be seen in Figure 6-1.


pi@leo: ~/ros_ws/src/leo_alvar_example/launch
File
Actions
Edit
View
Help
pi@leo: ~/ros_ws/src/leo_alvar_example/launch

```
pi@leo:~/ros_ws/src/leo_alvar_example$ ls
CMakeLists.txt  config  launch  package.xml
pi@leo:~/ros_ws/src/leo_alvar_example$ cd config/
pi@leo:~/ros_ws/src/leo_alvar_example/config$ ls
alvar.yaml
pi@leo:~/ros_ws/src/leo_alvar_example/config$ cd ../
pi@leo:~/ros_ws/src/leo_alvar_example$ cd launch/
pi@leo:~/ros_ws/src/leo_alvar_example/launch$ ls
alvar.launch
```

Figure 6-1: Alvar package directories and available files.

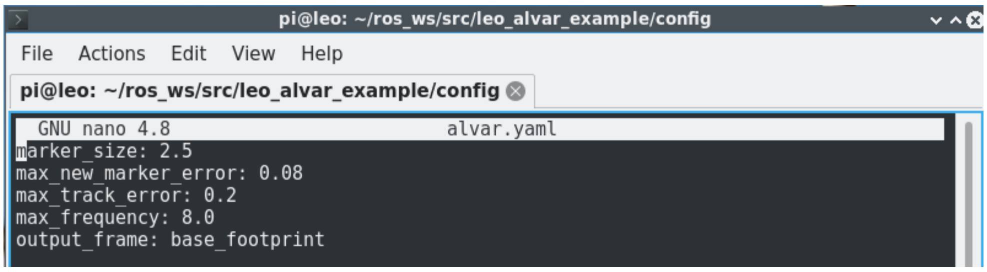
Inside the /launch directory of *leo_alvar_example*, ***alvar.launch*** file is modified along with the following script, shown in Figure 6-2. It is responsible for configuration and launching the *ar_track_alvar* ROS *nodes*, which facilitates individual marker detection. It also sets up the necessary parameters and remapping to ensure proper communication with the rover's camera ROS *topics*.



```
pi@leo: ~/ros_ws/src/leo_alvar_example/launch
File  Actions  Edit  View  Help
pi@leo: ~/ros_ws/src/leo_alvar_example/launch
  GNU nano 4.8                     alvar.launch                      Modified
<launch>
        <arg name="cam_image_topic" default="camera/image_raw" />
        <arg name="cam_info_topic" default="camera/camera_info" />

        <node name="ar_track_alvar" pkg="ar_track_alvar" type="individualMarkersNoK>
                <rosparam command="load" file="$(find leo_alvar_example)/config/alv>

                <remap from="camera_image"  to="$(arg cam_image_topic)" />
                <remap from="camera_info"   to="$(arg cam_info_topic)" />
        </node>
</launch>
```

Figure 6-2: Alvar package launch file [27].

Inside the */config* directory of *leo_alvar_example*, another *alvar.yaml* file is modified along with the following script, shown in Figure 6-3. It serves as a configuration file for fine-tuning parameters related to marker detection and tracking using the *ar_track_alvar* package. The maker size is set to 2.5 centimetres which specify the size of the AR-tag marker. The given marker size is used for further implementation to test it in front of both cameras.



```
pi@leo: ~/ros_ws/src/leo_alvar_example/config
File  Actions  Edit  View  Help
pi@leo: ~/ros_ws/src/leo_alvar_example/config
  GNU nano 4.8                     alvar.yaml
marker_size: 2.5
max_new_marker_error: 0.08
max_track_error: 0.2
max_frequency: 8.0
output_frame: base_footprint
```

Figure 6-3: Alvar Yaml file for fine-tuning parameters [27].

Using *creatMarker* command from *ar_track_alvar* package, *markterdate_0.png* file is generated that stores a size of (2.5cm x 2.5cm) marker with *id_0,* shown in Figure 6-4. This marker is printed it out on a sheet of paper and attached to 3D printed square box with the same dimension. The box is printed it out at USN printer machine.

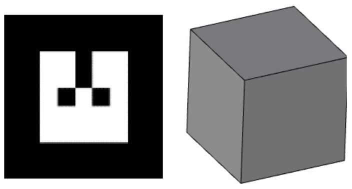
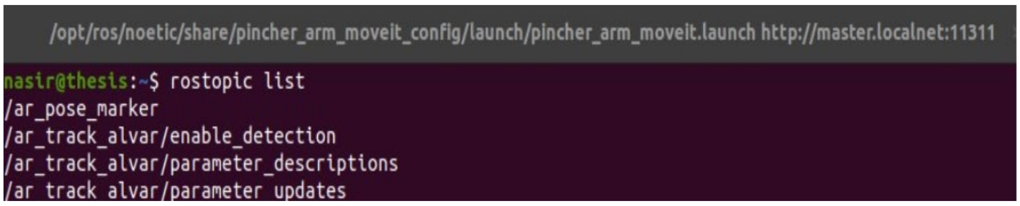


Figure 6-4: Marker (2.5cm x 2.5cm) with *id_0*.



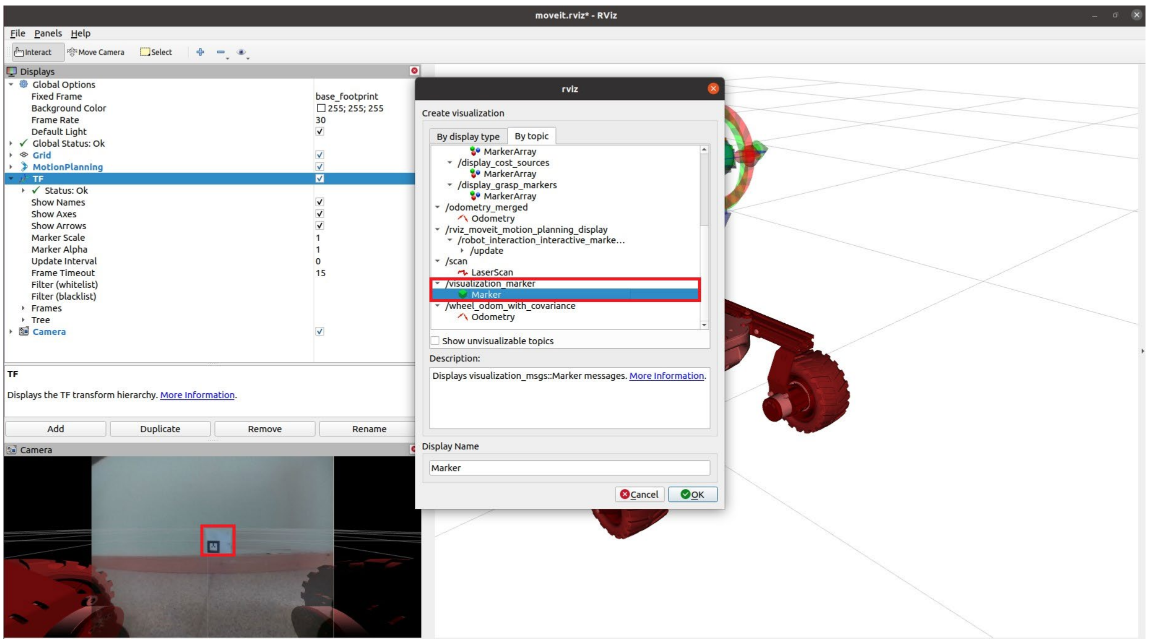
```
/opt/ros/noetic/share/pincher_arm_moveit_config/launch/pincher_arm_moveit.launch http://master.localnet:11311
nasir@thesis:~$ rostopic list
/ar_pose_marker
/ar_track_alvar/enable_detection
/ar_track_alvar/parameter_descriptions
/ar_track_alvar/parameter_updates
```

Figure 6-5: Alvar topics after launching *ar_track_alvar* launch file.

After launching ar_track_alvar files from its package, all ROS *topics* are available, shown in Figure 6-5.

6.3 Testing AR-tags using Leo Rover (RPi) camera

To visualize detected AR-tag in Rviz tool, related *topics* are enabled, including Fixed Frame (*base_foorprint*), Marker (*/visualization_marker*), Image (*/camera/image_raw*) and TF (transform tree), shown the left side of Figure 6-6. In the bottom-left side of the figure, an AR-tag is physically available in front of the Leo Rover camera, shown in red-colour squared box.

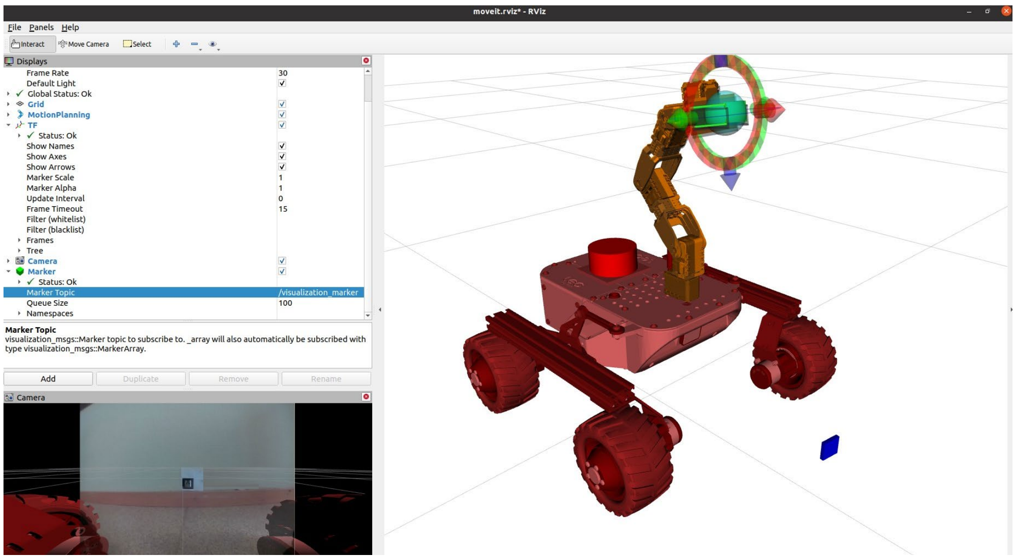


```
Global Options
  Fixed Frame            base_footprint
  Background Color       255; 255; 255
  Frame Rate             30
  Default Light          ✓
Global Status: Ok
Grid                     ✓
MotionPlanning           ✓
TF                       ✓
  Status: Ok
  Show Names             ✓
  Show Axes              ✓
  Show Arrows            ✓
  Marker Scale           1
  Marker Alpha           1
  Update Interval        0
  Frame Timeout          15
  Filter (whitelist)
  Filter (blacklist)
  Frames
  Tree
Camera                   ✓

TF
Displays the TF transform hierarchy. More Information.
```

```
Create visualization
By display type | By topic
      MarkerArray
    /display_cost_sources
      MarkerArray
    /display_grasp_markers
      MarkerArray
  /odometry_merged
    Odometry
  /rviz_moveit_motion_planning_display
    /robot_interaction_interactive_marke...
      /update
  /scan
    LaserScan
  /visualization_marker
    Marker
  /wheel_odom_with_covariance
    Odometry
☐ Show unvisualizable topics
Description:
Displays visualization_msgs::Marker messages. More Information.
Display Name
Marker
```

Figure 6-6: Enabling visualization marker, camera, TF in Rviz software.

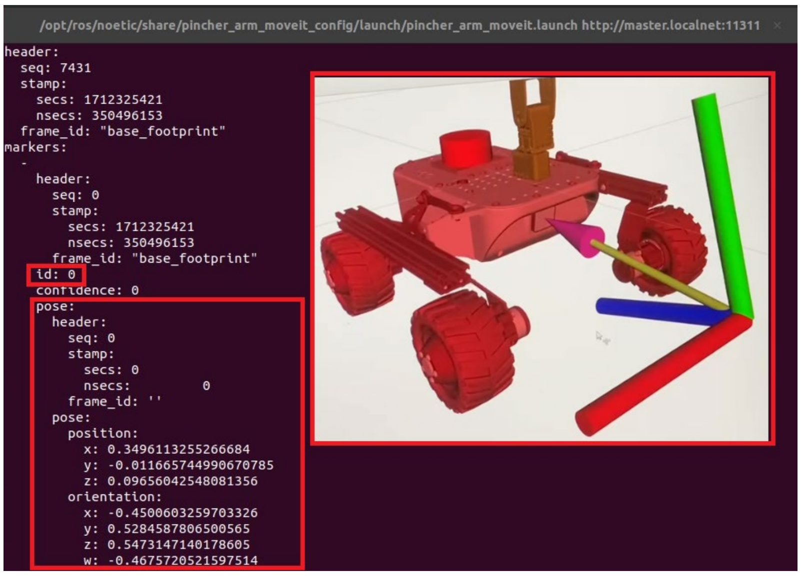
In Figure 6-7, detected AR-tag (in blue colour) is shown in front of the rover model, in Rviz tool. To measure the distance of AR-tag from camera, the *rostopic echo /ar_pose_marker* command is used on RPi terminal, results are shown in Figure 6-8. It checks the tag's ID, position, and orientation. It indicates that the AR-tag (with *Id_0*) is placed at "xyz" position coordinates (0.349, -0.11, 0.096) and "xyz**w**" orientation coordinates (-0.450, 0.528, 0.547, -0.467). The information of axes is written as approximate but it simultaneously change when the tag is moved. Moreover, the tag is approximately 0.34 meters away from the camera. The *TF* is activated to display tag's axis (x, y, and z) position, indicated in red, green, and blue colour respectively. Based on the tag's position and orientation, robotic arm will perform grasping activities, discussed in section.



Frame Rate	30
Default Light	✓
✓ Global Status: Ok	
Grid	✓
MotionPlanning	✓
TF	✓
✓ Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	✓
Marker Scale	1
Marker Alpha	1
Update Interval	0
Frame Timeout	15
Filter (whitelist)	
Filter (blacklist)	
Frames	
Tree	
Camera	✓
Marker	✓
✓ Status: Ok	
Marker Topic	/visualization_marker
Queue Size	100
Namespaces	

Marker Topic
visualization_msgs::Marker topic to subscribe to. _array will also automatically be subscribed with type visualization_msgs::MarkerArray.

Figure 6-7: Detected AR-Tag (*Id_0*).



```
/opt/ros/noetic/share/pincher_arm_moveit_config/launch/pincher_arm_moveit.launch http://master.localnet:11311
header:
  seq: 7431
  stamp:
    secs: 1712325421
    nsecs: 350496153
  frame_id: "base_footprint"
markers:
  -
    header:
      seq: 0
      stamp:
        secs: 1712325421
        nsecs: 350496153
      frame_id: "base_footprint"
    id: 0
    confidence: 0
    pose:
      header:
        seq: 0
        stamp:
          secs: 0
          nsecs:         0
        frame_id: ''
      pose:
        position:
          x: 0.3496113255266684
          y: -0.011665744990670785
          z: 0.09656042548081356
        orientation:
          x: -0.4500603259703326
          y: 0.5284587806500565
          z: 0.5473147140178605
          w: -0.4675720521597514
```

Figure 6-8: Detected AR-tag position and orientation place and its transform tree.

The Figure 6-9 shows the ROS nodes and topics list while running the Alvar package, drowned by *rqt_graph*. It tells the relationship between nodes and topics.

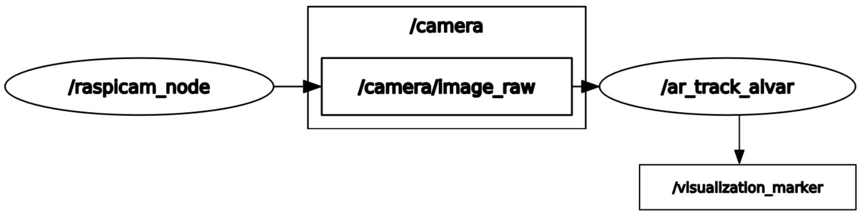
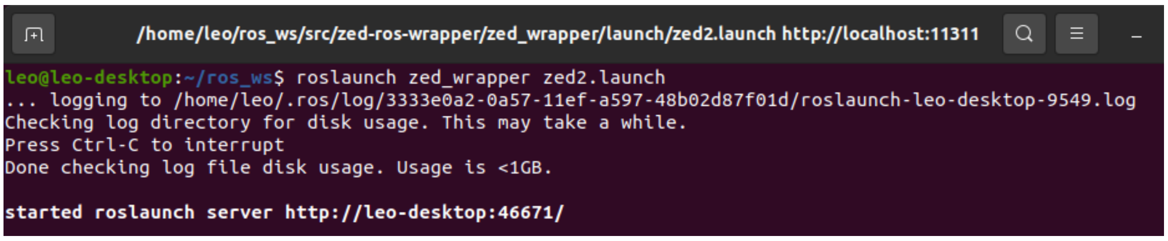


Figure 6-9: ROS *nodes* and *topics* launched by */ar_track_alvar* package.

6.4 Testing AR-tags using ZED2 stereo camera

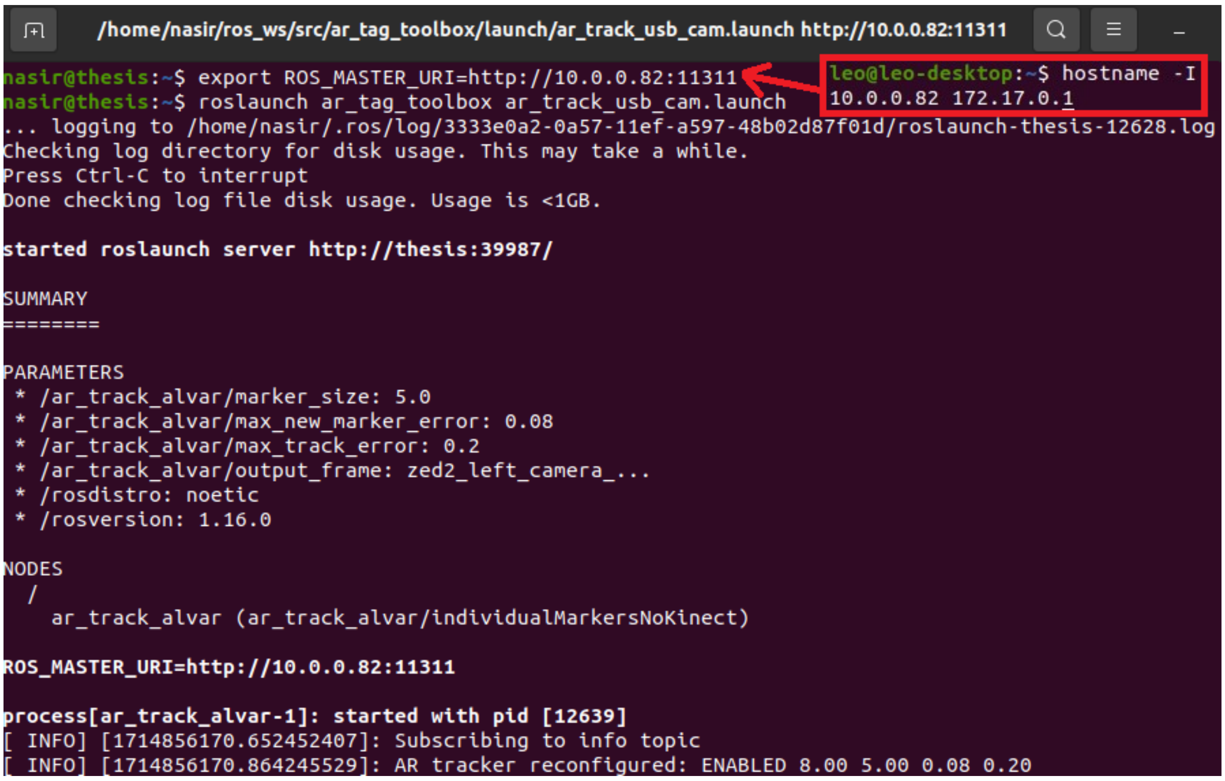
The *ar_track_alvar* package is also tested on ZED2 camera to detect and localize the AR-tag markers. To test the package, *zed2.launch* launch file is launched on additional computer, shown in Figure 6-10.



```
/home/leo/ros_ws/src/zed-ros-wrapper/zed_wrapper/launch/zed2.launch http://localhost:11311
leo@leo-desktop:~/ros_ws$ roslaunch zed_wrapper zed2.launch
... logging to /home/leo/.ros/log/3333e0a2-0a57-11ef-a597-48b02d87f01d/roslaunch-leo-desktop-9549.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://leo-desktop:46671/
```

Figure 6-10: Launching ZED2 camera.

The additional computer ROS network is exported to laptop using its IP address (10.0.0.82) and launched the *ar_track_usb_cam.launch*, shown in Figure 6-11. It automatically starts the ar_track_alvar and zed_wrapper nodes to test marker near to the camera.

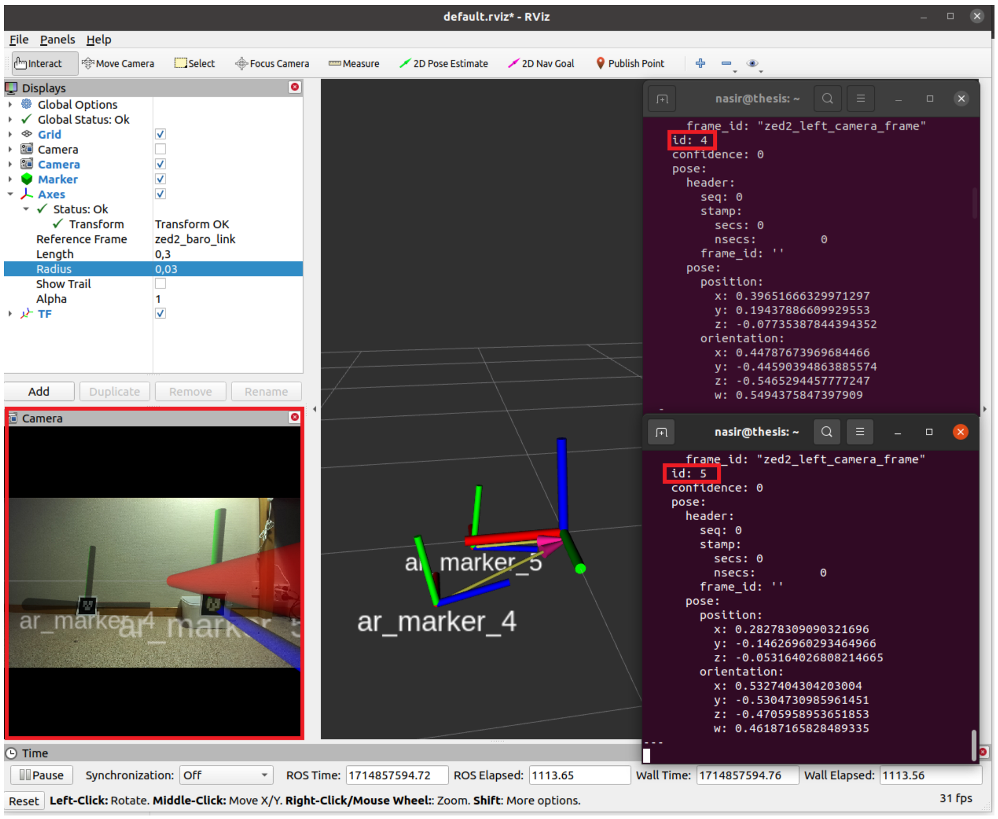


```
/home/nasir/ros_ws/src/ar_tag_toolbox/launch/ar_track_usb_cam.launch http://10.0.0.82:11311
nasir@thesis:~$ export ROS_MASTER_URI=http://10.0.0.82:11311     leo@leo-desktop:~$ hostname -I
nasir@thesis:~$ roslaunch ar_tag_toolbox ar_track_usb_cam.launch  10.0.0.82 172.17.0.1
... logging to /home/nasir/.ros/log/3333e0a2-0a57-11ef-a597-48b02d87f01d/roslaunch-thesis-12628.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://thesis:39987/

SUMMARY
========

PARAMETERS
 * /ar_track_alvar/marker_size: 5.0
 * /ar_track_alvar/max_new_marker_error: 0.08
 * /ar_track_alvar/max_track_error: 0.2
 * /ar_track_alvar/output_frame: zed2_left_camera_...
 * /rosdistro: noetic
 * /rosversion: 1.16.0

NODES
  /
    ar_track_alvar (ar_track_alvar/individualMarkersNoKinect)

ROS_MASTER_URI=http://10.0.0.82:11311

process[ar_track_alvar-1]: started with pid [12639]
[ INFO] [1714856170.652452407]: Subscribing to info topic
[ INFO] [1714856170.864245529]: AR tracker reconfigured: ENABLED 8.00 5.00 0.08 0.20
```

Figure 6-11: Launching Alvar package.

Necessary modifications have been made to package configuration files to enable the camera to capture the images from ZED2 camera. The camera consists of two frames: left and right. Left frame (zed2_lef_camera_frame) is set to test the AR marker. The Figure 6-12 shows the Rviz simulation acquired by ar_track_alvar and zed_wrapper packages. On the top-left corner of the figure, ROS topics such as *Grid, camera, axes, and TF* are activated to display the results.

Two AR-tags (Id_4 and Id_5) are placed in front of camera, as shown in the bottom-left corner of the figure. In the middle of the figure, simulation illustrate the ZED2 camera and tags, represented as three axis (x (green), y (red), and z(blue). By utilizing the *TF* topic, the distance and the names of the tags are displayed. On the right side of the figure, two terminals are running to find the exact pose (position (x, y, and z) and orientation (x, y, z, and w)) of AR-tags, including ar_marker_4(*id_4*) and ar_marker_5(*id_5*).



```
Status: Ok
  Transform           Transform OK
Reference Frame       zed2_baro_link
Length                0,3
Radius                0,03
Show Trail
Alpha                 1
```

```
    frame_id: "zed2_left_camera_frame"
  id: 4
  confidence: 0
  pose:
    header:
      seq: 0
      stamp:
        secs: 0
        nsecs:         0
      frame_id: ''
    pose:
      position:
        x: 0.39651666329971297
        y: 0.19437886609929553
        z: -0.07735387844394352
      orientation:
        x: 0.44787673969684466
        y: -0.44590394863885574
        z: -0.5465294457777247
        w: 0.5494375847397909
-
```

```
    frame_id: "zed2_left_camera_frame"
  id: 5
  confidence: 0
  pose:
    header:
      seq: 0
      stamp:
        secs: 0
        nsecs:         0
      frame_id: ''
    pose:
      position:
        x: 0.28278309090321696
        y: -0.14626960293464966
        z: -0.053164026808214665
      orientation:
        x: 0.5327404304203004
        y: -0.5304730985961451
        z: -0.4705958953651853
        w: 0.46187165828489335
---
```

```
Synchronization: Off   ROS Time: 1714857594.72   ROS Elapsed: 1113.65   Wall Time: 1714857594.76   Wall Elapsed: 1113.56
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.   31 fps
```

Figure 6-12: ZED2 camera results on Rviz.

In the next stage, it is important to determine which camera need to use for grasping activities using rover's arm. So far, both cameras have yielded identical results using the Fiducial marker system. However, each camera has its own pros and cons. For the further implementation, RPi camera is chosen to perform detection for grasping tasks. The reason behind this choice is that the RPi camera offers flexibility and comes with built-in capabilities for the rover. Moreover, it provides comparable results obtained by the ZED2 camera, making it a cost-effective and requires less computation processing. However, running the ZED2 camera on the RPi board would require additional resources, and currently, the RPi board struggles to handle all nodes simultaneously. Despite these challenges, optimize grasping is not yet performed.

7 Optimized grasping position

The chapter presents the simulated and physical testing of rover's arm. The Rviz simulation tool is utilized to perform motion planning and execution via *MoveIt*. It also describes the integration of vision and manipulation with rover's control system to conduct the preliminary tests, via arm's joint control system and Python language (position control system).

7.1 Pincher arm working span area

The PhantomX Pincher arm, manufactured by Trossen Robotics, is a robotic arm that is commonly used for mobile robot platforms. It utilizes the five AX-12A Dynamixel servo motors with unique Id's (1 till 5). These motors are connected to each other and are designed to deliver good performance including high torque. It also offers 4-degree-of-freedom, including an end-effector (gripper) and a full 360-degree range of rotation. Moreover, it relies on Arbotix controller, which serves as an interface between servo motors and rover's RPi board, allowing easy access to integrate with robotic operating system.

A quick overview and specification of the PhantomX Pincher arm is shown Table 7-1 and Figure 7-1.

Table 7-1: Overview of Pincher Arm [29].

PhantomX Pincher Arm	
Degree of Freedom (DOF)	4-DOF
AX-12A Dynamixel servo motors	5
Electronics board	Arbotix-R controller
Total span	64 cm
Gripper strength (holding)	500 grams
Vertical reach	35 cm
Horizontal reach	31 cm
Total weight of arm	550 grams

The workspace of a Pincher arm represents a specific range of motion, where the arm can operate functions. It has a specific workspace that is determined by its manufacturer.

The Figure 7-1 shows top and planner views, that represent the recommended workspace area and arm movements respectively. It covers 640 mm span (360 degree), while 95 mm is accessible height from the base for picking and placing small objects.

Similarly, the working payload of a Pincher arm represents maximum weight (500g), that arm can lift and manipulate easily while performing pick/place operations. It plays a crucial role to consider while selecting or designing an arm for specific tasks. Because it directly influences precision, speed, accuracy, and safety of arm's movements.

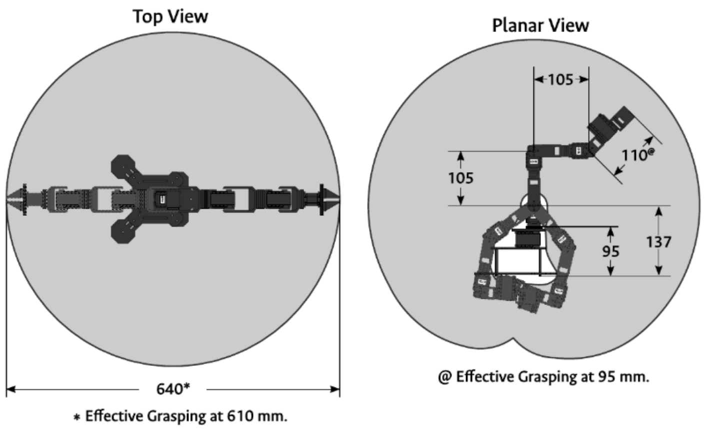

Top View
Planar View
105
105
110@
95
137
640*
* Effective Grasping at 610 mm.
@ Effective Grasping at 95 mm.

Figure 7-1: Pincher arm recommended workspace [29].

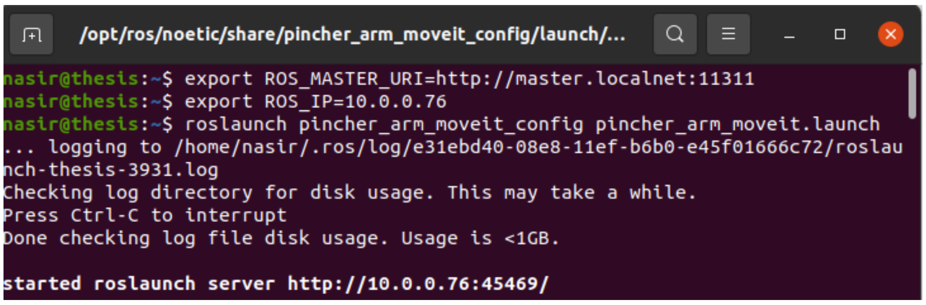
7.2 *MoveIt*

With *MoveIt*, the Leo Rover achieved seamless coordination between Pincher arm and computer-vision capabilities. It is an open-source software framework which facilitates precise motion planning, execution, and control using a set of tools, libraries, and APIs [14]. The pincher arm, with its 4 DOF and compatibility with *MoveIt*, provides valuable results for vision-based pick-and-place tasks using AR-tags. It also helps to develop a module for determining optimized grasp positions based on object's geometry.

7.3 Motion planning and execution with *MoveIt*

To test the *MoveIt* functionalities, it is required to configure laptop with ROS network to run the *nodes* that interfere with the rover's hardware, as well as graphical tools (like *rqt* and *Rviz*) directly on host machines, discusses in appendix section 12.5.7. After installing ROS on the host machine, environment variables are assigned to specify the rover's *master node* and IP address of the host machine.

To test the motion planning, *pincher_arm_moveit.launch* file is launched from *pincher_arm* package, shown in Figure 7-2.



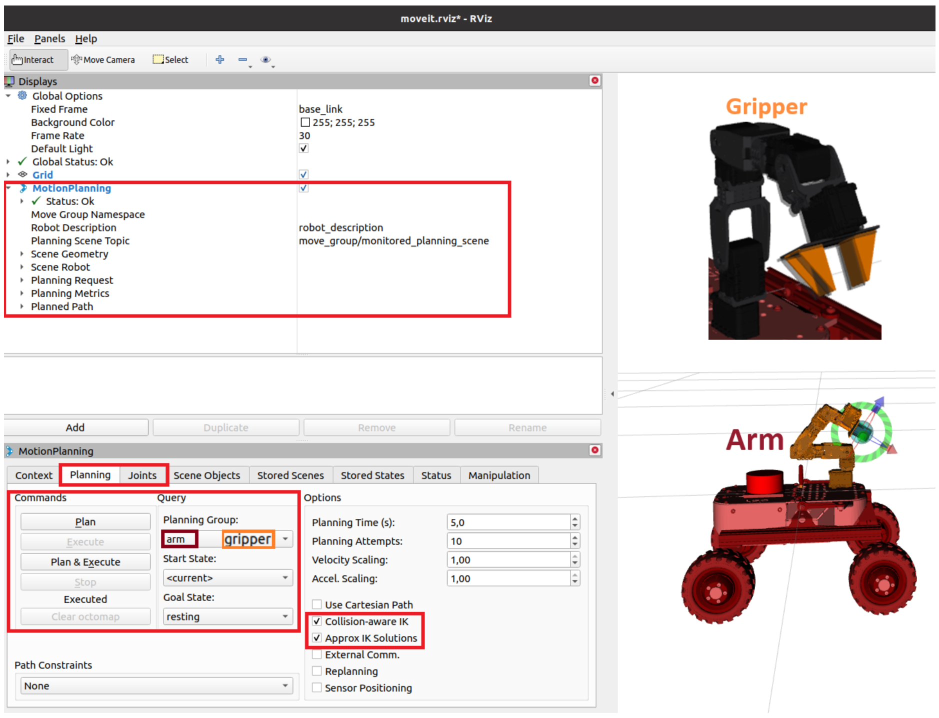
```
nasir@thesis:~$ export ROS_MASTER_URI=http://master.localnet:11311
nasir@thesis:~$ export ROS_IP=10.0.0.76
nasir@thesis:~$ roslaunch pincher_arm_moveit_config pincher_arm_moveit.launch
... logging to /home/nasir/.ros/log/e31ebd40-08e8-11ef-b6b0-e45f01666c72/roslau
nch-thesis-3931.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://10.0.0.76:45469/
```

Figure 7-2: Specifying the IP address of the laptop to access *master node* and launching pincher arm package.

7.3.1 Simulation of Pincher arm

The advantage of *MoveIt* is that it controls the Pinhcer arm using both forward and inverse kinematics. Forward kinematics deals with calculating the pose (position and orientation) of the end-effector (gripper) of an arm when given the joint angles or displacement. On the other hand, inverse kinematics is the reverse process which calculates the joint angles when given the end-effector pose (position and orientation) [30].

The robot model (Leo Rover), with Pincher arm and Lidar sensor, resulting Rviz windows with *MoveIt* plugin, shown in Figure 7-3. There are several tabs, but two most important tabs are used while performing its operations, including *Planning* and *Joints*. In the *Planning* tab, the "Commands" section is used for performing the *Plan & Execute,* which allows rover to follow the instruction after changing joints position. Morver, the "Query" section shows the positions of the arm and end-effector. For example, "Start State" and "Goal State" show current and goal position of arm and end-effector (Gripper). The arm state is checked by default (resting) but can be changed into right-up (straight) and forward. Similarly, the gripper state is also checked by default (gripper-open) and can be changed into gripper-mid and gripper-closed.



Global Options	
Fixed Frame	base_link
Background Color	☐ 255; 255; 255
Frame Rate	30
Default Light	✓
✓ Global Status: Ok	
Grid	✓
MotionPlanning	✓
✓ Status: Ok	
Move Group Namespace	
Robot Description	robot_description
Planning Scene Topic	move_group/monitored_planning_scene
Scene Geometry	
Scene Robot	
Planning Request	
Planning Metrics	
Planned Path	

Query
Planning Group: arm gripper
Start State: <current>
Goal State: resting

Path Constraints: None

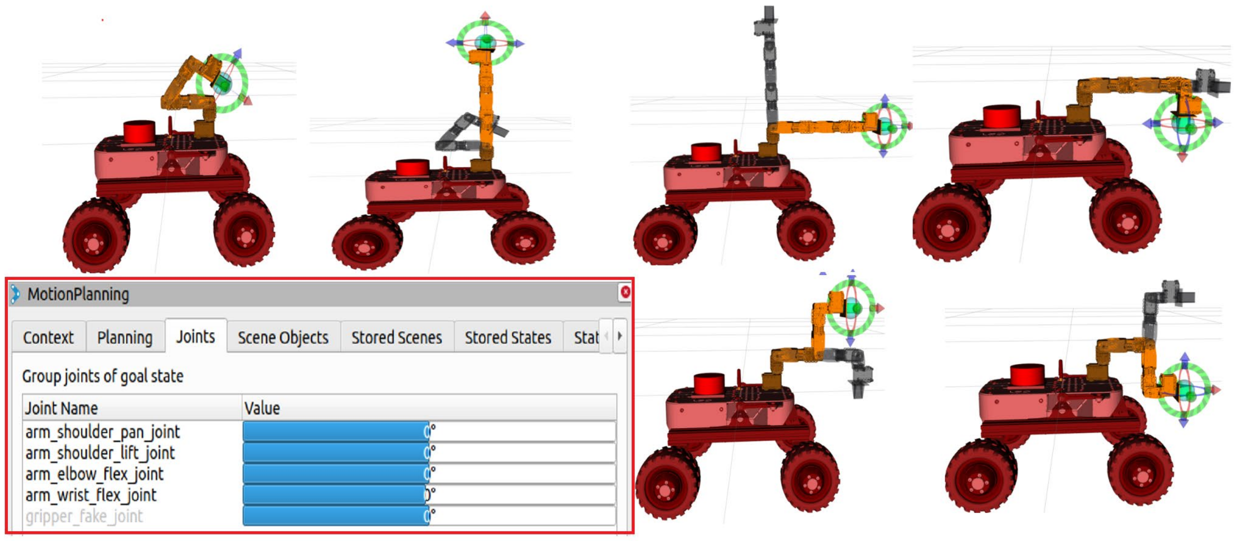
Options

Planning Time (s):	5,0
Planning Attempts:	10
Velocity Scaling:	1,00
Accel. Scaling:	1,00

☐ Use Cartesian Path
✓ Collision-aware IK
✓ Approx IK Solutions
☐ External Comm.
☐ Replanning
☐ Sensor Positioning

Figure 7-3: Robot model with *MoveIt* plugin.

Figure 7-4 and Figure 7-5 show a few different positions of simulated Pincher's arm and gripper positions in Rviz achieved with forward kinematics by utilizing the "Joints" tab. In the figures, the dark gray-shade shows the motion planning (preparation), while the orange-shade shows the execution(action). Motion planning determine the optimal path of trajectory for the robotic and while the execution perform the planned path which is followed by roboti arm system. It also involves actual movement, control, and interaction with in given environment. During the process, both Collision-aware and Approx IK Solutions are being activated.

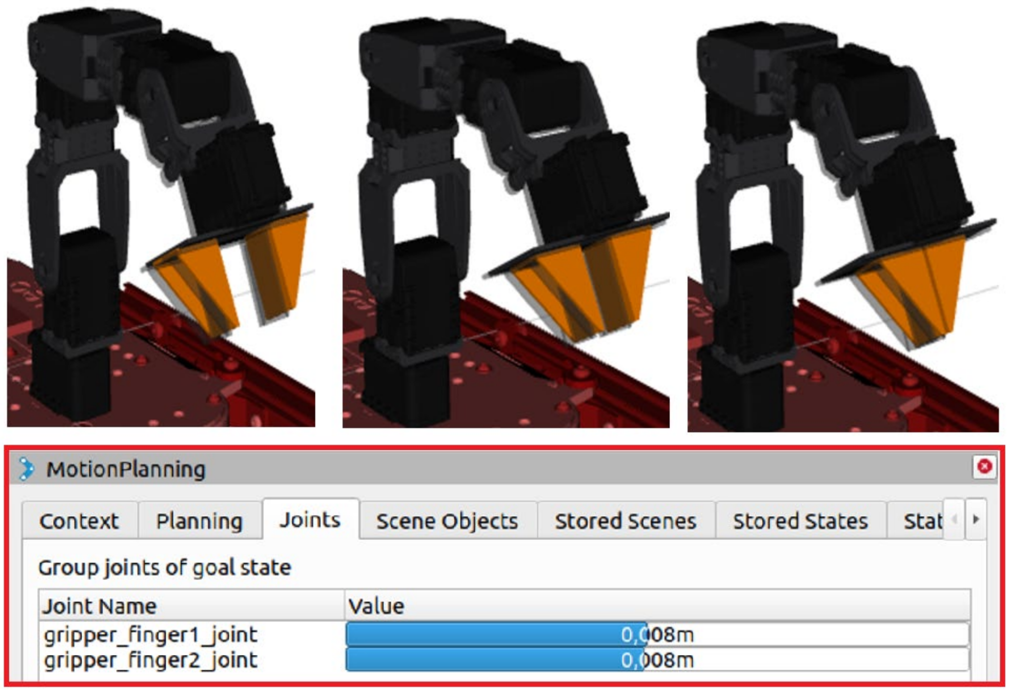

MotionPlanning

Context | Planning | Joints | Scene Objects | Stored Scenes | Stored States | Stat

Group joints of goal state

Joint Name	Value
arm_shoulder_pan_joint	0°
arm_shoulder_lift_joint	0°
arm_elbow_flex_joint	0°
arm_wrist_flex_joint	0°
gripper_fake_joint	0°

Figure 7-4: Pincher’s Arm Motion Planning


MotionPlanning

Context | Planning | Joints | Scene Objects | Stored Scenes | Stored States | Stat

Group joints of goal state

Joint Name	Value
gripper_finger1_joint	0,008m
gripper_finger2_joint	0,008m

Figure 7-5: Pincher’s Gripper Motion Planning.

Note: Motion Planning is also performed with the actual robotic arm which shows the exact same results. For example, when the gripper is closed in simulation, it also closed from the actual robot, etc.

7.3.2 Preliminary testing

There are three ways to control the physical arm. The first method is through commands using Arbotix Controller on Linux terminal, covered section in 4.3.6. The second method is with *MoveIt* in Rviz, covered in section 7.3.1. The third method is with Python scripts, covered in section 7.3.3.

The common thing in these three methods is the startup Pincher arm ROS package. The following command also required to launch the arm for physical testing shown in Figure 7-2. Both Figure 7-6 and Figure 7-7 show the same results which is achieved by *MoveIt*. The arm and gripper positions can be compared with Figure 7-4 and Figure 7-5, pose of arm is same in both simulation and real scenario respectively. Throughout the experiments, the specification results are compared with actual result, shows that the maximum and minimum limits for gripper are 0.016 m and 0.01m which is used for grasping small objects.

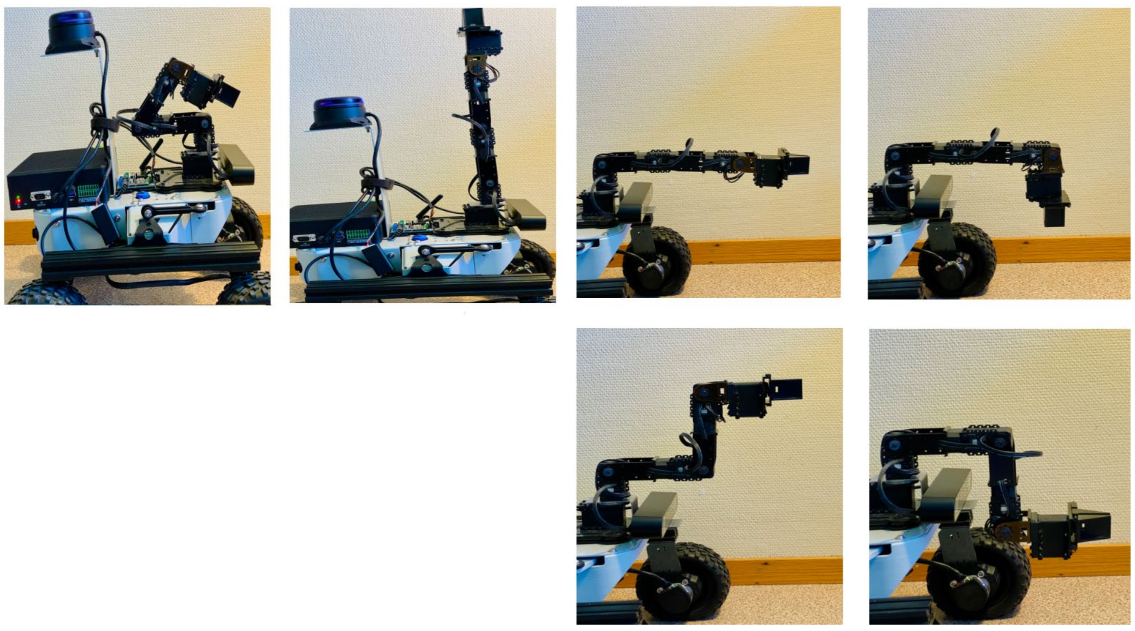


Figure 7-6: Physical testing for arm.

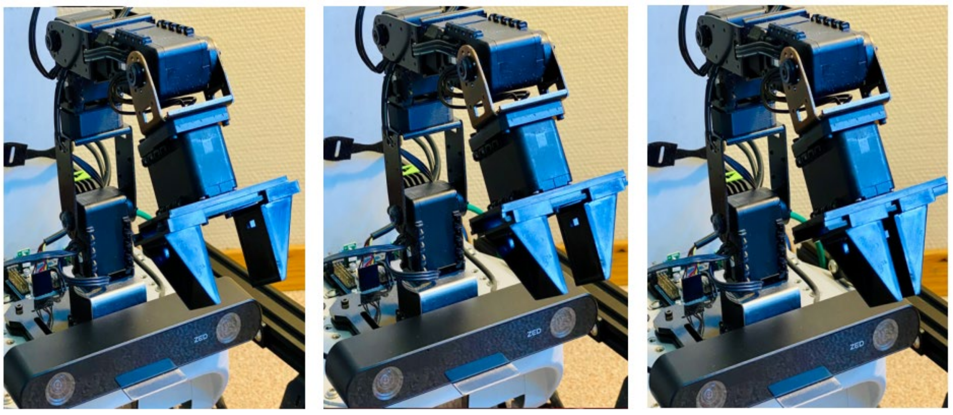
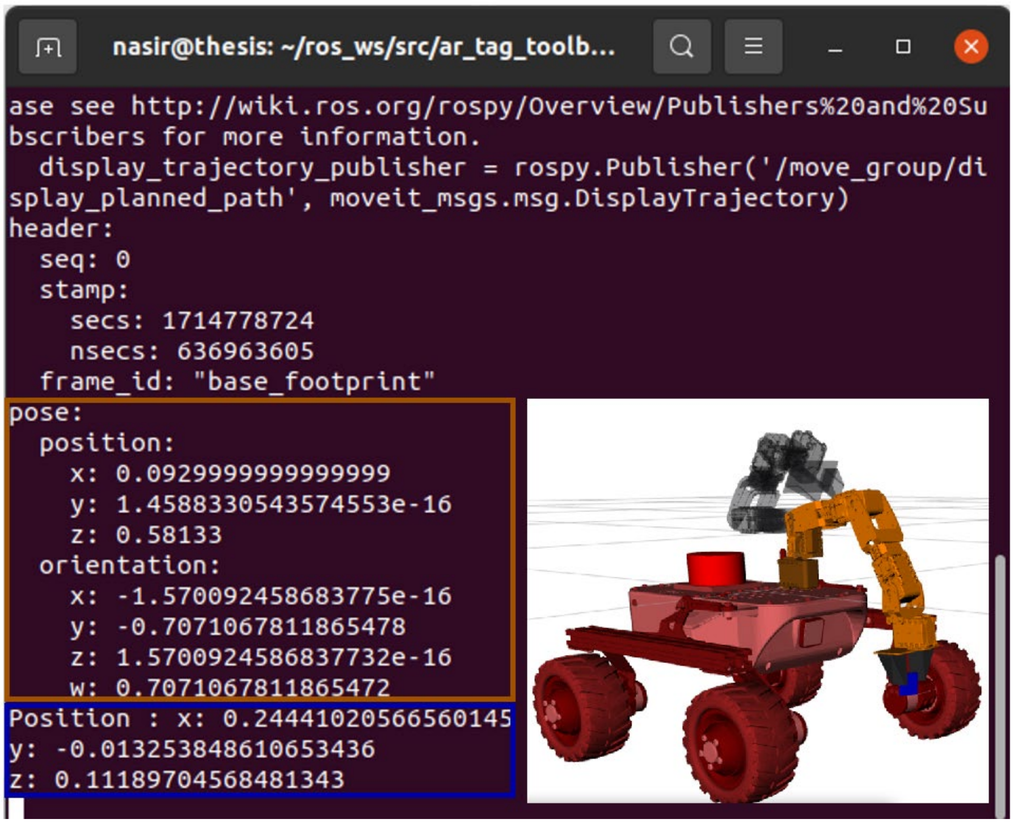


Figure 7-7: Physical testing of Gripper.

7.3.3 Testing Pincher arm using Python script

The following commands (attached with appendix 12.2) executes a Python script for forward kinematic. It allows the rover's arm to reach at specific position, where the object is located. It also identifies the exact pose (position and orientation) of Pincher arm and position of an AR-Tag based objects, shown in Figure 7-8. The orange box shows the current pose (position and orientation) of the arm and the blue box shows the current position of AR-tags. The terminal shows the pose values (x, y, z and w axes) based on the end-effector and tag position. In this case, the task is 0.24 meters away from the rover.



```
ase see http://wiki.ros.org/rospy/Overview/Publishers%20and%20Su
bscribers for more information.
  display_trajectory_publisher = rospy.Publisher('/move_group/di
splay_planned_path', moveit_msgs.msg.DisplayTrajectory)
header:
  seq: 0
  stamp:
    secs: 1714778724
    nsecs: 636963605
  frame_id: "base_footprint"
pose:
  position:
    x: 0.0929999999999999
    y: 1.4588330543574553e-16
    z: 0.58133
  orientation:
    x: -1.570092458683775e-16
    y: -0.7071067811865478
    z: 1.5700924586837732e-16
    w: 0.7071067811865472
Position : x: 0.24441020566560145
y: -0.013253848610653436
z: 0.11189704568481343
```

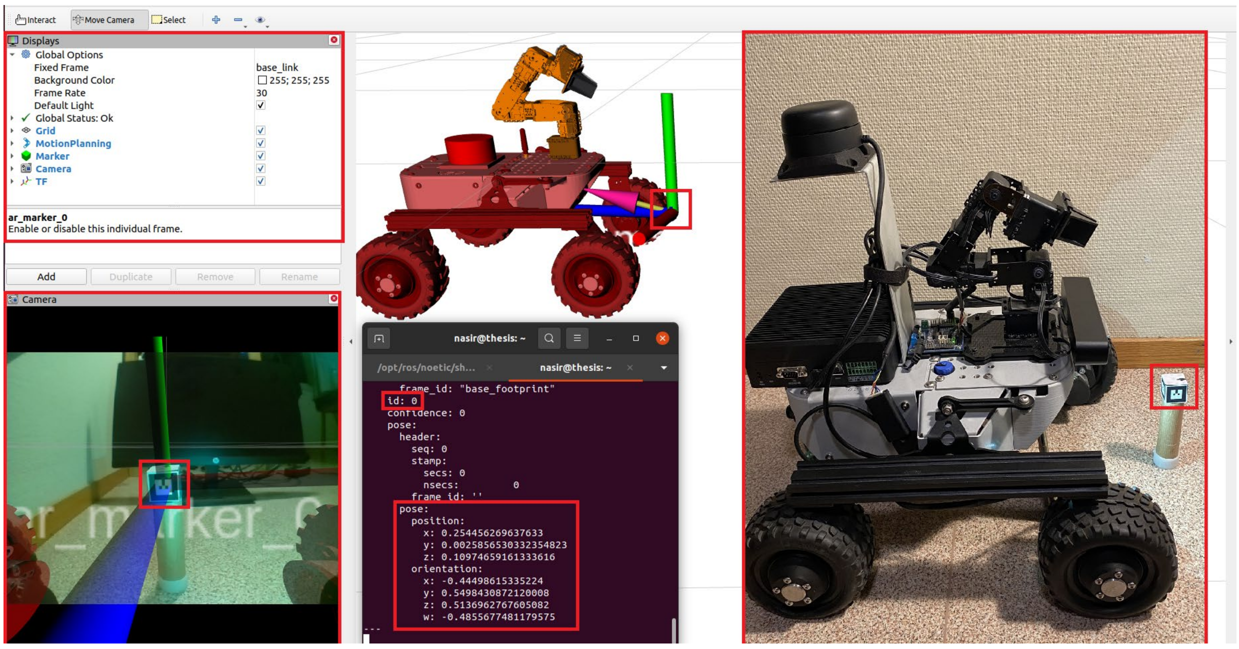
Figure 7-8: Python Script"" identify pose of arm and AR-tag.

7.4 Integration of grasping with AR-tags

To enhance the grasping capabilities of the rover, computer vision and *MoveIt* are integrated together, including the ability to identify AR-tags using both RPi camera and ZED2 stereo camera, perform grasping and return to a resting position. However, results from both cameras are consistent. Therefore, RPi camera is preferred to perform subsequent tasks.
The following two Python scripts are intended for identifying AR-tags based small objects, picking up a 3D printed small solid box, placing inside a plastic box, explained in three sections 7.4.1, 7.4.2 and 7.4.3.

7.4.1 AR-tag real-time identification

Figure 7-9 illustrates how rover detects an AR-tags based object, after running the Python script (attached with appendix section 12.2). The rover identified a small solid box (measuring 2.5 cm on each side) within its environment. Each side of object features an AR-tags (specifically with *Id_0*) attached to it. Remarkably, the RPi camera consistently detects the object and displays its pose (position (x, y, and z axis) and orientation (x, y, z, and w axis)). The object's information is acquired by using the command '*rostopic /echo /ar_pose_marker'* in the Linux terminal, also shown in the middle of figure. On the top-left side of the figure, two main parts are depicted: Rviz tabs and RPi camera. Rviz tabs activated *topics* such as *MotionPlanning, Marker, Camera,* and *TF*. On the bottom-left side of the figure, RPi camera captures the information about the marker (*ar_marker_0*). In the middle of the figure, a simulated rover equipped with an arm is shown along with terminal, identifying the object's location. The *TF* indicates the arrow and information of axis direction, including x (green), y (red), and z(blue). On the right side of the figure, a physical rover and AR-tag can be observed.



Global Options	
Fixed Frame	base_link
Background Color	☐ 255; 255; 255
Frame Rate	30
Default Light	✓
Global Status: Ok	
Grid	✓
MotionPlanning	✓
Marker	✓
Camera	✓
TF	✓

ar_marker_0
Enable or disable this individual frame.

```
    frame_id: "base_footprint"
  id: 0
  confidence: 0
  pose:
    header:
      seq: 0
      stamp:
        secs: 0
        nsecs:          0
      frame_id: ''
    pose:
      position:
        x: 0.254456269637633
        y: 0.0025856530332354823
        z: 0.10974659161333616
      orientation:
        x: -0.44498615335224
        y: 0.5498430872120008
        z: 0.5136962767605082
        w: -0.4855677481179575
---
```

Figure 7-9: AR-tag identification.

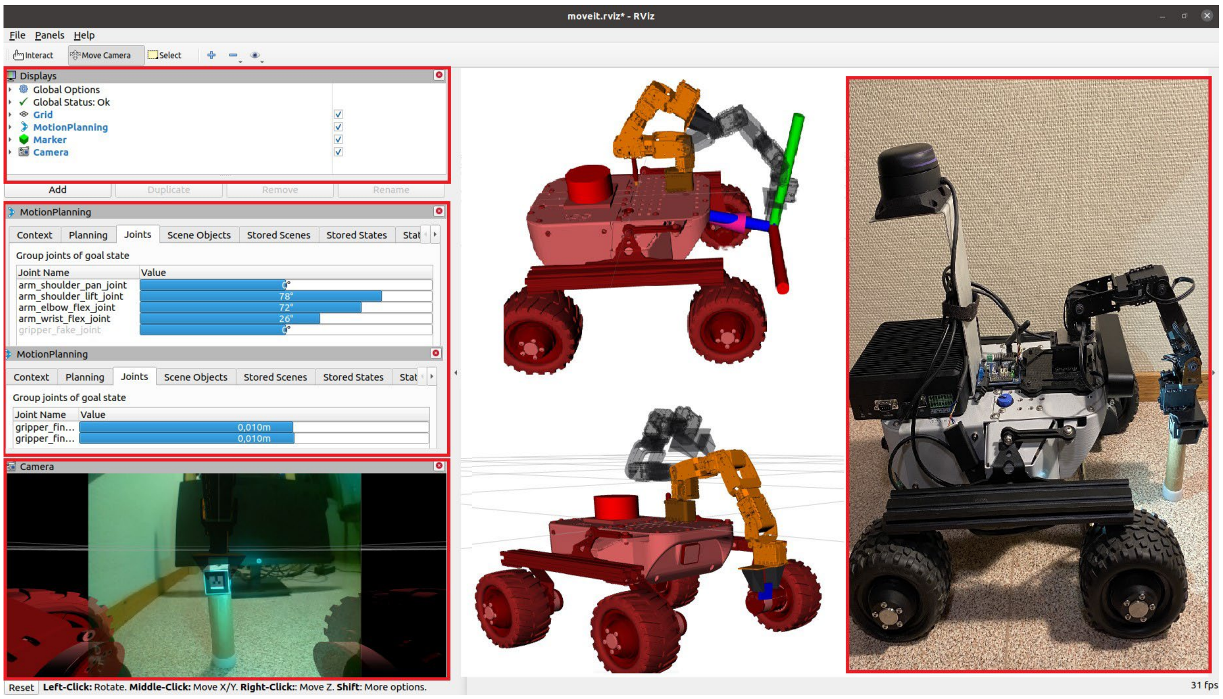
7.4.2 Grasping procedure

In the grasping procedure, there are two methods involve including picking up an AR-tag based object and placing a grasped object to a particular position, explained in sections 7.4.2.1 and 7.4.2.2.

7.4.2.1 Picking up an AR-tag based object

Figure 7-10 illustrates how rover pick an AR-tags based object, after running the Python script (attached with appendix section 12.3). Once an object is detected (shown Figure 7-9), the rover's 4 degree-of-freedom Pincher arm swings into action where the object is located. As shown in middle and right side of the figure, where rover's arm performs Planning (gray-shadow) and Execution (orange-shadow) using *MoveIt*. It also calculates the optimal position of end-effector for grasping, based on object's location, perform precise movements action which allows the rover's arm to securely grasp the small solid box.

On the top-left side of the figure, three main tabs are shown: Displays, *MotionPlanning* and RPi camera. Within the MotionPlanning tab, the Joints tab displays information such as the arm's angle of each joint and the current state of gripper. On the bottom-left side of the figure, it is shown that the arm is positioned where the tag is located. In the middle of the figure, a simulated rover equipped with an arm is shown, performing grasping actions. In the Rviz tool, the AR-tag based object (with *Id_0*) is shown in blue color, where *TF* indicates the arrow and tag's axis direction, including x (green), y (red), and z(blue).


moveit.rviz* - RViz
File Panels Help
Interact Move Camera Select
Displays
Global Options
Global Status: Ok
Grid
MotionPlanning
Marker
Camera
Add Duplicate Remove Rename
MotionPlanning
Context Planning Joints Scene Objects Stored Scenes Stored States Stat
Camera
Reset Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Move Z. Shift: More options.
31 fps

Group joints of goal state

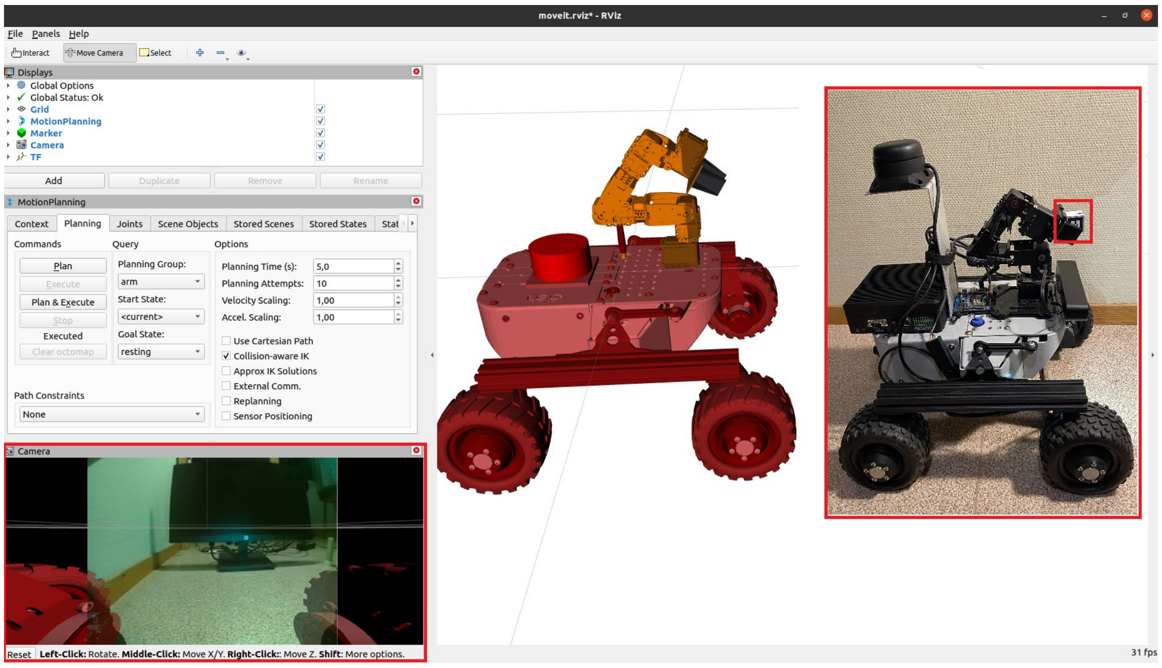
Joint Name	Value
arm_shoulder_pan_joint	0°
arm_shoulder_lift_joint	78°
arm_elbow_flex_joint	72°
arm_wrist_flex_joint	26°
gripper_fake_joint	0°

Group joints of goal state

Joint Name	Value
gripper_fin…	0,010m
gripper_fin…	0,010m

Figure 7-10: Rover's arm is picking an AR-tag based object.

After successful grasping an object, the rover's arm returns to a predefined resting position. It can be seen on the right side of Figure 7-11, the object is grasped, and arm return to predefined position where it started. On the right-side of the figure, actual arm performs the same trajectory execution which are being running in Rviz tool simultaneously.



Planning Group: arm
Start State: <current>
Goal State: resting
Planning Time (s): 5,0
Planning Attempts: 10
Velocity Scaling: 1,00
Accel. Scaling: 1,00
☐ Use Cartesian Path
☑ Collision-aware IK
☐ Approx IK Solutions
☐ External Comm.
☐ Replanning
☐ Sensor Positioning
Path Constraints: None

Reset **Left-Click:** Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Move Z. **Shift:** More options.

Figure 7-11: Rover's arm returns to resting position.

In the result, it is shown that the arm successfully grasped an object and there is no other object available in front of the rover's camera.

After picking up the object, rover needs to place the grasped object at specific position.

7.4.2.2 Placing a grasped object

In the next stage, another python script is utilized (attached with appendix section 12.4). In this scenario, the rover diligently follows the tag which is placed in front of the rover's camera and moves towards the location where it is located. The specific AR-tag (Id_5) affixed to a box (measuring 10cm x 10 cm), shown in Figure 7-12.

Once the rover reaches the designated location, the rover's arm performs a precise motion planning and execution using *MoveIt*, and place the grasped object inside the box, shown in Figure 7-13.

The left side of the figure shows the performed simulation results including motion planning and execution of robotic arm, while the right side shows the physical demonstration of rover and its arm. Notably, both simulation and actual execution show a consistent result. It is clearly shown in Figure 7-13, after placing a grasped object, rover goes back to return position.

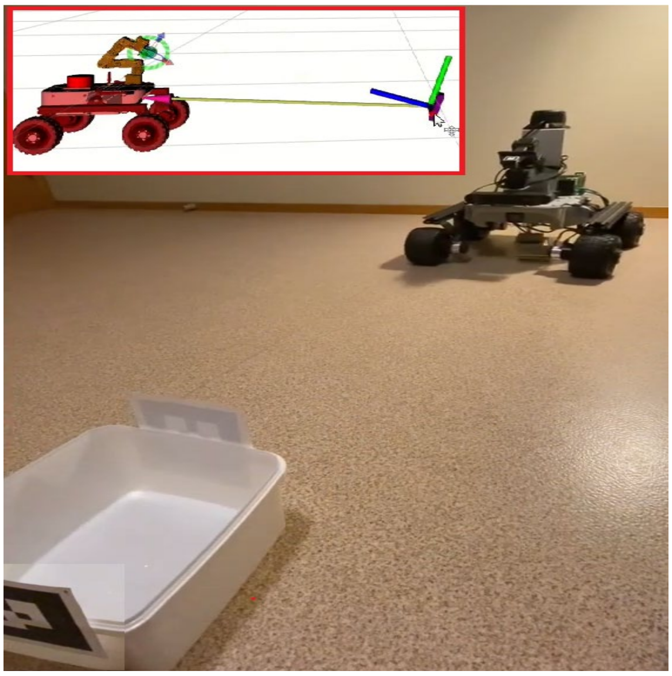


Figure 7-12: Rover is moving toward another AR-tag (*Id_5*).

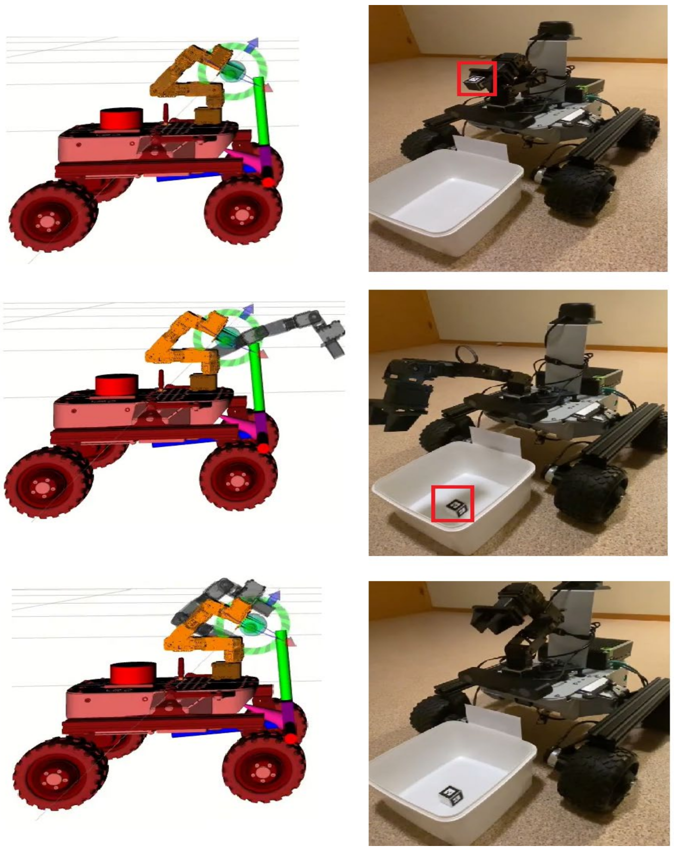


Figure 7-13: Placing an object inside a plastic box.

While running Python script, all ROS *node* and *topic* lists are drawn on *rqt_graph,* shown in Figure 7-14. It displays the relation between activated ROS *nodes* and *topics* while running the Python scripts.

Each ROS *node* and topic communicates with each other using the publish-subscribe protocol system. For example, a *node* that has specific data (*messages*) publishes it to a *topic.* Simultaneously, other *nodes* subscribe to that topic to receive the data (*messages*).

The same protocol applies to the given diagram shown in the Figure 7-14, such as.

- *raspicam_node node* publishes data (*message*) from RPi camera through the */camera/images_images topic*.
- */ar_track_node* subscribe to the */camera/images_raw topic* to get the data from the Rpi camera. This *node* detects an AR-tag and publishes the pose (position and orientation) to three *nodes* including */place*, */pick* and */follow_ar_track* simultaneously. These three nodes are generated by Python scripts.
- */pickup* and */place nodes* are responsible for picking up and placing an object. Both subscribe pose from /ar_track_alvar and publish motion commands to */move_group node,* simultaneously.
- */cmd_vel node* caries velocity command for the rover's wheels and publish movements commands toward LeoCore via */serial_node node.*
- */execute_trajecotry topic* is responsible for the *MoveIt* which perform planned trajectory for the rover's arm.

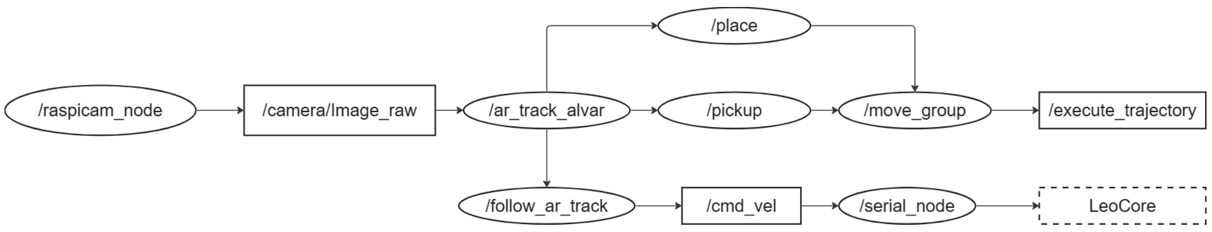


Figure 7-14: ROS *nodes* and *topic* lists (using *rqt_graph*) launched by Python script.

8 Risk assessment and ethical considerations

This chapter identifies potential safety issues and ethical factors related to rover and its robotic arm using preliminary hazard analysis (PHA) method.

Mobile robot equipped with robotic arm and advanced sensor presents a remarkable advancement in automation technologies, offering several benefits in various industries such as warehouses, agriculture sites and hard-to-reach areas. Despite from their benefits, it is also important to perform risk assessments to provide a safe and efficient environment for user. Risk assessment is a systematic process which is used to identify hazards that might occur while performing machine's (robots) operation [31].

8.1 Preliminary hazard analysis (PHA)

The Leo Rover equipped with robotic arm and Lidar sensor perform various functionalities, including autonomous navigation, AR-tags detection, and grasping capabilities. However, the absence of obstacle detection sensor, specifically Sonar sensor, present a safety concern. Sonar is a type of sensor commonly used in mobile robotics applications to detect objects and measure distances. It uses sound waves to detect and located the objects in both environments, underwater and air.

Without Sonar sensor, the rover has significant blind spots, especially above its main body and within the reach of robotic arm. This incapability to stopping the rover create a high collision risk. For example, during SLAM operation, a map is generated with Lidar sensor to test the autonomous navigation. If the navigation goal fell outside the predefined map boundaries, the rover persisted to reach the gaol, even if it beyond its operational range. In such situation, there is a risk of collisions with the boundaries, flipping over which potentially damaging the rover including surrounding electronic equipment such as its robotic arm, and attached sensors. This collision can lead disruption in operation and costly repairs if action is not promptly taken. The action should be taken manually by operator, requires holding this rover and stop it manually. Similarly, using PHA is performed to identify the potential hazards associated with rover and its arm. It is crucial first step in risk assessment, aiming to mention potential hazards early in the development stage, shown in Table 9-1.

Ethical considerations are also important when assessing risk assessment for Leo Rover because it helps to minimize harm, promotes fairness, and fosters trust in technologies, showing in Table 9-2.

Table 9-1: Preliminary Hazard Analysis (PHA) of rover and its robotic arm

System element	Hazard	No.	Hazardous events (what, where, and when)	Causes (Triggering events)	Consequence (Harming conditions)	Risk level (Likelihood and Severity)		Risk-reducing measurement	Responsible person
Leo Rover	Collision hazard	L1	Collision with boundaries while perform SLAM in a map.	Sonar sensor	It could damage Leo Rover including electronic equipment, such as Lidar sensor and Pincher arm.	Medium	High	• Implement Sonar sensor to stop the rover if it collides with boundaries. • Immediate turn-off rover if it goes beyond the boundaries.	Hardware and software engineer
Pincher Arm	Falling objects	P1	Object grasped by the arm falls during movement due to improper gripping strategy.	End-effector (Gripper)	It could damage sensitive objects. For example, if the arm is handling an egg with an AR-tag attached, improper gripping strategy could cause the egg to break.	Medium	Medium	• Design the end-effector securely hold sensitive objects of various shapes and size.	Design engineer
Pincher Arm	End-effector (Gripper) malfunction	P2	Excessive gripping force break the gripper's endpoints (hardware).	Haptic sensor	It could lead to costly repair servo motor and gripper endpoints.	High	Medium	• Equipped the gripper with object sensitivity sensor like Haptic sensor can significantly reduce the risk of damage.	Desing engineer

Table 9-2: Ethical considerations of rover

Consideration	Description
Data privacy and security	Securing stored, encrypted, and transmitted data between the system are important and need to secure communication and protection against the unauthorized access.
Autonomy and decision-making	If full autonomy is given to the rover, it might take some unethical decision. For example, prioritizing its objectives over potential risks to human safety.

9 Discussion

This chapter presents analysis, challenges, and limitations of developed autonomous navigation and advanced grasping system throughout the master thesis.

9.1 Setup and configurations

Two minicomputers were available for configuration of Leo Rover: built-in Raspberry Pi 4b module, and additional computer, specifically the Nvidia reCopmuter J2012. The robot operating system (ROS) was successfully installed on both computers, and their efficiency was compared. Each of them presented its own set of pros and cons for testing the rover's functionalities. The advantage of RPi board was that it has LeoCore controller through GPIO pins, responsible for controlling the rover's wheel motors and IMU sensor. However, configuring with an additional computer required manual intervention.

Upon installing ROS packages on both systems, their performance compared. The additional computer delivered outstanding performance and worked faster than RPi board. Its powerful processor ARM-based CPU and GPU support AI and machine learning tasks provided a significant performance. On the other hand, RPi board has become slow, especially loading multiple packages simultaneously. However, it is preferred to choose RPi board because it directly controlls wheel motors for performing tasks related to autonomous navigations.

9.2 Camera selection

Leo Rover is equipped with a built-in 5-megapixel camera featuring a fisheye lens and night-vision mode. However, for advanced image processing, it was recommended to use ZED2 stereo camera due to its superior capabilities. The RPi does not support stereo cameras due to graphic card limitations. Consequently, it was configured with an additional computer for testing. Initially, determining the optimal location for ZED2 camera was challenging, as the rover already had a built-in camera. After carefully analyses, it is attached with robotic arm servo motor using steel frame, which enabled 360-degree movement.

The object detection tasks were implemented using an OpenCV library, enable AR-tags marker detection, OpenCV library and its results were compared obtained by both cameras. Despite differences in field of view and depth capabilities, the ZED2 camera performed very well as compared to RPi camera. However, the results were the same but the RPi camera exhibited limitation in low-light conditions, particularly detecting small tags (2.5 cm). To address this issue, throughout research analysis, one result found which can be implemented in future work, shown in Figure 10-1.

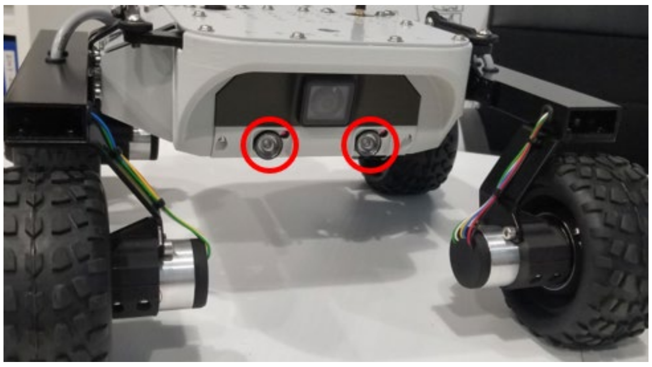


Figure 9-1: Additional front lights for the rover.

9.3 Autonomous navigation

Leo Rover successfully performed SLAM and navigation using Lidar and IMU sensor. Lidar sensor is attached with additional frame to avoid the obstacle while scanning. While IMU sensor is attached on bottom side of the rover and both sensor's axes are aligned with *base_link* of Leo Rover. In the software integration of Lidar sensor, necessary modifications were made in launch files to provide the same joint links for both simulated and physical rover.

Initially, rover faced several challenges to perform autonomous navigation. One of the issues arose when it switched the rover *base_link* to *map* in framework*,* which disrupt the G-mapping functionality. Consequently, rover began generating a new map based on the scanned map, which cause spinning the rover around a particular point where it stuck. To address this issue, all launch files were launched separately, which allowed more systematic and controlled approach to troubleshoot and resolve the navigation issue.

9.4 Object detection

Through literature review, several object detection machine learning algorithms were studied, including YOLO, SSD, and fiducial markers. After careful analysis, Alvar detection method was chosen for Leo Rover due to specific advantages and suitability for the rover's tasks. It is based on fiducial marker system and has ability to detect multiple AR-tags simultaneously.

One of the significant advantages of Alvar detection method, was their ability to provide real-time detection with minimal processing, made well-suited for the rover's electronic board (RPi). Moreover, it allowed to detect multiple tags simultaneously, which was crucial for picking and placing multiple small objects using rover's arm. It also delivered precise results, including distance measurements and pose (position and orientation) based on tag measurements. Unlike SSD and YOLO, demands significant computation resources, including powerful hardware like as ZED2 stereo camera along with additional computer, which costly expensive.

During testing, two approaches were used for utilizing the AR-tags: one for picking an AR-tags based small object and another for following the AR-tags. For example, in the context of picking AR-based small object (Id_0), gripper had specific limitations, which required predefined grasping limit for such tasks. After successfully picking up an object, it needed to be placed inside another AR-tag based box (*Id_5*).

The first approach utilized *Motion Planning* using *MoveIt*, while the second approach relied on following AR-tags, both higher computational resources from the RPi board. Rover was not able to perform both approaches simultaneously because both had different size of tags. To address the issue, both approaches were run separately due to the differences in the size of the tags (2.5 cm and 10 cm respectively). However, despite this issue, the detection results for both approaches were smooth and efficient. And managing different tag sizes, the AR-tag detection process proved successful, enabling the rover to perform its object manipulation and following tasks effectively.

9.5 Arm's motors Ids correction

Initially Pincher arm's motors were given wrong ID's, which lead to stopping the arm motors. In the simulation, it shows the Motional Planning was done. Although, the physical arm did not perform the simulated Motion Plalning.

To address this issue, all motors were given Id's number started from 1 till 5 in ascending orders. There were five motors namely, arm_should_pan_join(id_1), arm_shoulder_lift_joint (id_2), arm_elblow_flex_joint (id_3), arm_wrist_flex_joint (id_5) responsible for control the arm and gripper_joint(id_5) responsible for gripping objects.

9.6 MoveIt

There were multiple algorithms available for motion planning to utilize *MoveIt* configurations. The Pincher arm detected all kind of AR-tags pose (position and orientation) nearby the rover's camera and successfully performed grasping activities manually, by providing pose information via a Python script. However, it also automatically performs grasping based on tag position, but it encountered difficulties in execution tasks which were based on orientation. Despite the presence of several algorithm in its directories, attempted to follow them during grasping task but could not being successfully followed. Because the Path planning algorithms provided by *MoveIt* mainly focuses on 6DOF but for this rover 4DOF is utilized. To address this issue, AR-tags were placed in predefined orientations. This adjustment enabled smother motion planning and execution.

9.7 Simulation environment

All ROS packages were typically launched manually in Leo Rover operating systems, which are based on RPi board. After loading the Rviz tool on the same operating system simultaneously executing required package, as a result, the system slowed down significantly. It was not allowed to perform any task on RPi board because of its processor limitations. To overcome this issue, a solution was implemented whereby the same ROS version was installed on Linux operating system. By doing so, the ROS network was exported which enabled the seamless utilization of the Rviz tool with improved performance.

10 Conclusion

This master's thesis has successfully implemented a computer vision-guided grasping system into mobile robot Leo Rover, which equipped with a robotic arm, Lidar, RPi and stereo. Initially, literature review was conducted to understand the current stage of research areas in mobile robotics, highlighting the importance of computer vision, sensor fusion and machine learning techniques for object handling in mobile robotics. Subsequently, a system architecture was designed to ensure seamless communication and coordination among the connected components by utilizing multiple ROS packages into the rover's operating system.

Overall, Leo Rover perception abilities, including autonomous navigation, object detection based on AR-tags, and optimized grasping capabilities, were significantly improved. These improvement results were achieved through the utilization of simultaneous localization and mapping technique, employing Fiducial marker system, and execution of Motion Planning using *MoveIt,* respectively. As a result, Leo Rover independently navigates and locates AR-tags based objects within nearby areas to perform grasping activities (pick-and-place). These results were achieved through testing and evaluation in both simulated and actual environment.

10.1 Future work

Throughout the achievement, rover can perform autonomous navigation using Lidar. However, sometime it collides with boundaries if the path planning algorithm did not work properly within the map. To address this issue, the integration of Sonar sensor could serve to stop the rover upon colliding with boundaries, which will also enhance rover capabilities.

Additionally, the object detection was not good in low-light conditions. Attaching additional LED alongside the camera could improve visibility in darker areas. Consequently, the utilization of the RPi 4B module for testing the rover's functionalities on Rviz tool presented some limitation while running all ROS package simultaneously. Upgrading to a more powerful computer, like Nvidia Jetson, can increase the overall performance of rover and provide more sustainable results. Finally, while testing rover's arm, the AR-tag based object orientation needs to be adjusted to grasp it using robotic arm. In future work, employing a 6-DOF robotic arm would yield more efficient outcomes compared to a 4-DOF of arm. Addressing these aspects in future work will improve the results for further development in this project.

11 References

[1] T. F. Agidew, ‘Mechatronics & Robotics: Robotics’, Mechatronics & Robotics. Accessed: Apr. 07, 2024. [Online]. Available: https://tayeonblogger.blogspot.com/p/robotics.html

[2] K. Brush, ‘Mobile Robotics’, IoT Agenda. Accessed: Apr. 07, 2024. [Online]. Available: https://www.techtarget.com/iotagenda/definition/mobile-robot-mobile-robotics

[3] D. J. Yeong, G. Velasco-Hernandez, J. Barry, and J. Walsh, ‘Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review’, *Sensors*, vol. 21, no. 6, p. 2140, Mar. 2021, doi: 10.3390/s21062140.

[4] B. Yildiz, A. Durdu, A. Kayabaşi, and M. Duramaz, ‘CNN based sensor fusion method for real-time autonomous robotics systems’, *Turk. J. Electr. Eng. Comput. Sci.*, vol. 30, no. 1, pp. 79–93, Jan. 2022, doi: 10.3906/elk-2008-147.

[5] Y.-L. Chen, Y.-R. Cai, and M.-Y. Cheng, ‘Vision-Based Robotic Object Grasping—A Deep Reinforcement Learning Approach’, *Machines*, vol. 11, no. 2, p. 275, Feb. 2023, doi: 10.3390/machines11020275.

[6] M. B. Alatise and G. P. Hancke, ‘A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods’, *IEEE Access*, vol. 8, pp. 39830–39846, 2020, doi: 10.1109/ACCESS.2020.2975643.

[7] Z. Zhou, L. Li, A. Fürsterling, H. J. Durocher, J. Mouridsen, and X. Zhang, ‘Learning-based object detection and localization for a mobile robot manipulator in SME production’, *Robot. Comput.-Integr. Manuf.*, vol. 73, p. 102229, Feb. 2022, doi: 10.1016/j.rcim.2021.102229.

[8] Z. Xie, X. Liang, and C. Roberto, ‘Learning-based robotic grasping: A review’, *Front. Robot. AI*, vol. 10, p. 1038658, Apr. 2023, doi: 10.3389/frobt.2023.1038658.

[9] F.-C. Cheng, C.-C. Yen, and T.-S. Jeng, ‘Object Recognition and User Interface Design for Vision-based Autonomous Robotic Grasping Point Determination’, presented at the CAADRIA 2021: Projections, Hong Kong, 2021, pp. 633–642. doi: 10.52842/conf.caadria.2021.1.633.

[10] Fictionlab, ‘Leo Rover Documentation - Specification’. Accessed: Apr. 24, 2024. [Online]. Available: https://www.leorover.tech/documentation/specification

[11] ‘Robots/Leo Rover - ROS Wiki’. Accessed: Apr. 26, 2024. [Online]. Available: https://wiki.ros.org/Robots/Leo%20Rover

[12] ‘Raspberry Pi 4 Information’. Accessed: Apr. 25, 2024. [Online]. Available: https://www.pishop.ca/raspberry-pi-4-information/

[13] ‘CORE2 | Husarion’. Accessed: Apr. 25, 2024. [Online]. Available: https://husarion.com/manuals/core2/

[14] Fictionlab, ‘Leo Rover Integrations - PhantomX Pincher’. Accessed: Mar. 25, 2024. [Online]. Available: https://www.leorover.tech/integrations/phantomx-pincher

[15] S. Battle, ‘Principles of Robot Autonomy I’. Accessed: May 13, 2024. [Online]. Available: https://stanfordasl.github.io/PoRA-I/aa174a_aut2324/

[16] A. Ahmad and M. A. Babar, ‘Software architectures for robotic systems: A systematic mapping study’, *J. Syst. Softw.*, vol. 122, pp. 16–39, Dec. 2016, doi: 10.1016/j.jss.2016.08.039.

[17] Fictionlab, ‘Leo Rover Documentation - Hardware structure’. Accessed: May 11, 2024. [Online]. Available: https://www.leorover.tech/documentation/hardware-structure

[18] Halvorsen, *Software Development*.

[19] H.-P. Halvorsen, ‘Software Development a Practical Approach’, Docslib. Accessed: May 14, 2024. [Online]. Available: https://docslib.org/doc/1178867/software-development-a-practical-approach

[20] S. Sebo, ‘Intro Robotics’. Accessed: Apr. 24, 2024. [Online]. Available: https://classes.cs.uchicago.edu/archive/2021/winter/20600-1/ros_resources.html

[21] Fictionlab, ‘Leo Rover Integrations - RPLiDAR A2M8 / A2M12’. Accessed: Mar. 25, 2024. [Online]. Available: https://www.leorover.tech/integrations/rplidar

[22] Fictionlab, ‘Leo Rover Documentation - Software structure’. Accessed: May 11, 2024. [Online]. Available: https://www.leorover.tech/documentation/software-structure

[23] Fictionlab, ‘Leo Rover Integrations - ZED Stereo Camera’. Accessed: Mar. 25, 2024. [Online]. Available: https://www.leorover.tech/integrations/zed

[24] ‘ROS: Home’. Accessed: Apr. 19, 2024. [Online]. Available: https://www.ros.org/

[25] Fictionlab, ‘Leo Rover Developer Guides - ROS Development’. Accessed: Apr. 19, 2024. [Online]. Available: https://www.leorover.tech/guides/ros-development

[26] ‘Leo Rover Developer Guides - Autonomous Navigation’. Accessed: Apr. 20, 2024. [Online]. Available: https://www.leorover.tech/guides/autonomous-navigation

[27] Fictionlab, ‘Leo Rover Integrations - ARTag tracking with Alvar’. Accessed: May 12, 2024. [Online]. Available: https://www.leorover.tech/integrations/alvar

[28] S. Niekum, ‘ar_track_alvar: ALVAR 2.0.0’. Accessed: Apr. 13, 2024. [Online]. Available: https://docs.ros.org/en/melodic/api/ar_track_alvar/html/index.html

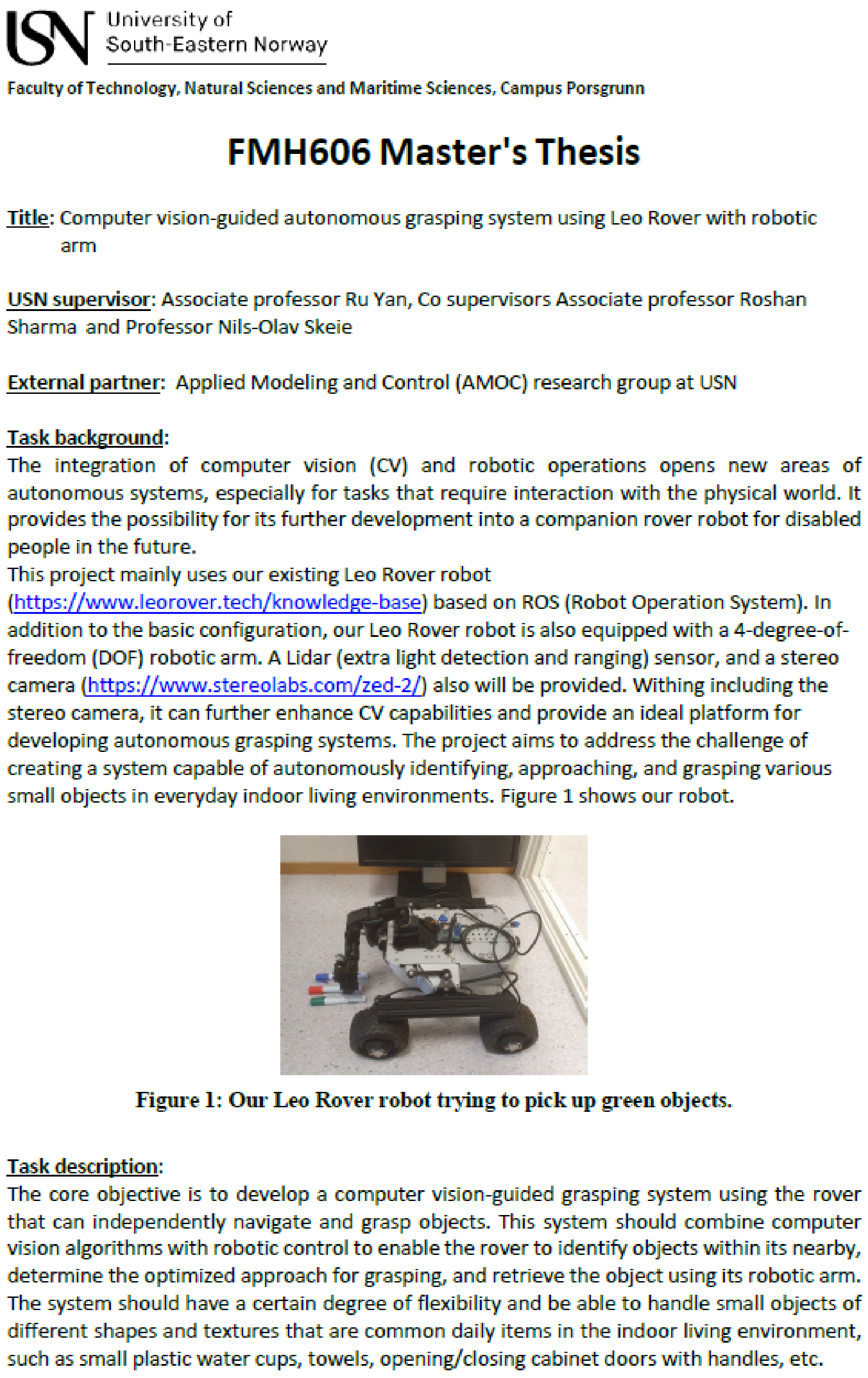
[29] H. Toquica Cáceres, *PhantomX Pincher Specifications*. 2018. doi: 10.13140/RG.2.2.28484.12160.

[30] A. Stevens, ‘Forward Kinematics’, Accessed: May 11, 2024. [Online]. Available: https://opentextbooks.clemson.edu/wangrobotics/chapter/forward-kinematics/

[31] C. Bernier, ‘Robotic Risk Assessment: Who, Why, and How? | HowToRobot’. Accessed: May 05, 2024. [Online]. Available: https://howtorobot.com/expert-insight/robotic-risk-assessment

12 Appendices

12.1 Appendix A – FMH606 Master thesis task description



University of
South-Eastern Norway

Faculty of Technology, Natural Sciences and Maritime Sciences, Campus Porsgrunn

FMH606 Master's Thesis

Title: Computer vision-guided autonomous grasping system using Leo Rover with robotic arm

USN supervisor: Associate professor Ru Yan, Co supervisors Associate professor Roshan Sharma and Professor Nils-Olav Skeie

External partner: Applied Modeling and Control (AMOC) research group at USN

Task background:
The integration of computer vision (CV) and robotic operations opens new areas of autonomous systems, especially for tasks that require interaction with the physical world. It provides the possibility for its further development into a companion rover robot for disabled people in the future.
This project mainly uses our existing Leo Rover robot (https://www.leorover.tech/knowledge-base) based on ROS (Robot Operation System). In addition to the basic configuration, our Leo Rover robot is also equipped with a 4-degree-of-freedom (DOF) robotic arm. A Lidar (extra light detection and ranging) sensor, and a stereo camera (https://www.stereolabs.com/zed-2/) also will be provided. Withing including the stereo camera, it can further enhance CV capabilities and provide an ideal platform for developing autonomous grasping systems. The project aims to address the challenge of creating a system capable of autonomously identifying, approaching, and grasping various small objects in everyday indoor living environments. Figure 1 shows our robot.

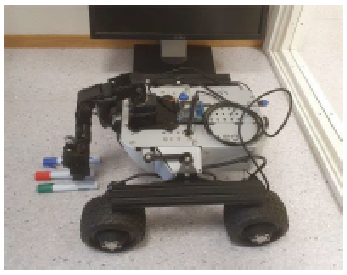


Figure 1: Our Leo Rover robot trying to pick up green objects.

Task description:
The core objective is to develop a computer vision-guided grasping system using the rover that can independently navigate and grasp objects. This system should combine computer vision algorithms with robotic control to enable the rover to identify objects within its nearby, determine the optimized approach for grasping, and retrieve the object using its robotic arm. The system should have a certain degree of flexibility and be able to handle small objects of different shapes and textures that are common daily items in the indoor living environment, such as small plastic water cups, towels, opening/closing cabinet doors with handles, etc.

The main tasks are:

1. Conduct a literature survey on computer vision (CV) and sensor fusion in robotics, machine learning for object handling, and robotic adaptive grasping.
2. Assemble the Lidar sensor and stereo camera on the rover, interfacing them with ROS.
3. Define the system architecture, including the integration of the stereo camera, robotic arm, and the computing platform for the rover.
4. Incorporate the "leo_navigation" package into the rover, ensuring functionality for SLAM (Simultaneous localization and mapping) and autonomous navigation with obstacle avoidance.
5. Select efficient algorithms for object detection, recognition, and the grasp planning needed for the robotic arm.
6. Develop or adapt a CV module for capable of real-time object identifying and locating.
7. Develop a module for determining optimized grasp positions based on object's geometry and the rover's capabilities.
8. Integrate vision and manipulation systems with the rover's control system and conduct preliminary tests.
9. Perform tests to evaluate the rover's object grasping capabilities, including a test report.
10. Conduct a risk assessment to identify any potential safety issues and ethical considerations related.

Student category: IIA

Is the task suitable for online students (not present at the campus)? No. IIA campus student only. This project will make full use of the Leo Rover at USN through hands-on tasks. Therefore, the student needs to be on campus for these tasks.

Practical arrangements:
Necessary equipment, including a Leo Rover, an AI stereo camera, a Lidar sensor, and a shared workstation computer equipped with Geforce RTX 4090 video card Core i9, 64GB DDR5 will be available to the students, together with the access to the Sensor lab at USN Porsgrunn campus.

Supervision:
As a general rule, the student is entitled to 15-20 hours of supervision. This includes necessary time for the supervisor to prepare for supervision meetings (reading material to be discussed, etc).

Signatures:

Ru Yan Digitally signed by Ru Yan
Date: 2024.02.05
12:09:33 +01'00'

Supervisor (date and signature):

Student (write clearly in all capitalized letters): NASIR ALI

Student (date and signature): [illegible] 05.02.2024

12.2 Appendix B – Python script for testing both AR-tag and end-effector pose

GitHub Link for the given below code is attached:

```python
#Name: Nasir Ali
#Date: 02.15.2024
#!/usr/bin/env python
import sys
import rospy
from ar_track_alvar_msgs.msg import AlvarMarkers
from geometry_msgs.msg import PoseStamped
import moveit_commander
from moveit_commander import PlanningSceneInterface
import time
import copy
import moveit_msgs.msg

class MoveArm():
    def __init__(self):
        rospy.init_node('arm_controller', anonymous=True)
        rospy.Subscriber("/ar_pose_marker", AlvarMarkers,self.callback)
        self.position = None
        self.orientation = None
        self.data = None  # Store data as an instance variable
        moveit_commander.roscpp_initialize(sys.argv)
        scene = PlanningSceneInterface()
        robot = moveit_commander.RobotCommander()
        self.arm_group= moveit_commander.MoveGroupCommander("arm")
        self.gripper_group= moveit_commander.MoveGroupCommander("gripper")
        current_state = robot.get_current_state()
        current_pose = self.arm_group.get_current_pose().pose

        # Print the current pose
        print("Current End Effector Pose:")
        print(current_pose)
        self.grasp_pose = PoseStamped()
        self.movearm()
        rospy.spin()

    def callback(self, data):
        self.data = data  # Store data
        for marker in data.markers:
            self.position = marker.pose.pose.position
            self.orientation = marker.pose.pose.orientation
            self.frame_id = marker.header.frame_id
            self.pose = marker.pose.pose

            #It displays the marker's detatils continouesly
            #print("Marker ID:", marker.id)
            #print("Position :", self.position)
            #print("Orientation :", self.orientation)

    def movearm(self):
        time.sleep(5)
        print()
        print("AR-Tag Pose:")
        print(self.pose)

        # print("AR-Tag Orientation :", self.orientation)
        if self.position is not None:

            self.grasp_pose.header.frame_id = "base_footprint"
            self.grasp_pose.pose.position.x = 0.2502012225613189
            self.grasp_pose.pose.position.y = -0.0032156971548337535
            self.grasp_pose.pose.position.z = 0.17652794280001272

            self.grasp_pose.pose.orientation.x = 0.007705798279333313
            self.grasp_pose.pose.orientation.y = 0.7534836526780807
            self.grasp_pose.pose.orientation.z = -0.006723028973121577
            self.grasp_pose.pose.orientation.w = 0.657387105670017
            self.arm_group.set_pose_target(self.grasp_pose)

            #plan = self.arm_group.plan()
            #self.arm_group.execute(plan[1])

if __name__ == '__main__':
    move_arm = MoveArm()
```

12.3 Appendix C – Python script for optimized grasping (Picking)

GitHub Link for the given below code is attached:

```
#Name: Nasir Ali
#Date: 02.15.2024
#!/usr/bin/env python
import sys
import rospy
from ar_track_alvar_msgs.msg import AlvarMarkers
from geometry_msgs.msg import PoseStamped
import moveit_commander
from moveit_commander import PlanningSceneInterface
import time
import copy
import moveit_msgs.msg
from sensor_msgs.msg import JointState
from std_msgs.msg import Float64

class MoveArm():
    def __init__(self):
        rospy.init_node('arm_controller', anonymous=True)
        rospy.Subscriber("/ar_pose_marker", AlvarMarkers, self.callback)
        self.position = None
        self.orientation = None
        self.data = None  # Store data as an instance variable
        moveit_commander.roscpp_initialize(sys.argv)
        scene = PlanningSceneInterface()
        robot = moveit_commander.RobotCommander()
        self.arm_group=moveit_commander.MoveGroupCommander("arm")
        self.gripper_group=moveit_commander.MoveGroupCommander("gripper")
        current_state = robot.get_current_state()
        current_pose = self.arm_group.get_current_pose().pose

        # Print the current pose
        # print("Current End Effector Pose:")
        # print(current_pose)
        # display_trajectory_publisher = rospy.Publisher('/move_group/
              display_planned_path', moveit_msgs.msg.DisplayTrajectory)
        self.joint1_pub = rospy.Publisher
            ('/gripper_finger1_joint/command', Float64, queue_size=10)
        self.joint2_pub = rospy.Publisher
            ('/gripper_finger2_joint/command', Float64, queue_size=10)
        self.arm_group.set_planning_time(10)
        self.grasp_pose=PoseStamped()
        self.gripper_pose=PoseStamped()
        # Initialize ROS node
        self.movearm()
        rospy.spin()

    def callback(self, data):
        self.data = data  # Store data
        for marker in data.markers:
            self.position = marker.pose.pose.position
            self.orientation = marker.pose.pose.orientation
            self.frame_id = marker.header.frame_id
            # print("Marker ID:", marker.id)
            # print("Position :", self.position)
            # print("Orientation :", self.orientation)

    def control_joints(self, joint_positions):
        # Publish joint positions
        self.joint1_pub.publish(joint_positions[0])
        self.joint2_pub.publish(joint_positions[1])

    def movearm(self):
        time.sleep(3)
        if self.position is not None:

            joint_positions = [0.016, 0.016]
            self.control_joints(joint_positions)
            print("AR TAG Position :", self.position)
            print()
            self.grasp_pose.header.frame_id = "base_footprint"
            # position is from marker
            self.grasp_pose.pose.position.x = self.position.x
            self.grasp_pose.pose.position.y = self.position.y + 0.0129
            self.grasp_pose.pose.position.z = self.position.z + 0.0247

            self.grasp_pose.pose.orientation.x = -0.03443147641271983
                                                  #self.orientation.x
            self.grasp_pose.pose.orientation.y = 0.7476661826604609
                                                   #self.orientation.y
            self.grasp_pose.pose.orientation.z = 0.030508461092136 84
                                                   #self.orientation.z
            self.grasp_pose.pose.orientation.w = 0.662479423484054
                                                   #self.orientation.w
            self.arm_group.set_pose_target(self.grasp_pose)
            print("ROBOTIC ARM Position :", self.grasp_pose)
            plan = self.arm_group.plan()
            self.arm_group.execute(plan[1])
            time.sleep(2)
            joint_positions = [0.01, 0.01]
            self.control_joints(joint_positions)
            time.sleep(2)
            self.arm_group.set_named_target("resting")
            self.arm_group.go()

if __name__ == '__main__':
    move_arm = MoveArm()
```

12.4 Appendix D – Python script for following AR-tag and optimized grasping (Placing).

The original code (GitHub Link) was written by the developer; however, necessary changes have been made to include the part of optimized grasping (Placing).

GitHub Link for the given below code is attached:

```
#Name: Nasir Ali
#Date: 02.15.2024
#!/usr/bin/env python3
import math
import sys
import rospy
import moveit_commander
from geometry_msgs.msg import Twist, Vector3, PoseStamped
from nav_msgs.msg import Odometry
from ar_track_alvar_msgs.msg import AlvarMarkers
from moveit_commander import PlanningSceneInterface
import time
import copy
import moveit_msgs.msg
from sensor_msgs.msg import JointState
from std_msgs.msg import Float64

def translate(value, leftMin, leftMax, rightMin, rightMax):
    value = min(max(value, leftMin), leftMax)

    # Figure out how 'wide' each range is
    leftSpan = leftMax - leftMin
    rightSpan = rightMax - rightMin

    # Convert the left range into a 0-1 range (float)
    valueScaled = float(value - leftMin) / float(leftSpan)

    # Convert the 0-1 range into a value in the right range.
    return rightMin + (valueScaled * rightSpan)

class MoveArm():
    def __init__(self):
        self.position = None
        self.orientation = None
        self.data = None  # Store data as an instance variable
        moveit_commander.roscpp_initialize(sys.argv)
        scene = PlanningSceneInterface()
        robot = moveit_commander.RobotCommander()
        self.arm_group=moveit_commander.MoveGroupCommander("arm")
        self.gripper_group=moveit_commander.MoveGroupCommander("gripper")

        self.arm_group.set_planning_time(10)
        self.grasp_pose=PoseStamped()
        self.joint1_pub = rospy.Publisher
          ('/gripper_finger1_joint/command', Float64, queue_size=10)
        self.joint2_pub = rospy.Publisher
```

```
            ('/gripper_finger2_joint/command', Float64, queue_size=10)
        self.grasp_pose=PoseStamped()

    def control_joints(self, joint_positions):
        # Publish joint positions
        self.joint1_pub.publish(joint_positions[0])
        self.joint2_pub.publish(joint_positions[1])

    def movearm(self):
        time.sleep(5)

        # print(self.arm_group.get_current_pose())
        self.grasp_pose.header.frame_id = "base_footprint"
        self.grasp_pose.pose.position.x = 0.3290330861089922
        self.grasp_pose.pose.position.y = -0.01086978096035385
        self.grasp_pose.pose.position.z = 0.30036546454443824

        self.grasp_pose.pose.orientation.x = 0.011299476542224629
        self.grasp_pose.pose.orientation.y = 0.4909875622814981
        self.grasp_pose.pose.orientation.z = -0.020041837152399145
        self.grasp_pose.pose.orientation.w = 0.8708627103500876
        self.arm_group.set_pose_target(self.grasp_pose)

        plan = self.arm_group.plan()

        self.arm_group.execute(plan[1])
        time.sleep(2)

        joint_positions = [0.016, 0.016]
        self.control_joints(joint_positions)
        time.sleep(2)

        joint_positions = [0.01, 0.01]
        self.control_joints(joint_positions)
        time.sleep(2)

        self.arm_group.set_named_target("resting")
        self.arm_group.go()

class ARTagFollower:
    def __init__(self):
        self.last_marker_ts = rospy.Time()
        self.last_marker_position = None
        self.marker_angle = 0.0
        self.marker_distance = 0.0

        self.last_odom_ts = None
        self.odom_position = Vector3()
        self.odom_yaw = 0.0
        self.twist_cmd = Twist()
        self.get_parameters()
        self.cmd_vel_pub = rospy.Publisher("cmd_vel", Twist, queue_size=1)

        self.ar_pose_sub = rospy.Subscriber
            ("ar_pose_marker", AlvarMarkers,
                                self.callback_ar_pose, queue_size=1)

        self.wheel_odom_sub = rospy.Subscriber(
            "wheel_odom_with_covariance",
            Odometry,
            self.callback_wheel_odom,
            queue_size=1,
        )
    def get_parameters(self):
        self.follow_id = rospy.get_param("~follow_id", 5)
        self.marker_timeout =
               rospy.Duration(rospy.get_param("~marker_timeout", 0.5))
        self.min_ang_vel = rospy.get_param("~min_ang_vel", 0.1)
        self.max_ang_vel = rospy.get_param("~max_ang_vel", 1.0)
        self.angle_min = rospy.get_param("~angle_min", 0.1)
        self.angle_max = rospy.get_param("~angle_max", 0.7)
        self.min_lin_vel_forward = rospy.get_param
                                      ("~min_lin_vel_forward", 0.05)
        self.max_lin_vel_forward = rospy.get_param
                                      ("~max_lin_vel_forward", 0.2)
        self.distance_min_forward =
                           rospy.get_param("~distance_min_forward", 0.5)
        self.distance_max_forward =
                           rospy.get_param("~distance_max_forward", 2.0)
        self.min_lin_vel_reverse =
                           rospy.get_param("~min_lin_vel_reverse", 0.05)
        self.max_lin_vel_reverse = rospy.get_param
                                      ("~max_lin_vel_reverse", 0.2)
        self.distance_min_reverse =
                          rospy.get_param("~distance_min_reverse", 0.5)
        self.distance_max_reverse =
                           rospy.get_param("~distance_max_reverse", 2.0)

    def run(self):
        rate = rospy.Rate(10)
        while not rospy.is_shutdown():
            self.update_cmd()
            self.cmd_vel_pub.publish(self.twist_cmd)
            rate.sleep()

    def update_cmd(self):
        # Check for a timeout
        if self.last_marker_ts + self.marker_timeout < rospy.get_rostime():
            self.twist_cmd.linear.x = 0.0
            self.twist_cmd.angular.z = 0.0
            return
        # Get the absolute angle to the marker
        angle = math.fabs(self.marker_angle)
        # Get the direction multiplier
        dir = -1.0 if self.marker_angle < 0.0 else 1.0

        # Calculate angular command
        if angle < self.angle_min:
            ang_cmd = 0.0
        else:
            ang_cmd = translate(
                angle,
                self.angle_min,
                self.angle_max,
                self.min_ang_vel,
                self.max_ang_vel,
            )
        # Calculate linear command
        if self.marker_distance >= self.distance_min_forward:
            lin_cmd = translate(
                self.marker_distance,
                self.distance_min_forward,
                self.distance_max_forward,
                self.min_lin_vel_forward,
                self.max_lin_vel_forward,
            )
        elif self.marker_distance <= self.distance_max_reverse:
            lin_cmd = -translate(
                self.marker_distance,
                self.distance_min_reverse,
                self.distance_max_reverse,
                self.max_lin_vel_reverse,
                self.min_lin_vel_reverse,
            )
        else:
            lin_cmd = 0.0
            move_arm = MoveArm()
            move_arm.movearm()
            return

        self.twist_cmd.angular.z = dir * ang_cmd
        self.twist_cmd.linear.x = lin_cmd

    def update_marker_angle_distance(self):
        if self.last_marker_position:
            position_x = self.last_marker_position.x - self.odom_position.x
            position_y = self.last_marker_position.y - self.odom_position.y
            self.marker_angle = math.atan(position_y / position_x) -
                                    self.odom_yaw

            self.marker_distance = math.sqrt(
                   position_x * position_x + position_y * position_y  )

    def callback_ar_pose(self, msg):
        for marker in msg.markers:
            if marker.id != self.follow_id:
                continue
            if marker.header.stamp < self.last_marker_ts:
                rospy.logwarn_throttle(
                    3.0, "Got marker position with an older timestamp"
                )
                continue

            self.last_marker_ts = marker.header.stamp
            self.last_marker_position = marker.pose.pose.position
            self.odom_position = Vector3()
            self.odom_yaw = 0.0

            self.update_marker_angle_distance()

    def callback_wheel_odom(self, msg):
        if self.last_odom_ts:
            start_ts = max(self.last_odom_ts, self.last_marker_ts)

            end_ts = msg.header.stamp
            if end_ts < start_ts:
                rospy.logwarn
            ("Reveived odometry has timestamp older than last marker
            position"  )

            step_duration = (end_ts - start_ts).to_sec()

            # Integrate the velocity using rectangular rule
            self.odom_yaw += msg.twist.twist.angular.z * step_duration
            self.odom_position.x += (
                msg.twist.twist.linear.x * math.cos(self.odom_yaw) *
                step_duration)
            self.odom_position.y += (
                msg.twist.twist.linear.x * math.sin(self.odom_yaw) *
                step_duration)
            self.update_marker_angle_distance()
        self.last_odom_ts = msg.header.stamp
if __name__ == "__main__":
    rospy.init_node("follow_ar_tag")
    ar_tag_follower = ARTagFollower()
    ar_tag_follower.run()
```

12.5 Appendix E – Configuration setup with rover and laptop

12.5.1 ROS1 installation guide for Linux operating system on Laptop

1. Download and install Ubuntu 20.04.6 LTS (Focal Fossa). Create a bootable USB drive and follow the installation process.
2. Install ROS Noetic and required ROS packages on laptop. It allows to communicate with nodes running on Leo rover which can easily launch graphical interface and visualization tools. The following command refer to Linux terminal and shows the installation process:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
/etc/apt/sources.list.d/ros-latest.list'
$ sudo apt install curl
$ curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -
$ sudo apt update
$ sudo apt install ros-noetic-desktop-full
$ sudo apt install python3-rosdep
$ sudo rosdep init
$ rosdep update
```

   It is important to source the *bash terminal* to setup the ROS environment. The following command automatically source the *bash terminal*. There will no need to source the terminal every time after editing */.bashrc* file.

```
$ vi ~/.bashrc
source /opt/ros/noetic/setup.bash
source ~/ros_ws/devel/setup.bash
```

3. Installing required ROS package for the rover:

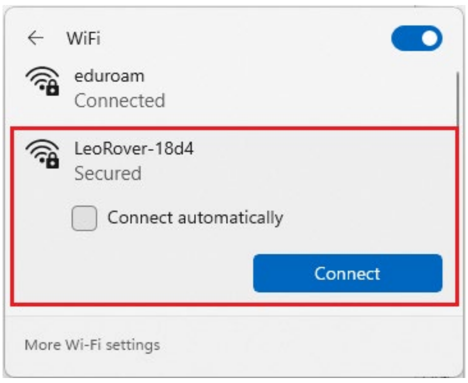
```
$sudo apt update
$sudo apt install ros-neotic-leo-viz
$sudo apt install ros-neotic-leo_description
$sudo apt install ros-neotic-rplidar-ros
$sudo apt install ros-noetic-moveit
$sudo apt install ros-neotic-rqt-graph
$sudo apt install ros-noetic-leo-examples
$sudo apt install ros-noetic-ar-track-alvar
$sudo apt install python3-pip
$pip3 install tflite-runtime
```

12.5.2 *LeoOS* installation for Leo Rover

1. Download the newest LeoOS image, LeoOS-1.2.0-2023-11-02-full.img.xz, and flash Raspberry pi image to microSD card (minimum 32 GB) using Etcher bootable software.
2. After the flashing completes, disconnect the microSD card and put it back into the rover. To access the microSD to Leo Rover, open the main electronics box by unscrewing the 4 socket-headed screws and insert it to the SD card slot on the Raspberry Pi.

12.5.3 Connect to Leo Rover AP

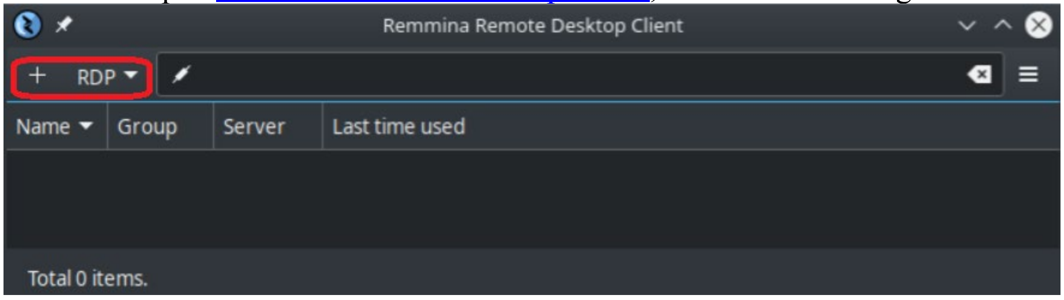
1. Turn ON the rover using its main power button located on the left side of the battery. The LED on the button should start blinking green. After approximately 30 seconds later, the LED should stop blinking and the Rover should be operational.
2. When the light stops blinking, check the host device for the WiFi network on Laptop. The default credential start from LeoRover-XXXX, but instead of XXXX keyword, it will appear unique identifies to the rover's computer, shown in given below figure. To connect with the SSID (LeoRover-18d4), use the default password "password".



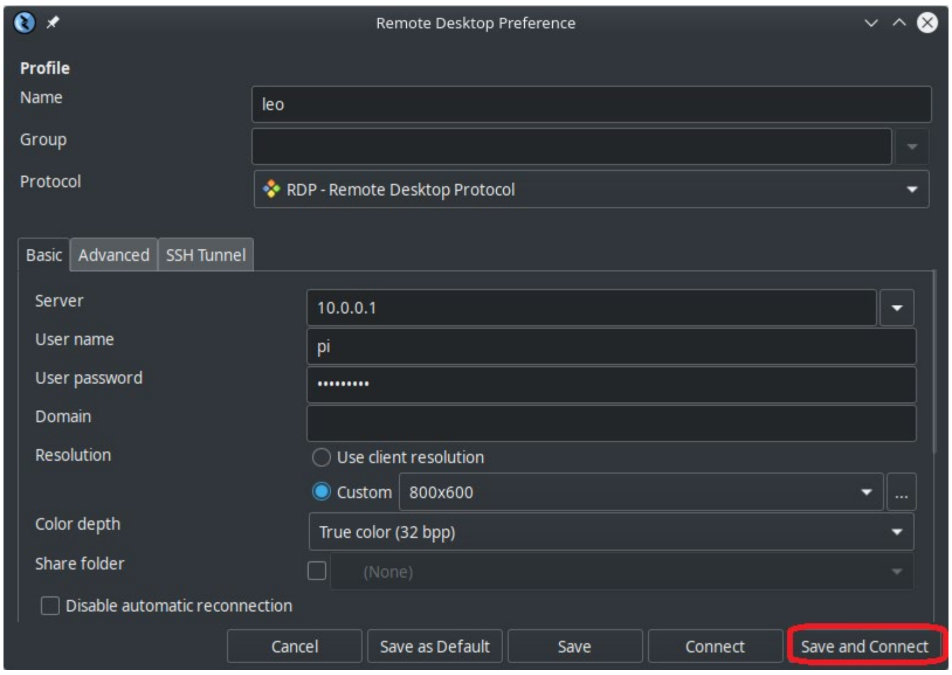
12.5.4 Connect via remote desktop

In the Leo Rover, full version of LeoOS is installed, which have desktop environments installed, as well as RDP (Microsoft Remote Desktop Protocol) server. It allows to remotely control desktop environment on Rover from laptop. Rover can be remotely controlled by all operating systems, Windows, Linux, Mac. For this session Linux operating system is used to install remote desktop.

1. Install and open [Remmina Remote Desktop Client](), shown in below figure:



2. Click on the + icon and fill the following fields:
   - Server: 10.0.0.1
   - Username: pi
   - User password: raspberry
   - Color depth: True color (32 bpp)
   - (Optional) Name - if you want to save the settings.
   - (Optional) Resolution - If you want to use a custom resolution.
3. After filling the setting, click on **Save and Connect,** shown in below figure:



Remote Desktop Preference

Profile

Name: leo

Group:

Protocol: RDP - Remote Desktop Protocol

Basic | Advanced | SSH Tunnel

Server: 10.0.0.1

User name: pi

User password: •••••••••

Domain:

Resolution: ( ) Use client resolution [✓ Custom] 800x600

Color depth: True color (32 bpp)

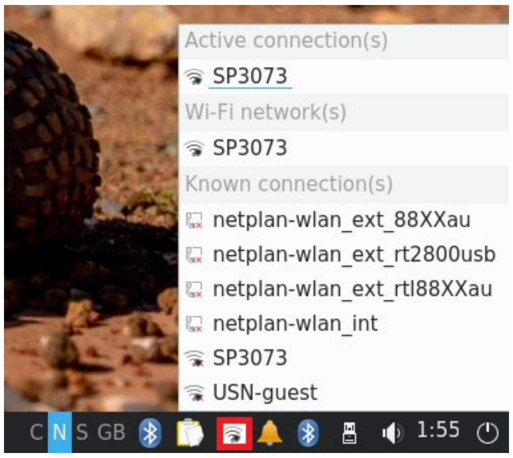
Share folder: ☐ (None)

☐ Disable automatic reconnection

12.5.5 Connect to a local network and the internet using remote desktop

By connecting the Rover to available local network, it provides Internet connection, and allows to download files to the Rover. The LeoOS uses NetworkManager to manage the Raspberry Pi's internal Wifi. The Wifi interface can connect to both 2.4 GHz and 5 GHz networks. The easiest way to connect the internet, use remote desktop.

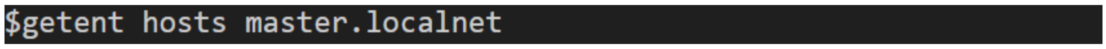
1. Click on the **NetworkManager** applet icon on the system tray, choose available network and type the password and connected it. In the below figure, available network is connected with the rover's computer,.



12.5.6 Connecting ROS network to the laptop

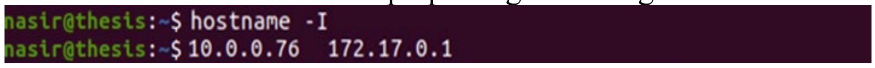
Configuring laptop with ROS network allows to run nodes that interfere with Leo Rover's hardware, as well as graphical tools (like rqt or rviz) directly on your host machine.

1. Connect the laptop to the same network of rover is connected to, using Rover's Access Point (LeoRover-18d4).
2. To properly communicate over the ROS network, laptop needs to be able to resolve the *master.localnet* hostname. Open a terminal on the laptop and type the following commands:



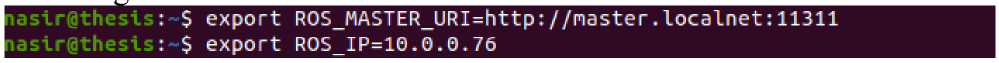
```
$getent hosts master.localnet
```

   Note: If you don't see any output, that means you cannot resolve the hostname.
3. Check the IP address of the laptop using following command.



```
nasir@thesis:~$ hostname -I
nasir@thesis:~$ 10.0.0.76  172.17.0.1
```

4. Specify the address of the rover's master node and laptop IP address, by using the following commands:



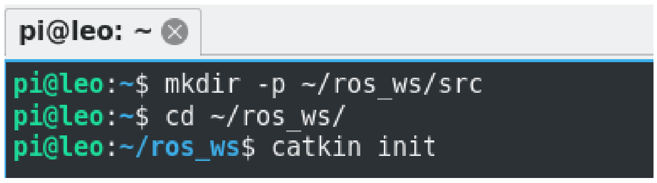
```
nasir@thesis:~$ export ROS_MASTER_URI=http://master.localnet:11311
nasir@thesis:~$ export ROS_IP=10.0.0.76
```

5. After this following the above steps, all ROS nodes will available on laptop which being running on Leo Rover, easy to visualize the Rviz tool for simulation.

12.5.7 Building necessary ROS packages

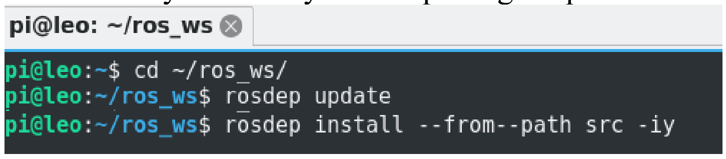
In Leo Rover, ROS uses its own build system for building packages. These packages are the main unit for organizing software in ROS. It lies on catkin workspace (catkin_ws) which built as a standalone project, but it also allows to create own workspaces depending on project requirements. To extend the ROS distribution, it is recommended to create empty workspace named ros_ws inside the home directory on Raspberry Pi.

1. To create empty workspace ros_ws, use the following command inside the RPi terminal.



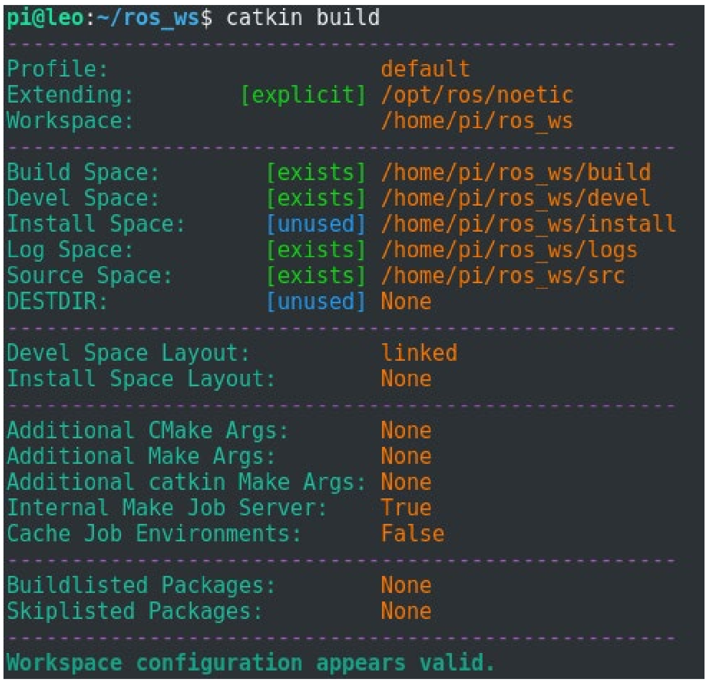
```
pi@leo: ~

pi@leo:~$ mkdir -p ~/ros_ws/src
pi@leo:~$ cd ~/ros_ws/
pi@leo:~/ros_ws$ catkin init
```

2. Some of the packages will require installing additional dependencies to build and run them. As the *leo_bringup* package is already installed on the system, this step is redundant. For any other package, it is recommended to install *rosdep* which will automatically install any related package dependencies:



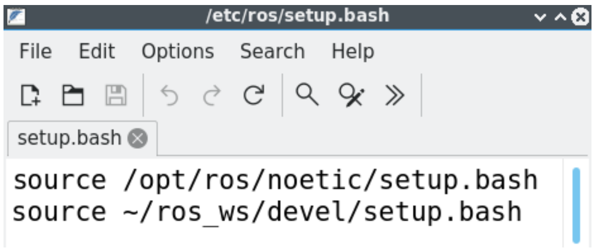
```
pi@leo: ~/ros_ws

pi@leo:~$ cd ~/ros_ws/
pi@leo:~/ros_ws$ rosdep update
pi@leo:~/ros_ws$ rosdep install --from--path src -iy
```

3. Build the workspace using the following command, shown in below figure. If workspace properly built, it would appear similar to like this.



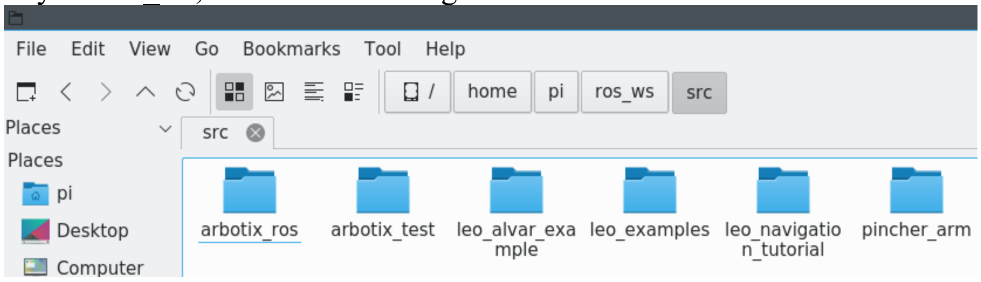
```
pi@leo:~/ros_ws$ catkin build
----------------------------------------------------------------
Profile:                     default
Extending:        [explicit] /opt/ros/noetic
Workspace:                   /home/pi/ros_ws
----------------------------------------------------------------
Build Space:        [exists] /home/pi/ros_ws/build
Devel Space:        [exists] /home/pi/ros_ws/devel
Install Space:      [unused] /home/pi/ros_ws/install
Log Space:          [exists] /home/pi/ros_ws/logs
Source Space:       [exists] /home/pi/ros_ws/src
DESTDIR:            [unused] None
----------------------------------------------------------------
Devel Space Layout:          linked
Install Space Layout:        None
----------------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
Internal Make Job Server:    True
Cache Job Environments:      False
----------------------------------------------------------------
Buildlisted Packages:        None
Skiplisted Packages:         None
----------------------------------------------------------------
Workspace configuration appears valid.
```

4. If everything works smoothly, a development space should be created inside the devel directory. The last step is to modify the */etc/ros/setup.bash* to use the overlay. Simply edit this file (e.g. with nano) by adding the source file, where necessary packages are going to installed.

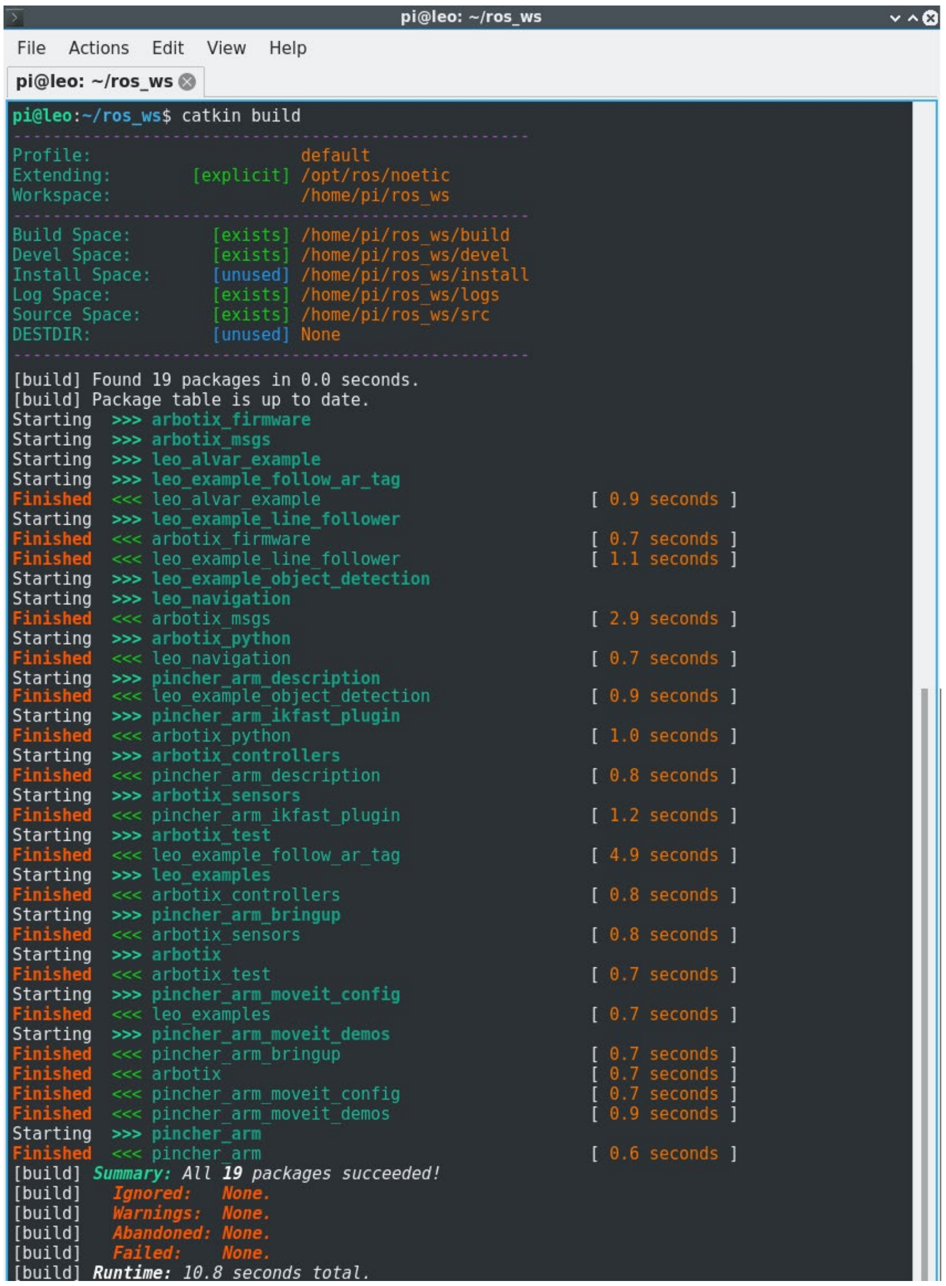


```
source /opt/ros/noetic/setup.bash
source ~/ros_ws/devel/setup.bash
```

5. After building development space in */ros_ws* directory of RPi computer. All necessary packages required to clone using GitHub links in the directory.
6. The GitHub links are attached to the packages name, including *arbotix_ros*, *leo_alvar_example*, *leo_example*, *leo_navigation_tutorial*, *pincher_arm* and *rplidar_ros*. User needs to clone GitHub links using the following command (as a sample) git clone https://github.com/LeoRover/leo_navigation_tutorial.git inside /src directory of /ros_ws, shown in below figure:

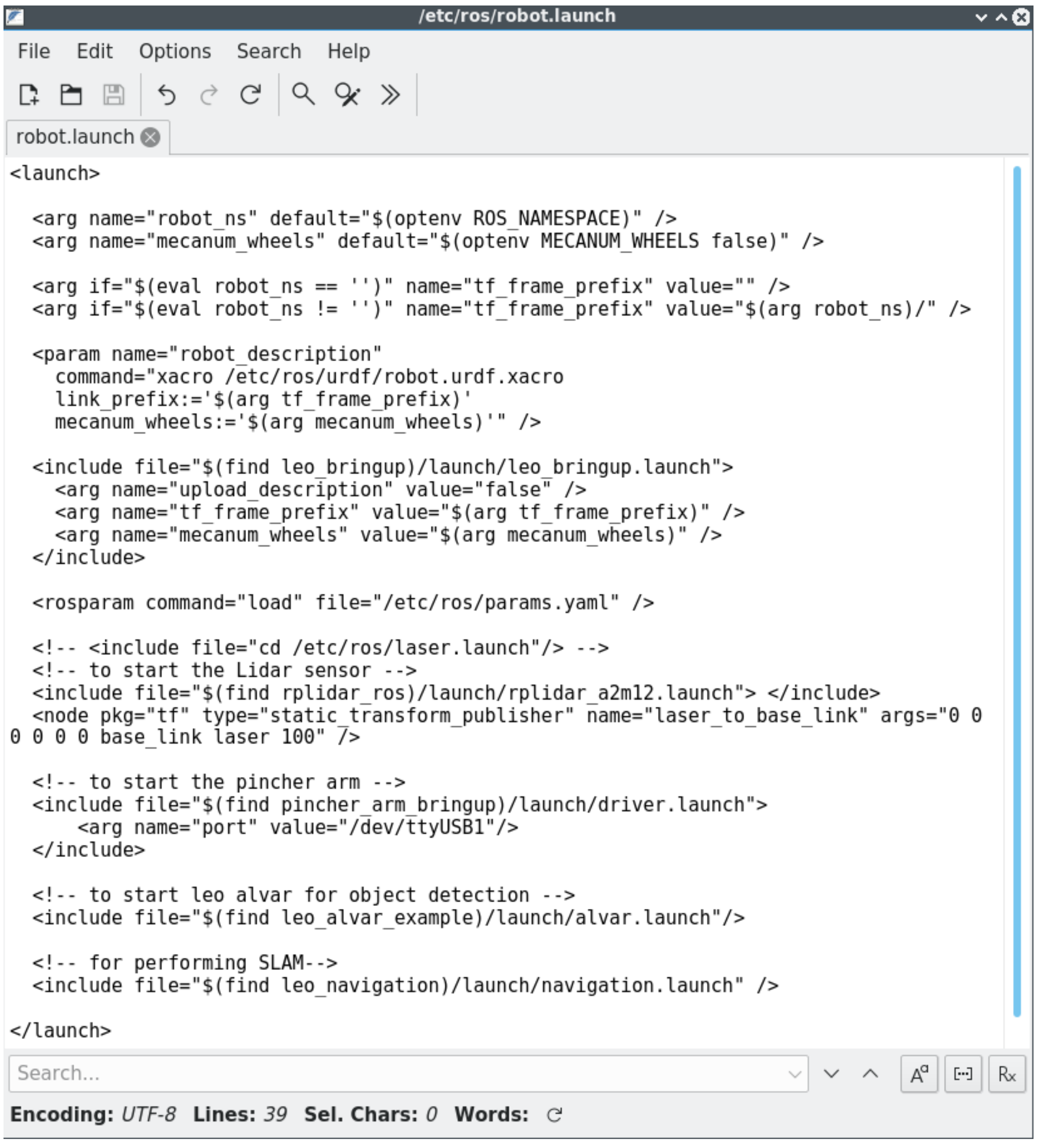


7. After cloning all required ROS package, it needs to build the workspace using command catin buit, it looks similar to below figure:



```
pi@leo:~/ros_ws$ catkin build
--------------------------------------------------------
Profile:                     default
Extending:          [explicit] /opt/ros/noetic
Workspace:                   /home/pi/ros_ws
--------------------------------------------------------
Build Space:          [exists] /home/pi/ros_ws/build
Devel Space:          [exists] /home/pi/ros_ws/devel
Install Space:        [unused] /home/pi/ros_ws/install
Log Space:            [exists] /home/pi/ros_ws/logs
Source Space:         [exists] /home/pi/ros_ws/src
DESTDIR:              [unused] None
--------------------------------------------------------

[build] Found 19 packages in 0.0 seconds.
[build] Package table is up to date.
Starting  >>> arbotix_firmware
Starting  >>> arbotix_msgs
Starting  >>> leo_alvar_example
Starting  >>> leo_example_follow_ar_tag
Finished  <<< leo_alvar_example                         [ 0.9 seconds ]
Starting  >>> leo_example_line_follower
Finished  <<< arbotix_firmware                          [ 0.7 seconds ]
Finished  <<< leo_example_line_follower                 [ 1.1 seconds ]
Starting  >>> leo_example_object_detection
Starting  >>> leo_navigation
Finished  <<< arbotix_msgs                              [ 2.9 seconds ]
Starting  >>> arbotix_python
Finished  <<< leo_navigation                            [ 0.7 seconds ]
Starting  >>> pincher_arm_description
Finished  <<< leo_example_object_detection              [ 0.9 seconds ]
Starting  >>> pincher_arm_ikfast_plugin
Finished  <<< arbotix_python                            [ 1.0 seconds ]
Starting  >>> arbotix_controllers
Finished  <<< pincher_arm_description                   [ 0.8 seconds ]
Starting  >>> arbotix_sensors
Finished  <<< pincher_arm_ikfast_plugin                 [ 1.2 seconds ]
Starting  >>> arbotix_test
Finished  <<< leo_example_follow_ar_tag                 [ 4.9 seconds ]
Starting  >>> leo_examples
Finished  <<< arbotix_controllers                       [ 0.8 seconds ]
Starting  >>> pincher_arm_bringup
Finished  <<< arbotix_sensors                           [ 0.8 seconds ]
Starting  >>> arbotix
Finished  <<< arbotix_test                              [ 0.7 seconds ]
Starting  >>> pincher_arm_moveit_config
Finished  <<< leo_examples                              [ 0.7 seconds ]
Starting  >>> pincher_arm_moveit_demos
Finished  <<< pincher_arm_bringup                       [ 0.7 seconds ]
Finished  <<< arbotix                                   [ 0.7 seconds ]
Finished  <<< pincher_arm_moveit_config                 [ 0.7 seconds ]
Finished  <<< pincher_arm_moveit_demos                  [ 0.9 seconds ]
Starting  >>> pincher_arm
Finished  <<< pincher_arm                               [ 0.6 seconds ]
[build] Summary: All 19 packages succeeded!
[build]   Ignored:   None.
[build]   Warnings:  None.
[build]   Abandoned: None.
[build]   Failed:    None.
[build] Runtime: 10.8 seconds total.
```

8. The last step to include these packages launch files *robot.launch* file, available at */etc/ros/* directory, will start theirs ROS nodes at boot using *leo-start* on terminal . All including launch files, it should be like the given below figure:


/etc/ros/robot.launch
File Edit Options Search Help
robot.launch

```
<launch>

  <arg name="robot_ns" default="$(optenv ROS_NAMESPACE)" />
  <arg name="mecanum_wheels" default="$(optenv MECANUM_WHEELS false)" />

  <arg if="$(eval robot_ns == '')" name="tf_frame_prefix" value="" />
  <arg if="$(eval robot_ns != '')" name="tf_frame_prefix" value="$(arg robot_ns)/" />

  <param name="robot_description"
    command="xacro /etc/ros/urdf/robot.urdf.xacro
    link_prefix:='$(arg tf_frame_prefix)'
    mecanum_wheels:='$(arg mecanum_wheels)'" />

  <include file="$(find leo_bringup)/launch/leo_bringup.launch">
    <arg name="upload_description" value="false" />
    <arg name="tf_frame_prefix" value="$(arg tf_frame_prefix)" />
    <arg name="mecanum_wheels" value="$(arg mecanum_wheels)" />
  </include>

  <rosparam command="load" file="/etc/ros/params.yaml" />

  <!-- <include file="cd /etc/ros/laser.launch"/> -->
  <!-- to start the Lidar sensor -->
  <include file="$(find rplidar_ros)/launch/rplidar_a2m12.launch"> </include>
  <node pkg="tf" type="static_transform_publisher" name="laser_to_base_link" args="0 0
0 0 0 0 base_link laser 100" />

  <!-- to start the pincher arm -->
  <include file="$(find pincher_arm_bringup)/launch/driver.launch">
      <arg name="port" value="/dev/ttyUSB1"/>
  </include>

  <!-- to start leo alvar for object detection -->
  <include file="$(find leo_alvar_example)/launch/alvar.launch"/>

  <!-- for performing SLAM-->
  <include file="$(find leo_navigation)/launch/navigation.launch" />

</launch>
```

Search...
Encoding: UTF-8 Lines: 39 Sel. Chars: 0 Words:

12.6 Appendix F – QuickStart Guide for Leo Rover

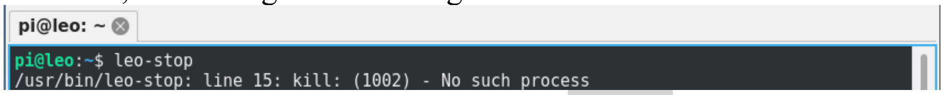
12.6.1 Prerequisites requirements:

- Ubuntu Linux 20.04 should run to the laptop.
- ROS1 Noetic and necessary package should installed to the rover.
- Lidar sensor, robotic arm and RPi camera should available and integrated with rover ROS network.

12.6.2 Procedure:

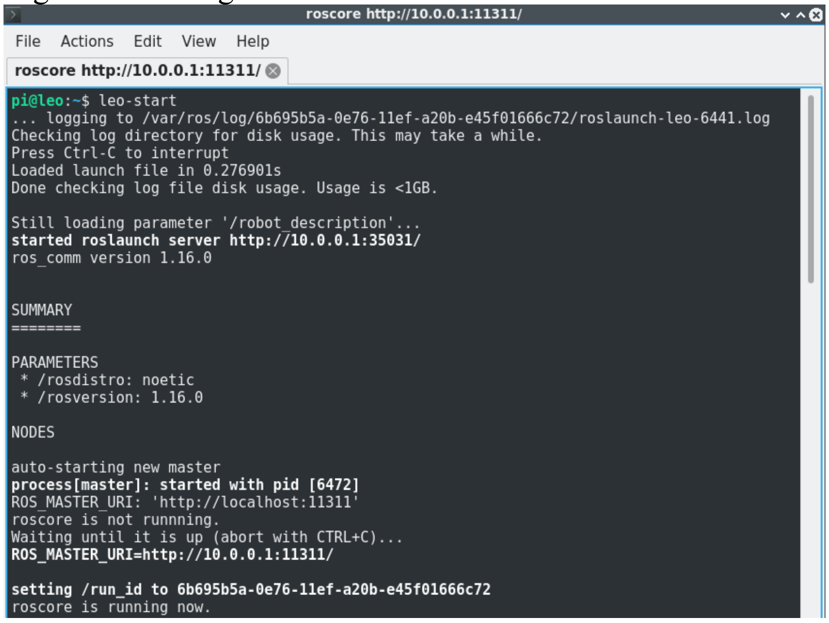
Follow the steps subsequently in the given order. All commands throughout the guide means to be type on Ubuntu terminal on laptop and rover's computer terminal.

1. Turn ON the rover using its main power button located on the left side of the battery.
2. Connect to rover's access point (LeoRover-18d4) with laptop.
3. If the rover is successfully connected with the rover's access point, Lidar sensor start rotating and robotic arm Arbotix controller's light start blinking.
   Note: It means that, rover's ROS *nodes* automatically are started. However, it is recommended to start the rover manually for proper functionalities.
4. Open the Remmina Remote Desktop Client from the laptop and click on saved rover's name.
   a. Open the RPi terminal and stop the rover using the leo-stop command on terminal, shown in given below figure:



```
pi@leo:~$ leo-stop
/usr/bin/leo-stop: line 15: kill: (1002) - No such process
```

   b. Now, to start the *nodes* manually using the leo-start command on terminal, shown in given below figure:



roscore http://10.0.0.1:11311/
File
Actions
Edit
View
Help
roscore http://10.0.0.1:11311/

```
pi@leo:~$ leo-start
... logging to /var/ros/log/6b695b5a-0e76-11ef-a20b-e45f01666c72/roslaunch-leo-6441.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Loaded launch file in 0.276901s
Done checking log file disk usage. Usage is <1GB.

Still loading parameter '/robot_description'...
started roslaunch server http://10.0.0.1:35031/
ros_comm version 1.16.0


SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.16.0

NODES

auto-starting new master
process[master]: started with pid [6472]
ROS_MASTER_URI: 'http://localhost:11311'
roscore is not runnning.
Waiting until it is up (abort with CTRL+C)...
ROS_MASTER_URI=http://10.0.0.1:11311/

setting /run_id to 6b695b5a-0e76-11ef-a20b-e45f01666c72
roscore is running now.
```

Note: Press Ctrl+C is used to stop the *nodes* and exit the script. After starting *nodes* manually, the Lidar sensor start rotating and available for performing autonomous navigation. Simultaneously, robotic arm controller's LED light start blinking and available for performing grasping AR-Tags based small objects.

5. Connect ROS network to the laptop for visualization of Leo Rover in simulated environment, type the following commands on laptop Ubuntu terminal.

```
nasir@thesis:~$ export ROS_MASTER_URI=http://master.localnet:11311
nasir@thesis:~$ export ROS_IP=10.0.0.76
```

Note: All ROS nodes and topics are available on laptop which are currently running on rover's RPi computer.

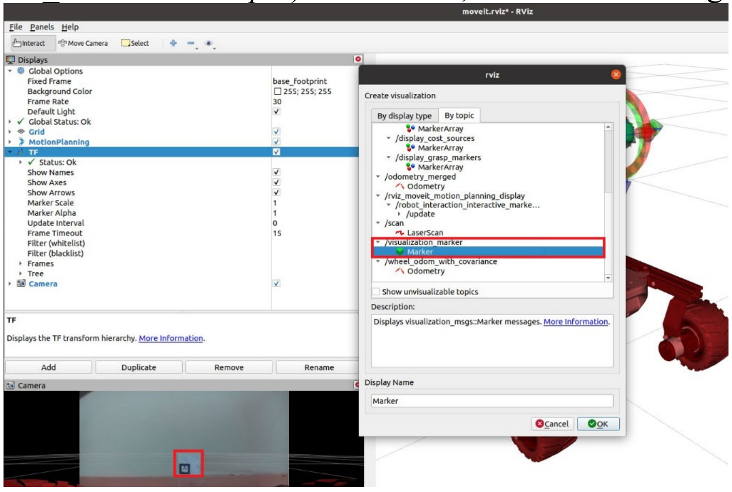
a. Type the following commands to test the autonomous navigation on Rviz simulation and physical environments for the Leo Rover.

```
nasir@thesis:~$ roslaunch leo_viz rviz.launch config:=navigation
```

b. Type the following commands to test the rover's arm using Motion Planning on Rviz simulation and physical environments.

```
nasir@thesis:~$ roslaunch pincher_arm_moveit_config pincher_arm_moveit.launch
```

c. To test the real-time object detection, activate available Marker (*/visualization_marker ROS topic*) in Rviz tool, shown in below figure.



Global Options
Fixed Frame base_footprint
Background Color 255; 255; 255
Frame Rate 30
Default Light ✓
Global Status: Ok
Grid ✓
MotionPlanning ✓
TF ✓
Status: Ok
Show Names ✓
Show Axes ✓
Show Arrows ✓
Marker Scale 1
Marker Alpha 1
Update Interval 0
Frame Timeout 15
Filter (whitelist)
Filter (blacklist)
Frames
Tree
Camera ✓

TF

Displays the TF transform hierarchy. More Information.

MarkerArray
/display_cost_sources
MarkerArray
/display_grasp_markers
MarkerArray
/odometry_merged
Odometry
/rviz_moveit_motion_planning_display
/robot_interaction_interactive_marke...
/update
/scan
LaserScan
/visualization_marker
Marker
/wheel_odom_with_covariance
Odometry

Show unvisualizable topics

Description:

Displays visualization_msgs::Marker messages. More Information.

Display Name

Marker

d. To find the exact position of AR-tag and arm' end-effector, run the python code mentioned in Appendix B – Python script for testing both AR-tag and end-effector pose.

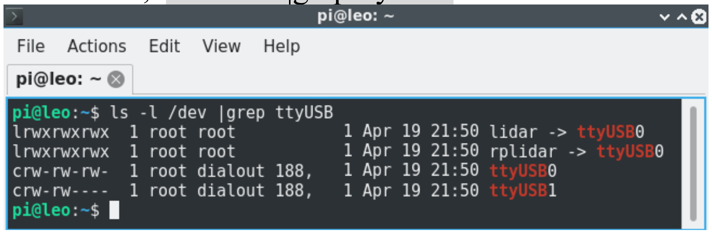
e. To perform the optimized grasping (picking) an AR-tag based small object, run the python code mentioned in Appendix C – Python script for optimized grasping (Picking).

f. To perform placing grasped small objects, run the python code mentioned in Appendix D – Python script for following AR-tag and optimized grasping (Placing)..

The QuickStart guide for the Leo Rover is now completed and available at USN, Porsgrun campus for testing in both simulated and physical environment.

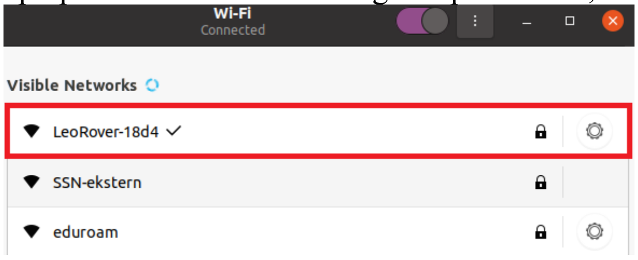
12.7 Appendix G – Testing the hardware and software for the rover

1. **Setup and Prerequisites**
   Ensure the following steps:
   a. Lidar sensor and robotic arm connected via USB to the Raspberry Pi (RPi).
   b. ZED2 stereo camera is connected to additional computer vis USB cable and the Ethernet cable is connected to both RPi board and additional computer.
   c. Basic familiarity with Linux command line and ROS.
2. **Hardware verfications**
   a. Power ON the rover from the left side of power box.
      i. Ensure the Rpi, Lidar sensor, robotic arm and ZED2 camera are properly connected and powered.
      ii. Verfiy that RPi board, Lidar sensor connection, arm's controller and ZED2 indicate LEDs ON.
   b. **Connection verfication**
      i. Check the connected USBs deviced to the RPi board, as following commands, ls -l /dev |grep ttyUSB.

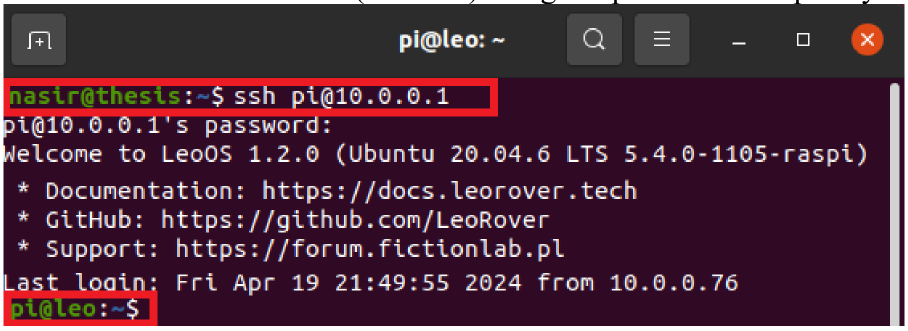


```
pi@leo:~$ ls -l /dev |grep ttyUSB
lrwxrwxrwx  1 root root           1 Apr 19 21:50 lidar -> ttyUSB0
lrwxrwxrwx  1 root root           1 Apr 19 21:50 rplidar -> ttyUSB0
crw-rw-rw-  1 root dialout 188,   1 Apr 19 21:50 ttyUSB0
crw-rw----  1 root dialout 188,   1 Apr 19 21:50 ttyUSB1
pi@leo:~$
```

3. **Software setup:**
   a. Connect your laptop to the rover's AP using its «passowrd», as follow:



   b. **Connect the baord via SSH connection**
      i. Use SSH to access the RPi (10.0.0.1) using the password «raspberry»

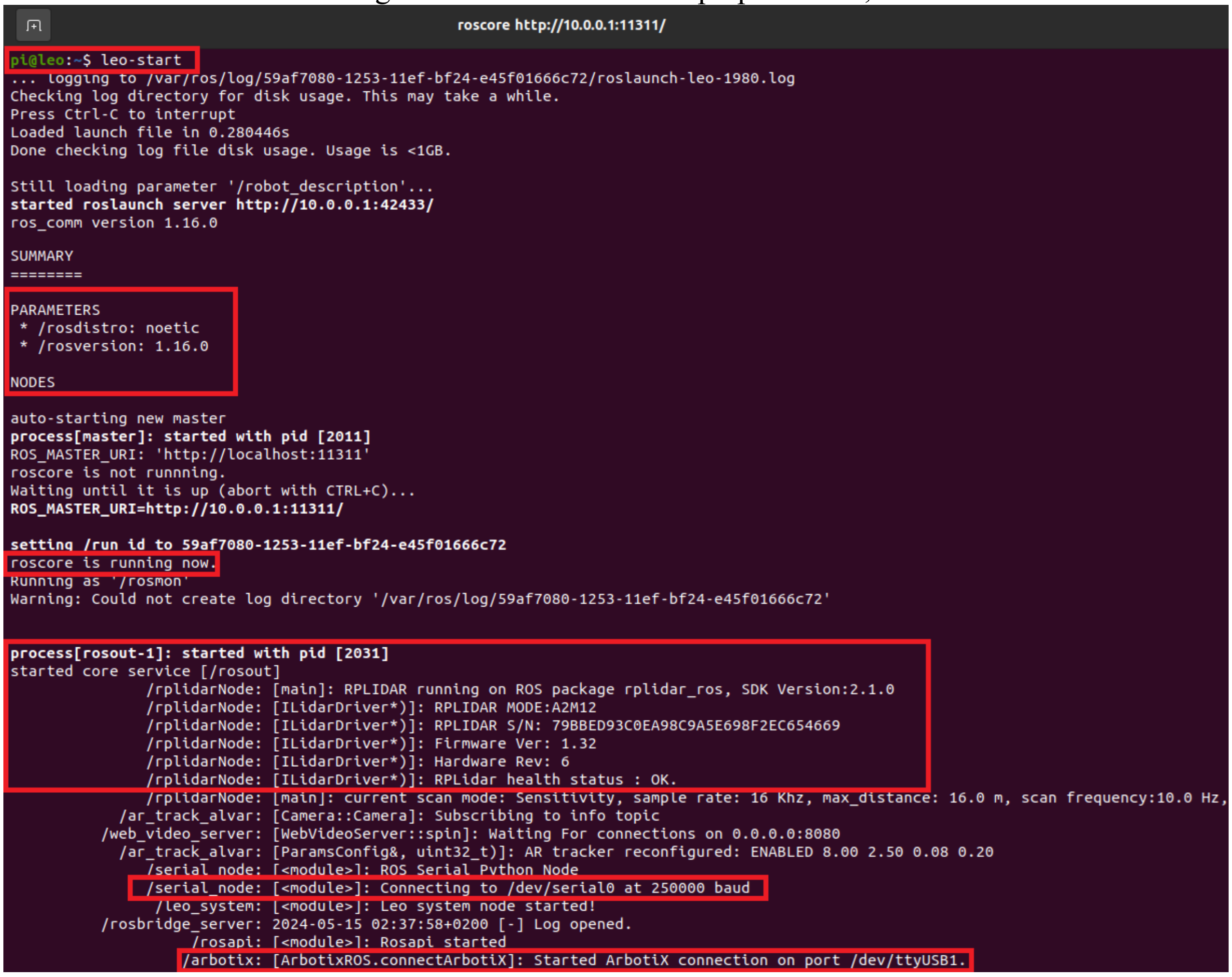


```
nasir@thesis:~$ ssh pi@10.0.0.1
pi@10.0.0.1's password:
Welcome to LeoOS 1.2.0 (Ubuntu 20.04.6 LTS 5.4.0-1105-raspi)

 * Documentation: https://docs.leorover.tech
 * GitHub: https://github.com/LeoRover
 * Support: https://forum.fictionlab.pl

Last login: Fri Apr 19 21:49:55 2024 from 10.0.0.76
pi@leo:~$
```

4. **Installing required ROS Packages**

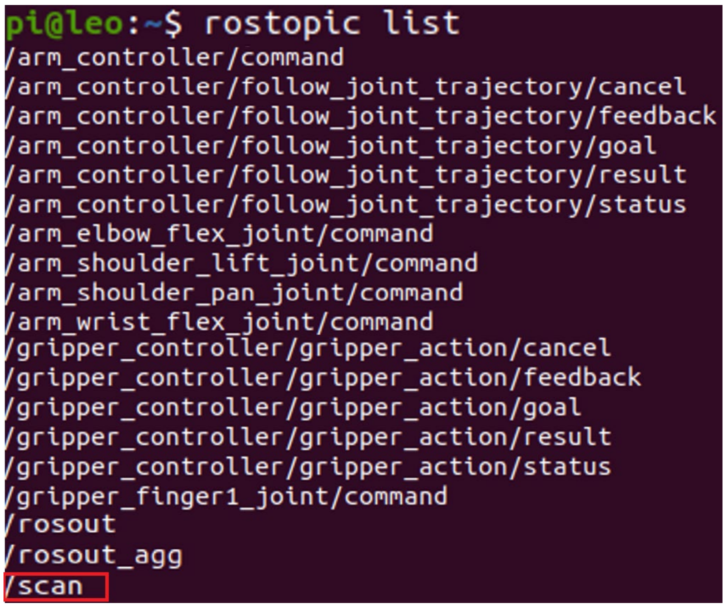
   a. Follow the Instalation process, discussed in appendix section 12.5.7.

5. **Testing ROS *nodes* and *topics* for Lidar**
   a. Start the rover using leo-start command on laptop terminal, as follows:



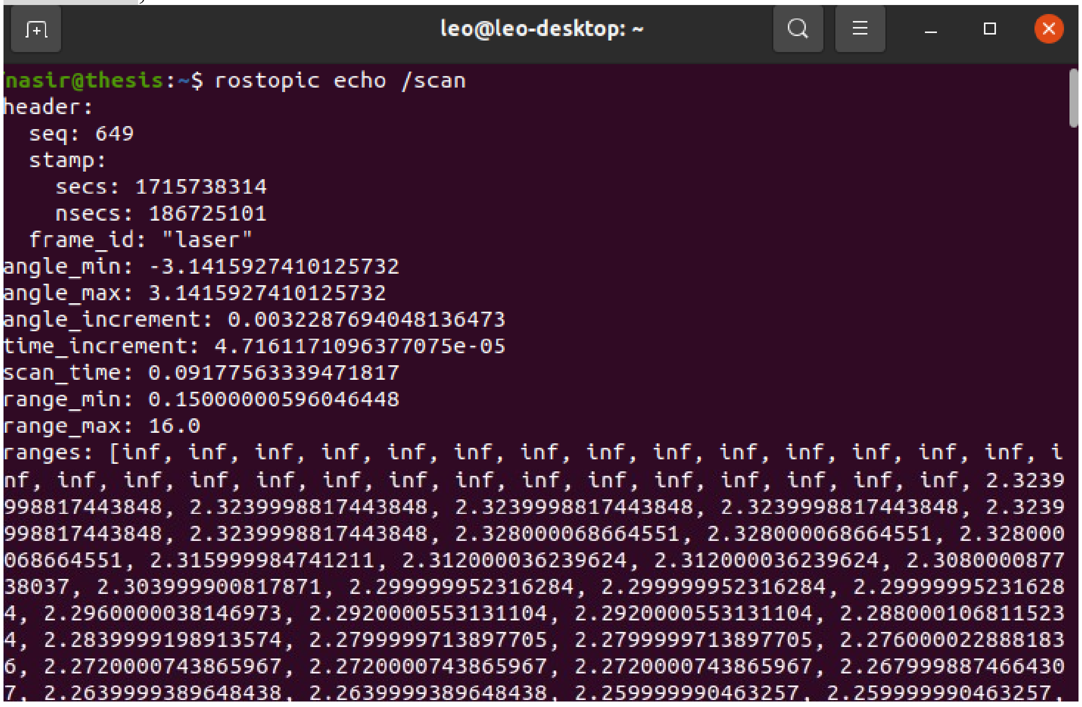
```
roscore http://10.0.0.1:11311/

pi@leo:~$ leo-start
... logging to /var/ros/log/59af7080-1253-11ef-bf24-e45f01666c72/roslaunch-leo-1980.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Loaded launch file in 0.280446s
Done checking log file disk usage. Usage is <1GB.

Still loading parameter '/robot_description'...
started roslaunch server http://10.0.0.1:42433/
ros_comm version 1.16.0

SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.16.0

NODES

auto-starting new master
process[master]: started with pid [2011]
ROS_MASTER_URI: 'http://localhost:11311'
roscore is not runnning.
Waiting until it is up (abort with CTRL+C)...
ROS_MASTER_URI=http://10.0.0.1:11311/

setting /run id to 59af7080-1253-11ef-bf24-e45f01666c72
roscore is running now.
Running as '/rosmon'
Warning: Could not create log directory '/var/ros/log/59af7080-1253-11ef-bf24-e45f01666c72'


process[rosout-1]: started with pid [2031]
started core service [/rosout]
               /rplidarNode: [main]: RPLIDAR running on ROS package rplidar_ros, SDK Version:2.1.0
               /rplidarNode: [ILidarDriver*)]: RPLIDAR MODE:A2M12
               /rplidarNode: [ILidarDriver*)]: RPLIDAR S/N: 79BBED93C0EA98C9A5E698F2EC654669
               /rplidarNode: [ILidarDriver*)]: Firmware Ver: 1.32
               /rplidarNode: [ILidarDriver*)]: Hardware Rev: 6
               /rplidarNode: [ILidarDriver*)]: RPLidar health status : OK.
               /rplidarNode: [main]: current scan mode: Sensitivity, sample rate: 16 Khz, max_distance: 16.0 m, scan frequency:10.0 Hz,
            /ar_track_alvar: [Camera::Camera]: Subscribing to info topic
          /web_video_server: [WebVideoServer::spin]: Waiting For connections on 0.0.0.0:8080
            /ar_track_alvar: [ParamsConfig&, uint32_t)]: AR tracker reconfigured: ENABLED 8.00 2.50 0.08 0.20
               /serial_node: [<module>]: ROS Serial Python Node
               /serial_node: [<module>]: Connecting to /dev/serial0 at 250000 baud
                /leo_system: [<module>]: Leo system node started!
          /rosbridge_server: 2024-05-15 02:37:58+0200 [-] Log opened.
                    /rosapi: [<module>]: Rosapi started
                   /arbotix: [ArbotixROS.connectArbotiX]: Started ArbotiX connection on port /dev/ttyUSB1.
```

   b. After starting the rover, its ROS *nodes* and *topics* are activated and available, as it shown in red-color boxes in section a.
   c. If the ROS *nodes* are succefully activated, the Lidar sensor start rotating and the robotic arm Arbotix controller's LED ligth start blinking continoesly.
   d. To test the availabe ROS *nodes* and *topics*, use the following command rostopic list, as shown in below figure.



```
pi@leo:~$ rostopic list
/arm_controller/command
/arm_controller/follow_joint_trajectory/cancel
/arm_controller/follow_joint_trajectory/feedback
/arm_controller/follow_joint_trajectory/goal
/arm_controller/follow_joint_trajectory/result
/arm_controller/follow_joint_trajectory/status
/arm_elbow_flex_joint/command
/arm_shoulder_lift_joint/command
/arm_shoulder_pan_joint/command
/arm_wrist_flex_joint/command
/gripper_controller/gripper_action/cancel
/gripper_controller/gripper_action/feedback
/gripper_controller/gripper_action/goal
/gripper_controller/gripper_action/result
/gripper_controller/gripper_action/status
/gripper_finger1_joint/command
/rosout
/rosout_agg
/scan
```

e. To the data being published on a ROS *topic /scan,* using following command rostopic echo /scan, as follow.

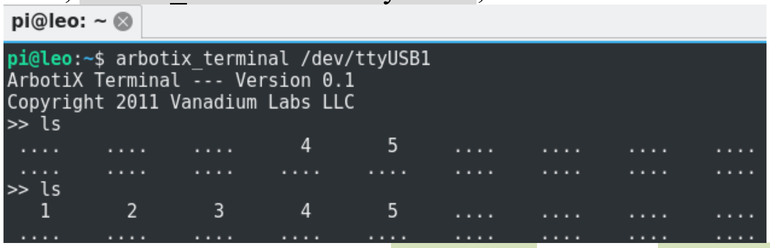


```
nasir@thesis:~$ rostopic echo /scan
header:
  seq: 649
  stamp:
    secs: 1715738314
    nsecs: 186725101
  frame_id: "laser"
angle_min: -3.1415927410125732
angle_max: 3.1415927410125732
angle_increment: 0.0032287694048136473
time_increment: 4.7161171096377075e-05
scan_time: 0.09177563339471817
range_min: 0.15000000596046448
range_max: 16.0
ranges: [inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, i
nf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, 2.3239
998817443848, 2.3239998817443848, 2.3239998817443848, 2.3239998817443848, 2.3239
998817443848, 2.3239998817443848, 2.328000068664551, 2.328000068664551, 2.328000
068664551, 2.315999984741211, 2.312000036239624, 2.312000036239624, 2.3080000877
38037, 2.303999900817871, 2.299999952316284, 2.299999952316284, 2.29999995231628
4, 2.296000038146973, 2.2920000553131104, 2.2920000553131104, 2.288000106811523
4, 2.2839999198913574, 2.2799999713897705, 2.2799999713897705, 2.276000022888183
6, 2.2720000743865967, 2.2720000743865967, 2.2720000743865967, 2.267999887466430
7, 2.2639999389648438, 2.2639999389648438, 2.259999990463257, 2.259999990463257,
```

Note: Scan topic shows the Publised data by Laser scanner attached to Lidar. You can also visulaize the Lase data in Rviz tool, shown in .

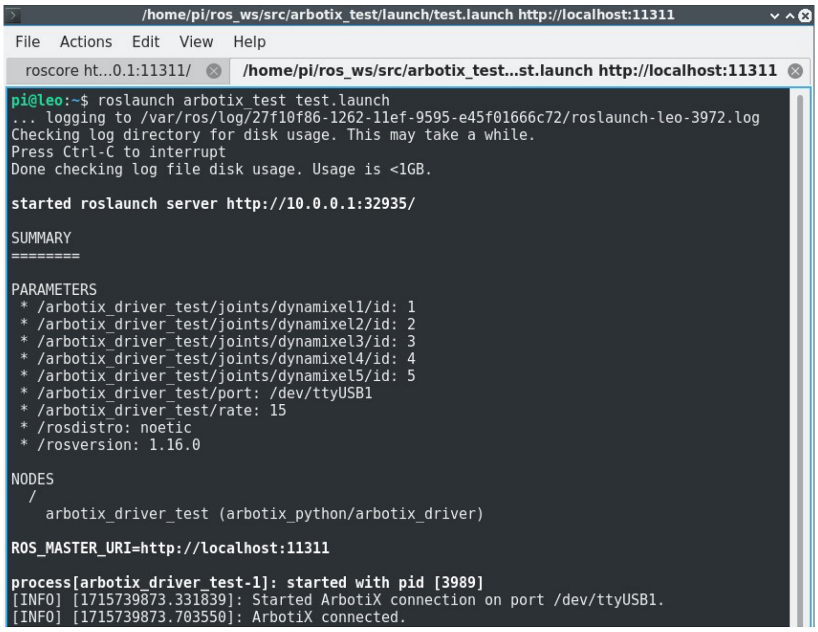
6. Testing ROS *nodes* and *topics* for Pincher Arm

a. To verify the rover arm's Dynamixel servo motors, use the follwoing commands, arbotix_terminal /dev/ttyUSB1, as follows:



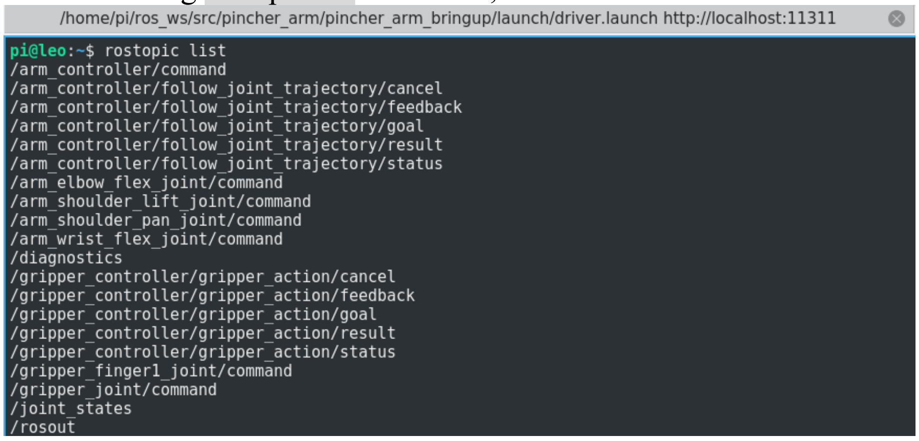
```
pi@leo: ~
pi@leo:~$ arbotix_terminal /dev/ttyUSB1
ArbotiX Terminal --- Version 0.1
Copyright 2011 Vanadium Labs LLC
>> ls
  ....    ....    ....     4       5     ....    ....    ....    ....
  ....    ....    ....    ....    ....   ....    ....    ....    ....
>> ls
   1       2       3       4       5     ....    ....    ....    ....
  ....    ....    ....    ....    ....   ....    ....    ....    ....
```

b. To test the servo motors, launch the test.launch file from arbotix_test package, as follows:



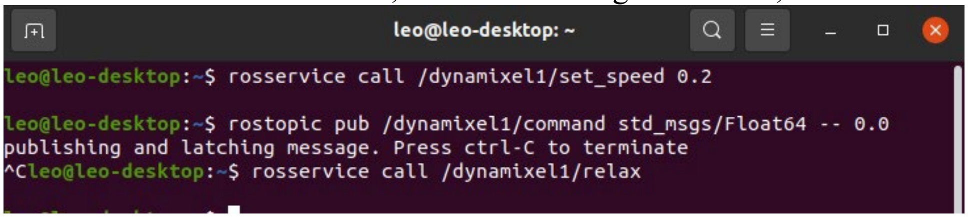
```
/home/pi/ros_ws/src/arbotix_test/launch/test.launch http://localhost:11311
File  Actions  Edit  View  Help
roscore ht...0.1:11311/    /home/pi/ros_ws/src/arbotix_test...st.launch http://localhost:11311
pi@leo:~$ roslaunch arbotix_test test.launch
... logging to /var/ros/log/27f10f86-1262-11ef-9595-e45f01666c72/roslaunch-leo-3972.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://10.0.0.1:32935/

SUMMARY
========

PARAMETERS
 * /arbotix_driver_test/joints/dynamixel1/id: 1
 * /arbotix_driver_test/joints/dynamixel2/id: 2
 * /arbotix_driver_test/joints/dynamixel3/id: 3
 * /arbotix_driver_test/joints/dynamixel4/id: 4
 * /arbotix_driver_test/joints/dynamixel5/id: 5
 * /arbotix_driver_test/port: /dev/ttyUSB1
 * /arbotix_driver_test/rate: 15
 * /rosdistro: noetic
 * /rosversion: 1.16.0

NODES
  /
    arbotix_driver_test (arbotix_python/arbotix_driver)

ROS_MASTER_URI=http://localhost:11311

process[arbotix_driver_test-1]: started with pid [3989]
[INFO] [1715739873.331839]: Started ArbotiX connection on port /dev/ttyUSB1.
[INFO] [1715739873.703550]: ArbotiX connected.
```

c. After the launching the file all ROS *nodes* and *topic* related to arm will available using rostopic list command, as follows:



```
/home/pi/ros_ws/src/pincher_arm/pincher_arm_bringup/launch/driver.launch http://localhost:11311
pi@leo:~$ rostopic list
/arm_controller/command
/arm_controller/follow_joint_trajectory/cancel
/arm_controller/follow_joint_trajectory/feedback
/arm_controller/follow_joint_trajectory/goal
/arm_controller/follow_joint_trajectory/result
/arm_controller/follow_joint_trajectory/status
/arm_elbow_flex_joint/command
/arm_shoulder_lift_joint/command
/arm_shoulder_pan_joint/command
/arm_wrist_flex_joint/command
/diagnostics
/gripper_controller/gripper_action/cancel
/gripper_controller/gripper_action/feedback
/gripper_controller/gripper_action/goal
/gripper_controller/gripper_action/result
/gripper_controller/gripper_action/status
/gripper_finger1_joint/command
/gripper_joint/command
/joint_states
/rosout
```

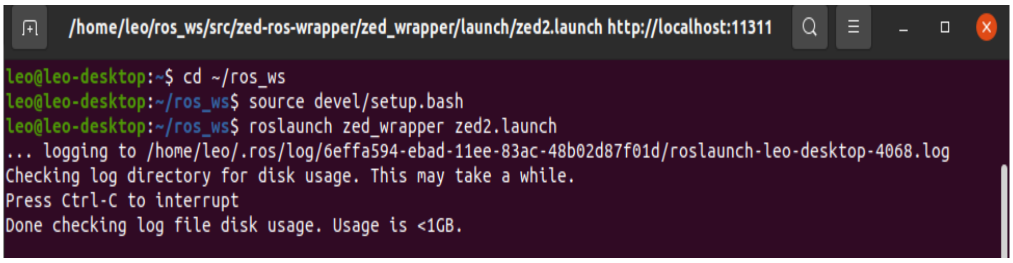
d. To test the motors on terminal, use the following commands, as follows:



```
leo@leo-desktop:~$ rosservice call /dynamixel1/set_speed 0.2

leo@leo-desktop:~$ rostopic pub /dynamixel1/command std_msgs/Float64 -- 0.0
publishing and latching message. Press ctrl-C to terminate
^Cleo@leo-desktop:~$ rosservice call /dynamixel1/relax
```

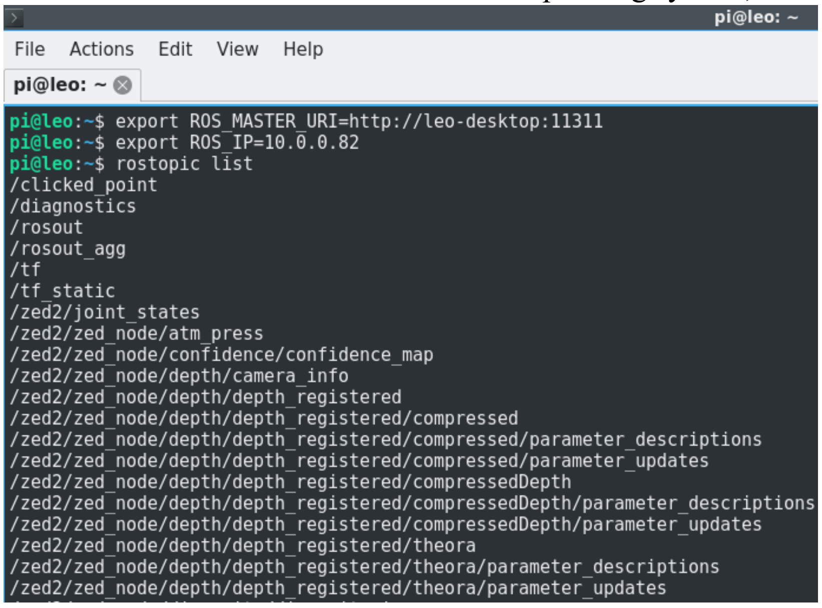
Note: The same commands will be used to test the other servo motors. You just need to change the motor name /dynamixel?.

7. **Testing ROS *nodes* and *topics* for ZED2 camera**
   a. Make sure the Ethernet cable is connected to additional computer and has same network as RPi.
   b. Launch the zed2.luanch file from its installed ROS package zed_wrapper, available at workspace directory */ros_ws*, as follows*:*


/home/leo/ros_ws/src/zed-ros-wrapper/zed_wrapper/launch/zed2.launch http://localhost:11311

```
leo@leo-desktop:~$ cd ~/ros_ws
leo@leo-desktop:~/ros_ws$ source devel/setup.bash
leo@leo-desktop:~/ros_ws$ roslaunch zed_wrapper zed2.launch
... logging to /home/leo/.ros/log/6effa594-ebad-11ee-83ac-48b02d87f01d/roslaunch-leo-desktop-4068.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
```

   c. Export the IP address of the additional computer (10.0.0.82) to RPi board to test the camera functionalities in rover's operating system, as follows:

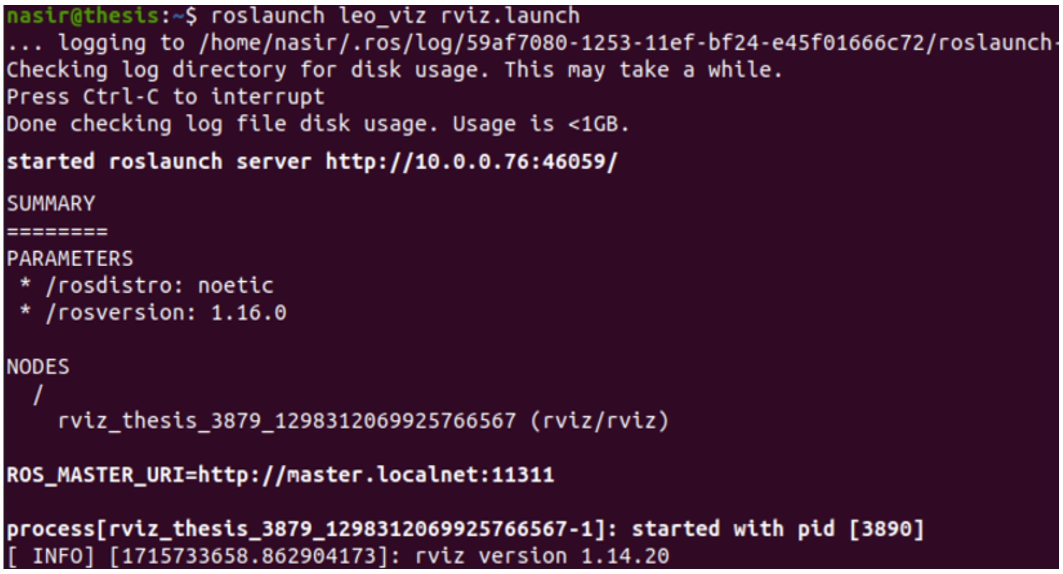


```
pi@leo:~$ export ROS_MASTER_URI=http://leo-desktop:11311
pi@leo:~$ export ROS_IP=10.0.0.82
pi@leo:~$ rostopic list
/clicked_point
/diagnostics
/rosout
/rosout_agg
/tf
/tf_static
/zed2/joint_states
/zed2/zed_node/atm_press
/zed2/zed_node/confidence/confidence_map
/zed2/zed_node/depth/camera_info
/zed2/zed_node/depth/depth_registered
/zed2/zed_node/depth/depth_registered/compressed
/zed2/zed_node/depth/depth_registered/compressed/parameter_descriptions
/zed2/zed_node/depth/depth_registered/compressed/parameter_updates
/zed2/zed_node/depth/depth_registered/compressedDepth
/zed2/zed_node/depth/depth_registered/compressedDepth/parameter_descriptions
/zed2/zed_node/depth/depth_registered/compressedDepth/parameter_updates
/zed2/zed_node/depth/depth_registered/theora
/zed2/zed_node/depth/depth_registered/theora/parameter_descriptions
/zed2/zed_node/depth/depth_registered/theora/parameter_updates
```

Note: There are multiple ROS *nodes* and *topic* for ZED2 camera, however few of them are listed in the figure.

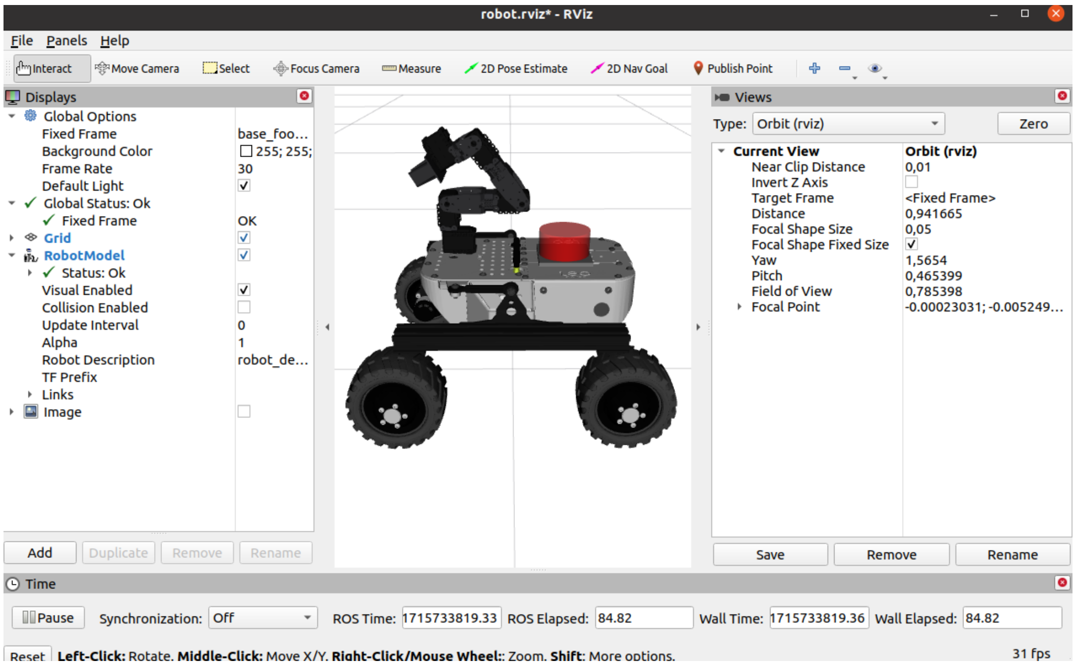
8. **Testing Lidar sensor in Rviz tool**

a. To test the Lidar sensor, run the follwoing command, as follows:



```
nasir@thesis:~$ roslaunch leo_viz rviz.launch
... logging to /home/nasir/.ros/log/59af7080-1253-11ef-bf24-e45f01666c72/roslaunch
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://10.0.0.76:46059/

SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.16.0

NODES
  /
    rviz_thesis_3879_1298312069925766567 (rviz/rviz)

ROS_MASTER_URI=http://master.localnet:11311

process[rviz_thesis_3879_1298312069925766567-1]: started with pid [3890]
[ INFO] [1715733658.862904173]: rviz version 1.14.20
```

b. Rover will look like this as shown in given below figure:



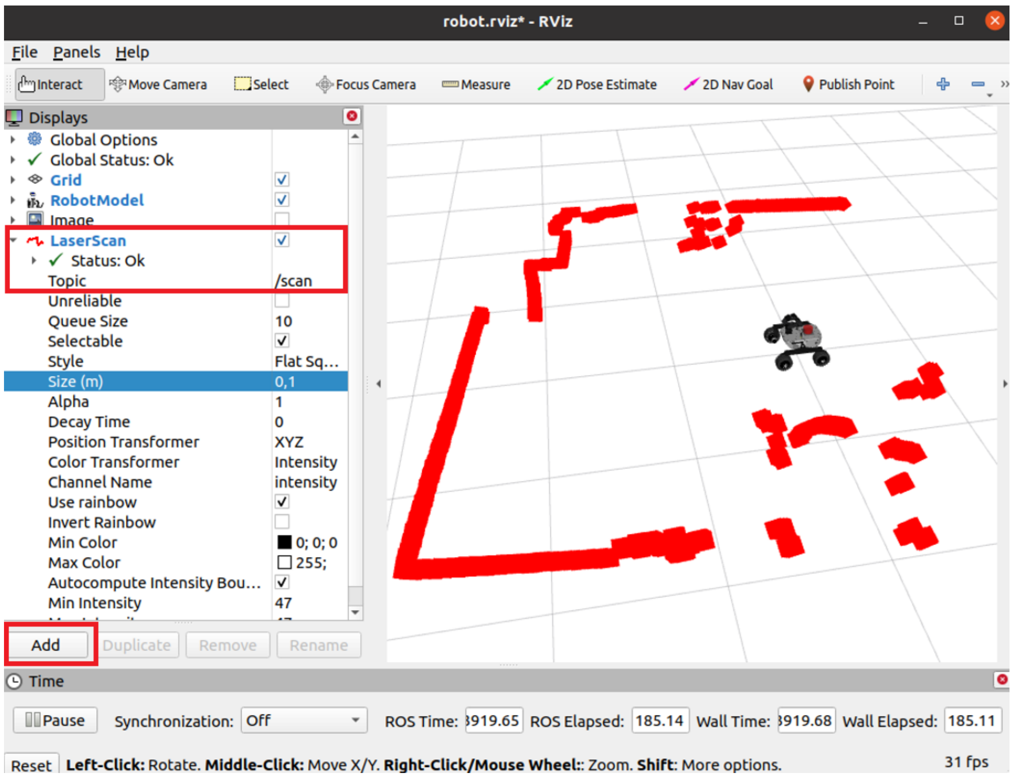
```
Global Options
  Fixed Frame            base_foo...
  Background Color       255; 255;
  Frame Rate             30
  Default Light          ✓
Global Status: Ok
  Fixed Frame            OK
Grid                     ✓
RobotModel               ✓
  Status: Ok
  Visual Enabled         ✓
  Collision Enabled
  Update Interval        0
  Alpha                  1
  Robot Description      robot_de...
  TF Prefix
  Links
Image
```

```
Type: Orbit (rviz)
Current View             Orbit (rviz)
  Near Clip Distance     0,01
  Invert Z Axis
  Target Frame           <Fixed Frame>
  Distance               0,941665
  Focal Shape Size       0,05
  Focal Shape Fixed Size ✓
  Yaw                    1,5654
  Pitch                  0,465399
  Field of View          0,785398
  Focal Point            -0.00023031; -0.005249...
```

```
Synchronization: Off   ROS Time: 1715733819.33   ROS Elapsed: 84.82   Wall Time: 1715733819.36   Wall Elapsed: 84.82
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.   31 fps
```

Note: If the Lidar sensor and robotic arm are succefully connected to rover, it will also visulize in Rviz tool. Make sure the postion of the attached devices should be same as in physical model.

c. To check the publised Lidar data, enable */scan topic* by pressing Add tab , shown in red-colour box, as follows:



Global Options	
Global Status: Ok	
Grid	✓
RobotModel	✓
Image	
LaserScan	✓
Status: Ok	
Topic	/scan
Unreliable	
Queue Size	10
Selectable	✓
Style	Flat Sq...
Size (m)	0,1
Alpha	1
Decay Time	0
Position Transformer	XYZ
Color Transformer	Intensity
Channel Name	intensity
Use rainbow	✓
Invert Rainbow	
Min Color	0; 0; 0
Max Color	255;
Autocompute Intensity Bou...	✓
Min Intensity	47

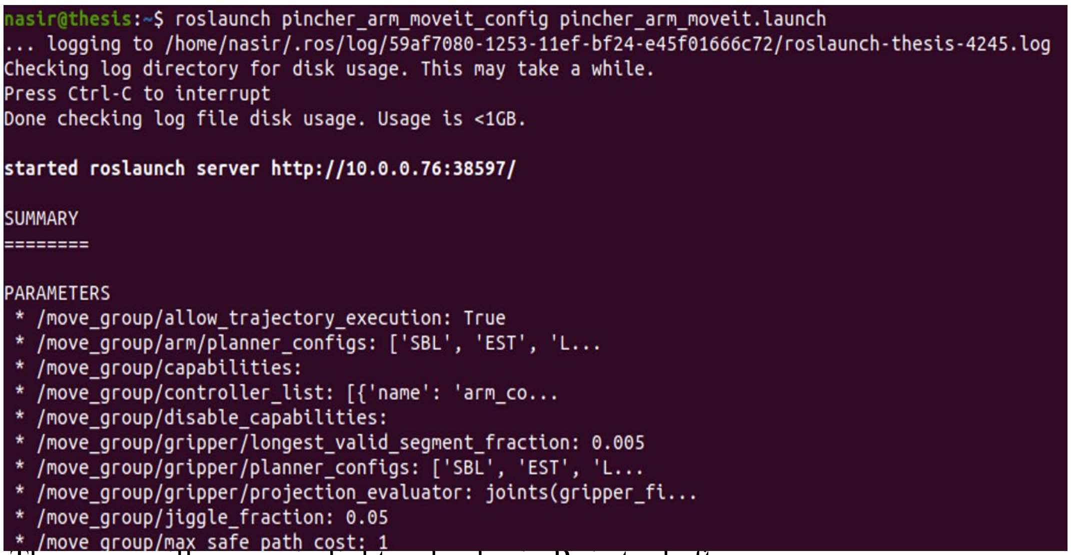
Synchronization: Off ROS Time: 3919.65 ROS Elapsed: 185.14 Wall Time: 3919.68 Wall Elapsed: 185.11

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options. 31 fps

Note: The Laser data is shown in red-dotted lines along with the rover in Rviz tool. This process ensure that the Lidar sensor is sucessfully connected and working properly.

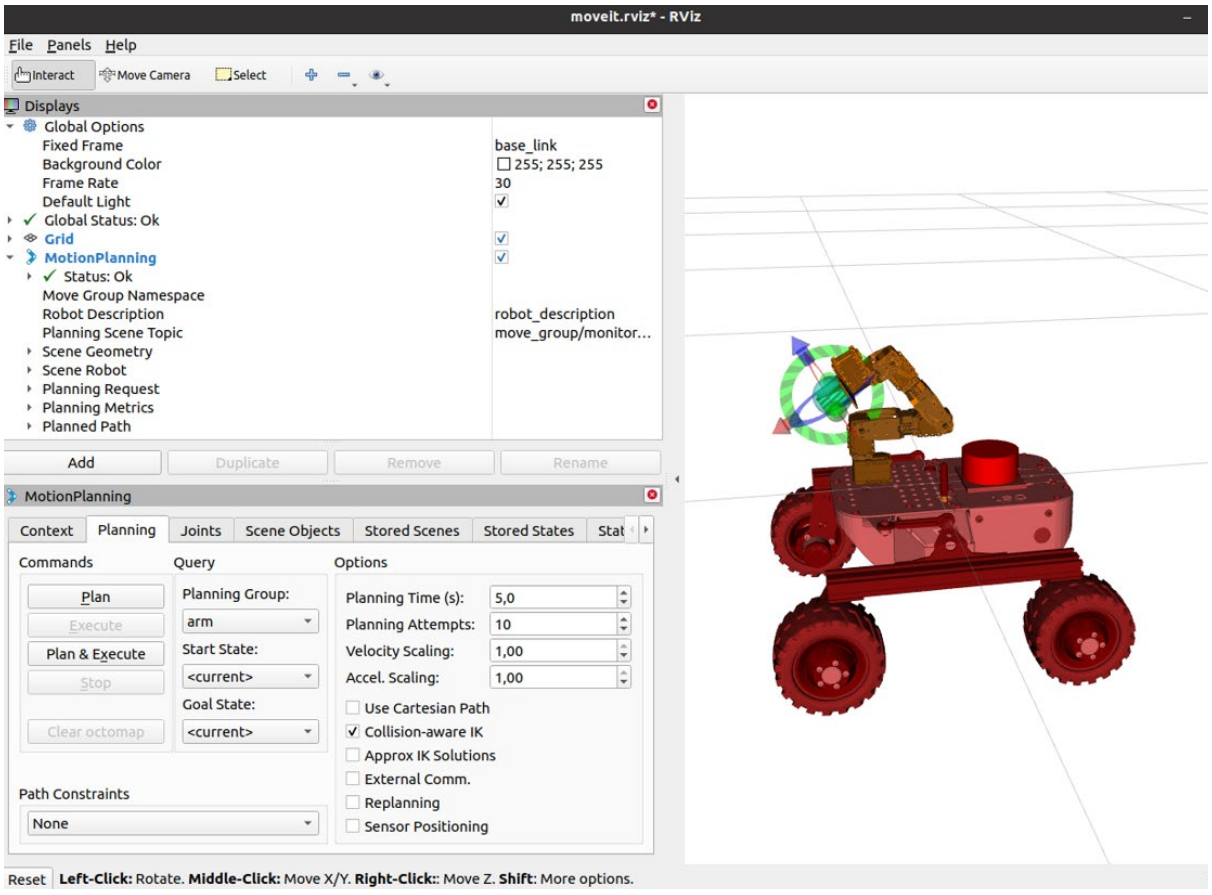
9. Testing Robotic arm in Rviz tool

a. Launch the pincher_arm_moveit.launch file from its ROS package pincher_arm_moveit_config, as follows:



```
nasir@thesis:~$ roslaunch pincher_arm_moveit_config pincher_arm_moveit.launch
... logging to /home/nasir/.ros/log/59af7080-1253-11ef-bf24-e45f01666c72/roslaunch-thesis-4245.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://10.0.0.76:38597/

SUMMARY
========

PARAMETERS
 * /move_group/allow_trajectory_execution: True
 * /move_group/arm/planner_configs: ['SBL', 'EST', 'L...
 * /move_group/capabilities:
 * /move_group/controller_list: [{'name': 'arm_co...
 * /move_group/disable_capabilities:
 * /move_group/gripper/longest_valid_segment_fraction: 0.005
 * /move_group/gripper/planner_configs: ['SBL', 'EST', 'L...
 * /move_group/gripper/projection_evaluator: joints(gripper_fi...
 * /move_group/jiggle_fraction: 0.05
 * /move_group/max_safe_path_cost: 1
```

b. The rover will appear in light-red color in Rviz tool after.


moveit.rviz* - RViz
File Panels Help
Interact Move Camera Select
Displays
Add Duplicate Remove Rename
MotionPlanning
Context Planning Joints Scene Objects Stored Scenes Stored States Stat
Commands
Plan
Execute
Plan & Execute
Stop
Clear octomap
Reset

Global Options	
Fixed Frame	base_link
Background Color	☐ 255; 255; 255
Frame Rate	30
Default Light	✓
Global Status: Ok	
Grid	✓
MotionPlanning	✓
Status: Ok	
Move Group Namespace	
Robot Description	robot_description
Planning Scene Topic	move_group/monitor...
Scene Geometry	
Scene Robot	
Planning Request	
Planning Metrics	
Planned Path	

Query
Planning Group: arm
Start State: <current>
Goal State: <current>

Path Constraints
None

Options
Planning Time (s): 5,0
Planning Attempts: 10
Velocity Scaling: 1,00
Accel. Scaling: 1,00
☐ Use Cartesian Path
☑ Collision-aware IK
☐ Approx IK Solutions
☐ External Comm.
☐ Replanning
☐ Sensor Positioning

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Move Z. **Shift:** More options.

Note: In the left side of the figure, there are several tabs to test the arm fucntionalities. This process ensure that the robotic arm is succesfully connected and ready for testing.