Computational Science and Engineering
Department of Informatics

Technische Universität München

Master's Thesis

BIM based Autonomous Navigation of a Quadruped Robot

Anuj Berwal

Computational Science and Engineering
Department of Informatics

Technische Universität München

Master's Thesis

BIM based Autonomous Navigation of a Quadruped Robot

Author:	Anuj Berwal
Supervisors:	Prof. Dr.-Ing. André Borrmann Miguel A. Vega Torres M.Sc Chair of Computational Modeling and Design TUM School of Engineering and Design
Submission Date:	14. July 2023

Abstract

In recent years, the usage of robotic systems has increased significantly in all fields, and there is an increasing focus on developing intelligent systems capable of autonomously navigating in various scenarios. The ROS framework offers multiple solutions to the key SLAM problem in autonomous navigation research. There are still a few challenges to autonomous navigation in indoor environments, such as the lack of reliable Global Positioning System (GPS) data or the need to install WiFi tags or QR code markers. A potential solution is to use 3D BIM models of facilities to aid the localization part of autonomous navigation algorithms by using the semantic information of the static structures like walls and floors of the building. Most modern buildings have a digital twin available, which accurately describes the static physical structure of the building. By feeding these digital models to robotic systems, remotely operated mapping and navigation can be achieved efficiently. In this research work, a basic BIM-based autonomous navigation framework will be designed to simulate the navigation of a quadruped robot in the Gazebo simulator. Then this framework will be implemented on a Unitree Go1 quadruped robot to verify the real world performance.

Contents

List of Figures

1. Introduction

1.1. Motivation

In recent years, there has been a rapid increase in the usage of robotic systems in different industries and research institutes. There have been significant improvements in robotic hardware design: from wheeled robots to legged robots and now flying quadcopter drones. In the earlier phases, the development was majorly focused on wheeled robots that were easier to control and move in well-defined areas. However, such wheeled robots perform poorly in navigating real world environments, as such scenarios often involve uneven terrains and multiple floors connected by staircases or electrical lifts. To overcome these issues of wheeled robots, recent research has been focused on developing sophisticated legged robots whose walking or running patterns can be dynamically designed and controlled. These developments have enabled the usage of robots in industrial environments where using them might have seemed unfathomable a few years ago. The advantages include optimum costs and flexible design controls.

Following the popular usage of robotic systems in other industries, the construction industry has also begun focusing on more intelligent systems that involve less human control, provide more accurate measurements, and enable a faster problem analysis leading to optimized and rapid development. The modern construction industry faces several issues, including the rise in energy costs, lack of skilled workforce, and challenging work conditions, including environments where it is near impossible for a human to work, e.g., high-rise buildings, extreme weather environments, and oceans. These issues are not limited to just new construction projects. Instead, issues such as natural disasters lead to chaos and the breakdown of existing building structures which become extremely difficult to access for humans. Robotic machines offer excellent opportunities in this industry to support humans in such hazardous environments while minimizing the costs of search and rescue operations.

One central area of focus in the robotics research community is Simultaneous Localization and Mapping (SLAM), i.e., mapping an environment using mobile robots while simultaneously localizing it in that environment. The robot is moved around in the unknown environment to create the map either through remote control or onsite control. As the robot moves around in the environment, it draws a map with reference to its position within the map. The mapping quality of an environment depends on the precise estimation of the robot's position. This will be further explained in the upcoming chapters.

Mapping can be further split into two categories: Outdoor and Indoor mapping. While outdoor mapping can be done without many issues, as it is often aided by GPS signals

or visible landmarks, the challenges for indoor mapping are higher because of the lack of reliable information about the robot's current position. GPS connectivity is often either of poor quality or completely missing in indoor environments compared to outdoor environments. However, several solutions have been proposed in recent years, for instance, using physical objects like Radio frequency identification (RFID) tags[13], WiFi routers for communication, fiduciary markers, or other electronic tags. However, these solutions require an additional cost and are not feasible for large-scale usage.

In this research work, a Building Information Modelling (BIM) based solution is proposed for localizing the robot in an indoor environment. A BIM model contains the 3D geometrical model of the environment and also provides basic semantic information about the environment, like the position of doorways, staircases, or walls. By feeding this information to the SLAM module, the localization of a robotic agent can be improved, thus improving the mapping quality.

The improved localization performance can be used to control the robot remotely if the localization performance is highly accurate. In general, the focus will be on dynamic environments like under-construction building projects or high-traffic indoor areas. Compared to wheeled robots, a quadruped robot performs much better in such environments. Autonomous navigation is preferred as it provides more flexibility in design and control compared to an onsite control module, saving precious time and costs. This thesis will explore the latest research works in robotics to develop a BIM-aided autonomous navigation framework. This framework will first be developed in a simulation environment and then integrated into a quadruped robot to verify real world performance.

1.2. Research objectives

The main focus of this thesis is to achieve a basic framework for the autonomous navigation of a quadruped robot in the real world by utilizing information from BIM models. The BIM models will be used to create a 2D map of the indoor environment for localization and navigation. The core objectives of this research work are listed below.

- The first objective is to create a virtual simulation framework in Robot Operating System (ROS) using the Gazebo simulator and Rviz visualization tools. This would require creating an accurate ROS compliant description of the quadruped robot and its operating environment.
- The next objective is to perform a robot and sensor simulation in a virtual environment. The simulations should focus on measuring the performance of the robot's mapping, localization, and navigation capabilities in this virtual environment. A comparison should be made between the various state-of-the-art localization algorithms and packages already available in ROS. The optimal package that promises good performance in the real world should be selected for navigating the quadruped robot in a real world environment.
- Another objective is to extract relevant information from a BIM model of a real world environment. This information should be provided to the ROS based framework to verify if it leads to an improvement in the SLAM performance.
- When the simulation provides satisfactory results without errors or crashes, the autonomous navigation stack shall be implemented and tested in a real quadruped robot. The path planning algorithm's performance in planning a path between the current position and a target goal shall be measured. The robot is expected to navigate autonomously through the calculated waypoints without issues. The robot should be able to detect and avoid obstacles in its path.
- At the end, conclusions shall be drawn from the simulations and real world experiments done in this research work. Furthermore, some potential extensions for this research work shall be discussed.

1.3. Reading guide

The thesis document is divided into the following chapters:

- In Chapter 2 , the theoretical background necessary to understand the entire research work is detailed.
- In Chapter 3 , the latest research work related to the thesis is discussed and summarized.
- In Chapter 4 , the process of setting up the experimental framework and workflow for this thesis are explained.
- In Chapter 5 , the details of the virtual and real world experiments and the results are described and discussed.
- In Chapter 6 , this research work is summarized, and concluding statements are provided. The future scope of extension of this work is also discussed.

2. Theoretical Background

This chapter describes the relevant theoretical background that is essential to understand this document.

2.1. SLAM

Simultaneous localization and mapping (SLAM) is a classical computational problem in the autonomous navigation domain that involves creating a map of an unknown environment while simultaneously locating the agent's position that intends to perform autonomous navigation. This is often compared to the classic chicken-and-egg problem since it is difficult to localize the agent in a map without the knowledge of the environment, and it is similarly challenging to map the environment without knowing where the agent is. Extensive research in the past decades has attempted to solve this problem, and several potential solutions exist with the aid of different kinds of sensors.

Two significant approaches famous in the robotic community are Lidar based SLAM and Camera based SLAM solutions. Lidars provides depth and edge features of objects present in the environment. Lidar based SLAM solutions focus on detecting these objects and then represent them as point clouds. The camera sensor provides high-quality images of the environment and the SLAM solutions often focus on contour detection or separation of colors to draw objects of interest in the environment. SLAM has been one of the most popular research topics in the robotics community and multiple algorithms are available for either mapping a new environment or localizing an agent in an existing map of the environment.

2.2. Building Information Models

Building Information Modeling is the process of creating a three dimensional virtual model of a building or a construction facility, and the model resulting from the process is called a Building Information Model (BIM Model). These models facilitate shared project development between multiple stakeholders by making it easier to modify the stored information over time. They contain important information about a structure, such as the coordinates of individual parts, the texture and color of surfaces, and the location of different parts of the building. They also specify semantic information about objects like floors, walls, doors, or ceilings inside the building.

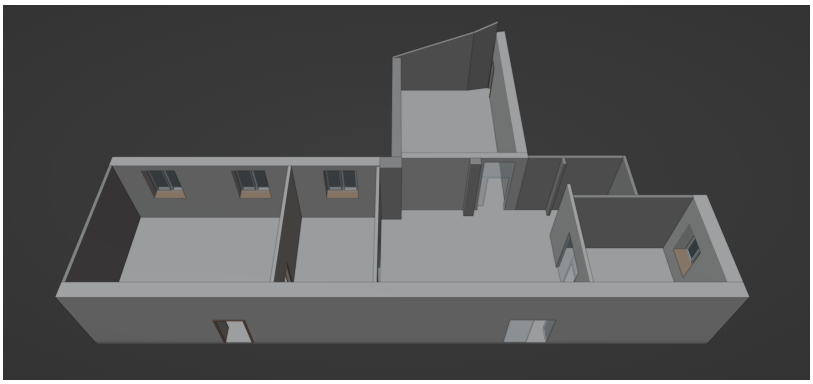


Figure 2.1.: A sample BIM Model

BIM models can also be extended to four dimensions or higher by adding further information, such as time, that tracks how the model changed gradually over time. The digital form is especially helpful in designing simulations and testing environments for those scenarios that are challenging to create in real life. In this research work, we utilized 3D BIM models for creating 2D and 3D maps that will be, in turn, used for autonomously navigating a robot. Figure 2.1 shows a sample BIM model in Industry Foundation Classes (IFCs) format, which is recognized as a standard ISO format (ISO 16739-1:2018, [31]).

2.3. Quadruped Robots

In the past few years, many robotic systems have been developed to support navigation on different kinds of indoor and outdoor terrains. The two major robotic systems are wheeled robots and legged robots. The wheeled robots generally consist of four-wheel setups that are based on differential drive systems similar to cars. Initial robotic development in ROS to achieve the goals of remote navigation and autonomous navigation focused on such wheeled robots. They tend to suit flat terrains and support elevated flat terrains like slopes but struggle to move on uneven terrains and climb staircases in multi-floor areas. To overcome these issues, legged robots were developed that were more sophisticated and could travel across uneven terrains and climb stairs. Though two-legged robots currently exist and are being actively developed, they are often larger in size and less flexible compared to four-legged robots. It is also quite tricky to add payloads to a two-legged robot, and there exist limitations to their stability during navigation. In comparison, four-legged robots are more robust and efficient in traversing uneven terrains while being equally good at moving indoors or doing trivial tasks like climbing stairs. The four-legged robots have become quite popular recently due to their apparent advantages over two-legged robots and accessibility to multiple open source ROS-based algorithms that facilitate navigation. For our research objectives, four-legged robots are best suited as they can carry a payload,

including a perception sensor system that will be mounted on the robot.

2.4. Basics of Robot Operating System

Robot Operating System (ROS) is an open source platform that provides a vast collection of software tools and libraries that can be used to simulate or program robotic applications. ROS intends to provide a standard low-level framework for any robotics development. It primarily acts as a communication service between multiple hosts and individual applications running on them and is synchronized in real-time using ROS. ROS is language agnostic and can work with both C++ and Python efficiently.

There are two major *generations* of ROS, i.e., ROS1 and ROS2, each having multiple software releases. ROS2 is relatively newer and is still under active development. This work utilizes ROS1 and related tools as a starting step, but this project can be migrated to ROS2 as well with some modifications.

2.4.1. Communication

Communication in ROS is facilitated by Nodes and Topics. XML-based launch files are used to load the nodes and topics in ROS.

Nodes

```
$ # List all nodes
$ rosnode list

$ # Information about a specific node
$ rosnode info <node_name>
```

Source Code 2.1.: Display information about nodes

Nodes are individual processes that perform a computational task with the ability to communicate with each other. E.g., One node could be dedicated to processing an input Lidar scan, and another could control the robot's movement by processing the wheel sensor information. Nodes communicate with each other via ROS topics. Code 2.1 shows the shell commands to generate information about nodes.

Ros Graph

ROS provides a simple-to-use GUI-based tool called ROS Graph that displays the different nodes and the flow of information between them. The graph can be launched using

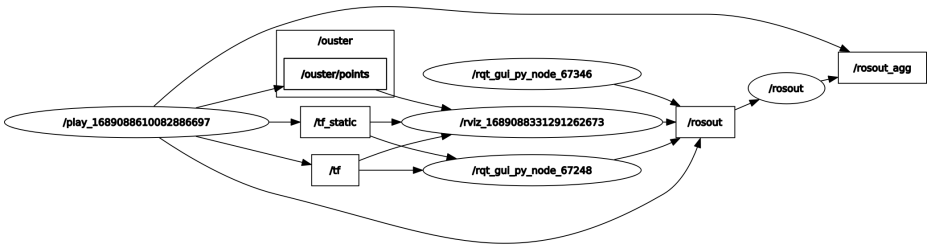


Figure 2.2.: Example of a Rqt graph

```
$ rosrun rqt_graph rqt_graph
```

Source Code 2.2.: Start ROS Graph

the command listed in Code 2.2. This information comes in handy when there are a large number of nodes interacting with each other via multiple topics.

Topics

```
$ # List all topics
$ rostopic list

$ # Information about a specific topic
$ rostopic info <topic_name>

$ # Messages published by a specific topic
$ rostopic echo <topic_name>
```

Source Code 2.3.: Display information about topics

Topics are used to send or receive data between nodes. Nodes are classified into two types: publisher and subscriber. As the names indicate, a publisher node publishes certain information, and a subscriber node subscribes or receives that information. Topics facilitate this flow of information between publisher and subscriber nodes. Topics can be of different types, and it is essential to specify the message type for each topic. E.g., a node publishing 3D lidar data will use "sensor_msgs/PointCloud2" [57] message type, and similarly, a node that wants to receive and process 3D lidar data should use a subscriber having message type "sensor_msgs/PointCloud2". Code 2.3 shows the shell commands to

generate information about nodes.

Transformation in ROS

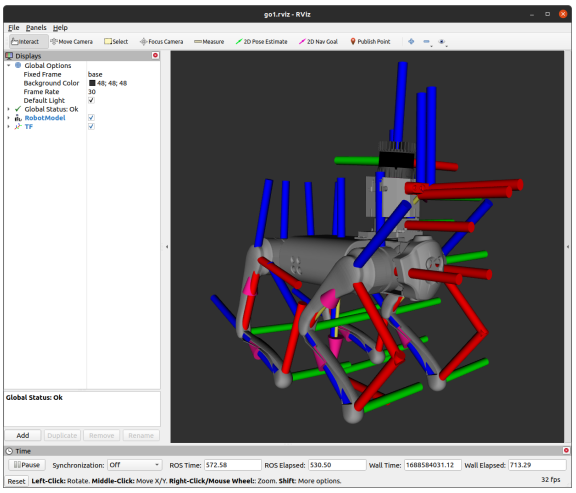


Figure 2.3.: Transform visualization in Rviz

In a robotic system, multiple sensors support a robot and produce data in their own coordinate system. ROS requires that a basic structure should be established among these parts using transformation so that every device operates in the same coordinate system. For instance, the quadruped robot is made up of several mechanical parts, which are defined as joints and links in ROS and all their data should arrive in the same coordinate system. While links are fixed, joints can perform multiple motions based on their design objective. An XML-based Unified Robotics Description Format (URDF) file makes this information available to ROS. In this URDF, the transforms between different frames are defined.

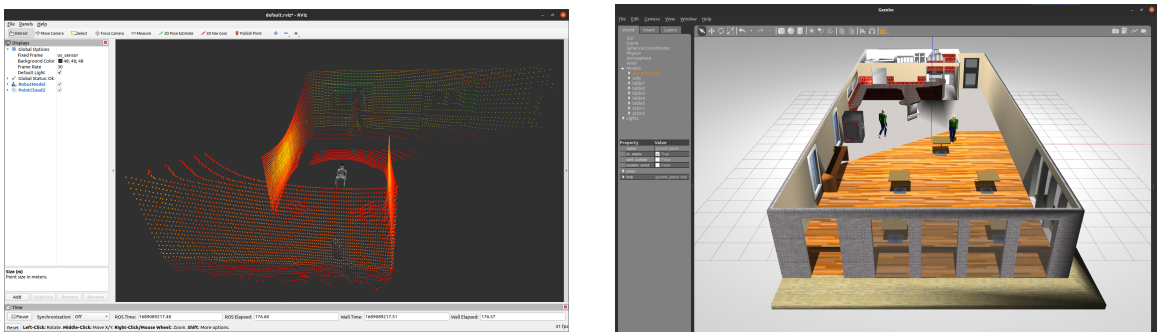
ROS utilizes a coordinate transformation library *tf* to either publish or receive coordinate transformations. This is done with the help of a *Broadcaster* that broadcasts information about different coordinate frames and a *Listener* that listens/receives such information. The exchange of information is done in a message format *tf/tfMessage*. This format also includes the coordinate transform information and the timestamp of the incoming messages, which helps nodes synchronize multiple inputs from different nodes. E.g., the robotic system should know when a 3D point cloud was received from the lidar sensor or the timestamp of the current velocity. A minor delay or desynchronization in this information can lead to dangerous consequences. The user must set up this transformation as a connected tree by defining a ”base” frame to which all other frames will deliver information. The frames in the tree have either a parent or child frame to detail the flow of information. An unconnected transformation tree would lead to multiple problems, and

ROS would have difficulties processing the messages correctly. The transform tree can be seen in real time by running the command listed in Code 2.4. Figure 2.3 shows a visualization of the cooridnate systems of a robot's joints and links using *tf* library in Rviz tool.

```
$ rosrun rqt_tf_tree rqt_tf_tree
```

Source Code 2.4.: Display tranformation tree

2.4.2. Visualization



(a) Visualization of Lidar data in Rviz

(b) Sample Cafe world in Gazebo

Figure 2.4.: ROS visualization tools

Rviz

ROS visualization (Rviz) is a powerful visualization tool in ROS that displays both 2D and 3D information about robotic systems. It can display a virtual model of the robot and incoming data from sensors like lidar and camera in 2D and 3D, which capture the robot's operating environment. It is possible to view other information like obstacles or the planned path of the robot. It is also possible to design custom plugins for Rviz and visualize the data. Rviz was extensively used for this research work as a primary visualization tool to quickly and efficiently configure the used robotic system. Figure 2.4a shows a sample visualization of 3D lidar pointcloud data in Rviz tool.

Gazebo

Gazebo is a 3D simulator tool that simulates the physical dynamics of robots and sensors. It facilitates the creation of custom 3D environments that accurately represent the physical

world in which the robot will operate. It also provides the capability to simulate lidar and camera sensors of various kinds. The robot's movement can be designed to work as it works in the real world, and configure the sensors to test various scenarios that can be created in the real world. It can be integrated into ROS using a set of packages named *gazebo_ros_pkgs*. This tool was utilized extensively in this research work to simulate the robot's 3D sensor data and perform autonomous navigation. Figure 2.4b shows a sample 3D environment in *Gazebo* simulator.

2.4.3. Navigation Stack

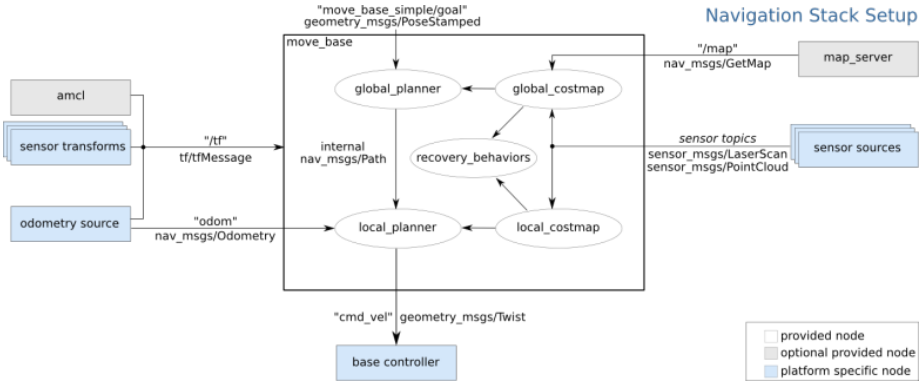


Figure 2.5.: ROS Navigation Stack

ROS provides a navigation stack that can be used to program any robot for navigation tasks in a 2D *x-y plane*. The navigation stack can receive information from multiple sensors like odometry, lidar, or camera and use this information to send translational and rotational velocity commands to the robot. It is necessary to properly set up the transformation *tf tree* for the robot so that all the information can be accessed from the robot's base frame. A proper setup ensures that the information from multiple sensors is synchronized in time, thus avoiding costly mistakes that might be caused due to sending the wrong velocity command or causing collisions due to incorrect detection of an object as free space. Through the robot's URDF file, the navigation stack also receives information about the robot's size and shape, which is critical in determining if the robot can traverse a planned path. Configuration files necessary for the navigation stack are written in YAML format. Figure 2.5 shows an overview of the ROS navigation stack.

Sensor sources

The navigation stack can utilize data from different sensors to process the environment's information and identify objects, surfaces, and free space. It is required that these messages should be either of type sensor *sensor_msgs/LaserScan* or type sensor *sensor_msgs/PointCloud*.

Most popular sensors in the market usually provide a ROS interface or drivers detailing how to generate this information.

Odometry

Another critical information for setting up the Navigation stack is Odometry data which is generated by motion sensors. These sensors measure the robot's position in the environment by estimating the translational and rotational change in the position. Inertial measurement unit (IMU) sensors are popularly used to determine odometry information, but they can also be estimated from perception sensors like Cameras or Lidar. The navigation stack does not emphasize a particular input sensor. However, it expects the data to be in *nav_msgs/Odometry* message format, which ensures compatibility with multiple sensors and flexibility in case one of them goes bad or starts sending junk information. This is facilitated by fusing the information from different sensors using Kalman Filters.

Map

In addition to the sensor data, the navigation stack requires a 2D occupancy grid map of the operating environment. Occupancy grid maps are designed as two-dimensional grids to simplify the localization and tracking of an agent in the map by using its 2D coordinates. Each cell in the grid map contains numerical information to specify if an object occupies it or it contains free space. This map can either be created before running the navigation stack or during the navigation operation. Initial map creation can be done by multiple methods depending on available resources. For instance, in environments like a small room, it might be sufficient to roughly draw the map on painting software and then use lidar or camera scan matches to tune the map for accuracy. This painting method becomes complicated for larger environments and requires more accurate tools like 3D to 2D map projection or map slicing from 3D files. Though, for most projects, it might be much easier first to move the robot around an environment: either small or large, and create the map using the SLAM algorithms provided by ROS. As the name suggests, these algorithms can create a map in real-time by measuring the odometry data from IMU sensors and using the perception sensor data for edge detection and matching. This provides an easy way to build a map for any kind of environment incrementally.

Costmaps

The navigation stack uses costmaps to identify obstacles and free space in the environment. The costmaps are created based on the input 2D map and object detection information from perception sensors. The costmaps are designed as occupancy grid maps in which a cell is assigned a value or cost depending on whether it is an obstacle (high cost) or free space (low cost). This information is necessary when the robot is given a target point and is directed to navigate autonomously to that point. There are two types of costmaps: Global

costmap and Local costmap. The Global costmap stores information about the whole environment and is used for path planning for long-distance targets. The Local costmaps store information about the robot's immediate surroundings, detecting obstacles in the long-term/global path and creating short local plans. Both of these costmaps have several configuration parameters that can be set up easily for different robots and environments. Some vital information that these costmaps require are:

- Robot dimensions: It is used to represent the robot as a point of proportional size in the map
- Sensor data type: Whether the sensor data is in 2D (sensor_msgs/LaserScan) or 3D (sensor_msgs/PointCloud)
- Sensor detection range and obstacle range: To determine and update obstacles in the given range

The global costmap displays information in a global "map" frame and requires the robot's "base" frame to track it. Whereas local costmap displays information in a local odometry-based "odom" frame that moves dynamically with respect to the global "map" frame. The size of this local costmap can be configured depending on requirements, but the global costmap is always configured to the entire map published on "/map" topic.

Planner

Similar to costmaps, the navigation stack has two planners: Global planner and Local planner. Given a target on the map, a global planner computes a path in the entire map to reach that point. Global planner provides a set of waypoints that should be traversed to reach the target, considering the obstacles. The availability of a pre-built map of the environment is not necessary for a global planner, and it can provide an initially expected path and keep updating itself incrementally over time to improve the quality of the path. However, this might reduce the accuracy of long-distance path planning. In contrast, the local planner plans immediate short paths in the local costmap and is responsible for computing and publishing the velocity commands to the robot. It checks the obstacles in real-time and updates frequently to reflect changes in the map. Thus it is very critical to configure it correctly to perform in a highly dynamic environment. In comparison, the global planner accesses vast global information for planning the path and is not updated as frequently as the local planner. Global planner can support many planner algorithms like Dijkstra's[14] or A*[24] path algorithm. For local planner, ROS provides multiple algorithms like *base_local_planner* [59], *dwa_local_planner* [58] and *teb_local_planner* [60].

Base controller

To navigate the robot, the navigation stack sends velocity commands via a base controller on "cmd_vel" topic, which describes the required values of velocity in the x and y direction

(vx, vy) and rotational information (vtheta). *cmd_vel* topic computes these values based on the configuration parameters and by interacting with the costmaps and planners. It then publishes this information using messages of type *geometry_msgs/Twist.msg*. Therefore, a node should be created to accept these *geometry_msgs/Twist.msg* messages, and configure the robot's controllers to move it accordingly.

```
<!-- geometry_msgs/Twist.msg -->
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular

<!-- geometry_msgs/Vector3.msg -->
float64 x
float64 y
float64 z
```

2.5. Mapping and Localization in ROS

As discussed in previous sections, a map is required to create costmaps and planners for autonomous navigation in ROS. The map can be built during runtime while moving the robot in the environment or provided as a static map created before navigation. Depending on the chosen map strategy, ROS provides several algorithms that are capable of performing SLAM in 2D and 3D, i.e., simultaneously building a map and localizing the robot in the generated map. Some algorithms focus only on localization of the robot in a pre-built map. In this research work, we used both strategies and explored some algorithms. For 2D map creation Gmapping [61] and Hector mapping [35] were used, and for localization AMCL [54] was used. Some 3D localization algorithms like A-LOAM [22], LegoLOAM [64], and HDL Graph SLAM [37] were also used to understand the 3D pose tracking of the robot.

3D pointcloud to 2D scan

If a 3D lidar or camera sensor is used for mapping and localization, it is necessary to convert the incoming 3D data into a 2D planar laser scan. For a 3D lidar, this can be achieved by using the *pointcloud_to_laserscan*[55] package that takes a 3D input cloud (sensor_msgs/PointCloud2) from lidar sensor and publishes a 2D laser scan (sensor_msgs/LaserScan). Similarly for a RGBD camera, a package *depthimage_to_laserscan*[56] is available that subscribes to *image* topic (sensor_msgs/Image) and *camera_info* topic (sensor_msgs/CameraInfo), and publishes a 2D laser scan topic (sensor_msgs/LaserScan). Both of these packages provide several configuration parameters that can be modified to work on different kinds of lidars and cameras.

2.6. Maps in ROS

As discussed in the previous sections, an occupancy grid map comprises cells containing numerical information about the presence of an object or free space in the cell. This research work focuses on autonomous navigation in a pre-built 2D map created from a 3D BIM model. In this section, the details of map creation from BIM are discussed.

2.6.1. Map from BIM

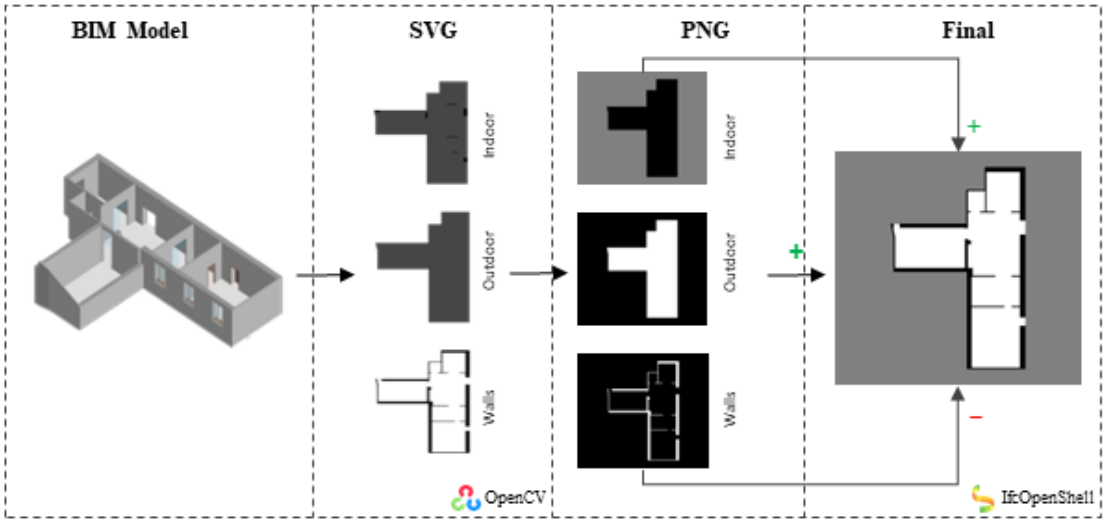


Figure 2.6.: 2D Map from BIM model

It is imperative that the BIM model accurately defines the core structure of the environment. Missing or outdated information can result in an inaccurate or wrong map generation. The strategy chosen for this work is first to extract the geometrical information of the 3D BIM model, then create a 2D image of the floor on which the robot will be navigated. Several tools can convert a 3D BIM model into other file types or generate 2D images from the model. The primary tool used for this work was *IfcConvert* [10], an application from the open source project *IfcOpenShell* [12] that provides several tools for managing or modifying IFC based BIM models. *IfcConvert* provides several options to convert 3D BIM models of *.ifc* file format into other file formats like 3D meshes *.obj*, *.dae* or 2D layers like *Scalable Vector Graphics (.svg)* [11].

SVG and OpenCV

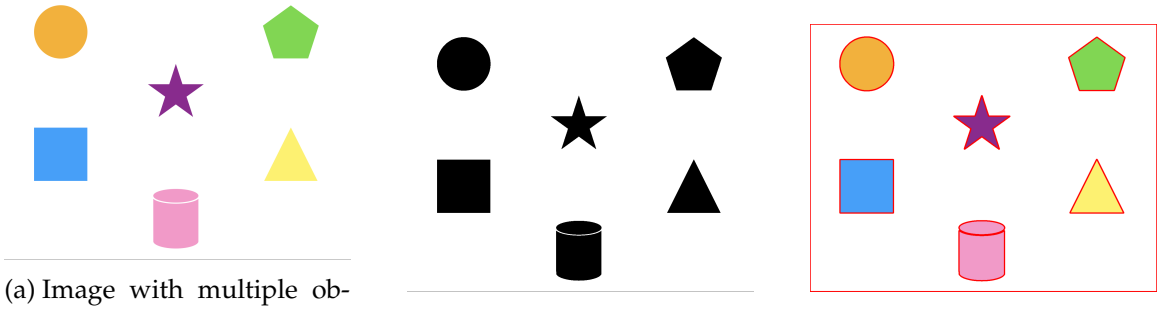
Scalable Vector Graphics is a widely used 2D vector graphics format written in *XML* language. Compared to other 2D image formats like PNG, which are pixel-based, SVG files

consist of a complex mathematical setup that is made up of dots, lines, and shapes. This information is available in XML format and hence can be easily searched for semantic information. Another advantage of SVG is that it can be easily extended or compressed, while other pixel-based images lose the granularity and clarity when scaled up.

After obtaining the SVG file, a corresponding map is drawn on a blank digital canvas using the popular library *OpenCV* [4]. OpenCV (Open Source Computer Vision Library) is a cross platform open source library widely used in research and industrial computer vision development. It provides a vast number of functions and algorithms for image processing and image conversions. The SVG file was converted into a *Portable Network Graphics (PNG)* format file for easier image processing. The obtained PNG file provides more image processing flexibility than the SVG file. This library was used mainly to transform this PNG file into a 2D occupancy grid map. Some of the openCV functions used to process the PNG image and generate the required 2D map are detailed below.

Opencv functions

The OpenCV library was mainly used to perform grayscale conversion of images, binary thresholding to convert the input image into a black and white image, and for detecting object contours.



(a) Image with multiple objects

(b) binary threshold

(c) contours

Figure 2.7.: Contour detection with OpenCV

- *cv2.cvtColor*: This function converts the color space of the input image into another color space. It was used to convert the image into a grayscale image in this research work.
- *cv2.threshold*: This function converts a grayscale image into a binary image (Black and White). It checks each pixel in the image and its value against an input threshold. Values lower than the threshold are set to 0 (black), and the values higher than the threshold are set to the maximum value (white). Figure 2.7b shows an example of an image generated after applying a binary threshold to figure 2.7a.

- *cv2.findContours*: This is a function to detect and store information about the contours or borders of objects in an image
- *cv2.drawContours*: This is a function to draw the contours of objects in an image based on input boundary points. Figure 2.7c shows an example of an image generated after finding and drawing the contours of objects in the image from figure 2.7a. As seen with this function, the contours of an object are drawn in red.
- *bitwise_not*: Then we perform a bitwise inversion operation to invert the colors, i.e., convert black to white and white to black and save the image as a png file.
- *cv2.morphologyEx*: This is a type of morphological function from openCV that help in either increasing or decreasing the borders of an object in the image. This function is useful in removing noise around the contours of an image.

2.6.2. Other Maps

Multiple other ROS based methods can also generate 2D maps. One such way is to utilize the Gazebo tool to visualize a 3D model of the environment and then slice a layer from it by choosing a *z-coordinate* value in the 3D model. The open source project *pgm_map_creator* [75] provides a ROS package and plugin for this. The plugin *libcollision_map_creator.so* is added to a Gazebo world file that represents a 3D environment. Then a configurable ROS launch file is executed to generate the 2D map.

ROS can also work well with 3D maps. A point cloud mesh can be extracted from a 3D model using the CloudCompare [9] tool. Figure 2.8 shows an example of a 3D map generated from the CloudCompare tool. This map can also be converted to other formats like Octomap based on the requirements of the localization algorithm.

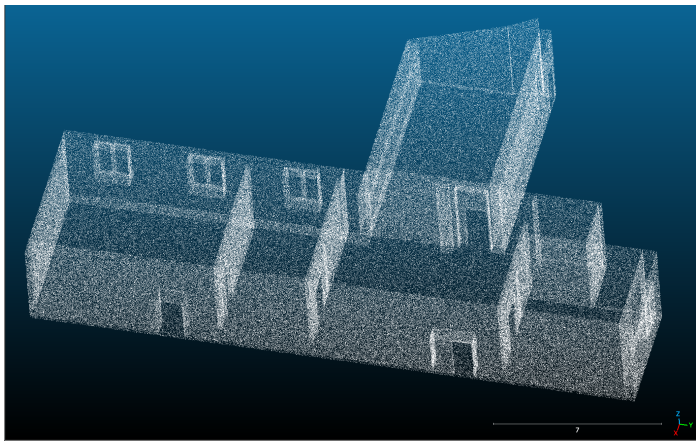


Figure 2.8.: 3D Pointcloud map

3. Related Work

In this chapter, related research and work has been described.

3.1. Autonomous Navigation with Quadruped Robots

In recent years, the navigation of legged robots has become a hot research topic. Multiple research organizations have dedicated time and efforts to enable custom-designed robot development and produced software packages to operate them. The majority of the research in the past decade has focused on utilizing and integrating the solutions into the standard robotics development framework ROS [48]. ROS framework has a large collection of algorithms and packages required for robotics development and testing. ROS started as a personal project at Stanford in 2005 to fix the problem of redeveloping the basic software architecture each time a new study started and focused on utilizing that time for actual robotic application development. Over the next few years, ROS took the shape of an open source project, and multiple libraries and algorithms were added to it. A significant part of general robotic development is based on controlling and moving a robotic agent in the real world. This also led to an increased focus in the area of autonomous navigation, which aimed at building smarter self-reliant systems. The core part of these systems is the ROS navigation stack [42] that was introduced in 2010 to help different kinds of robots to adapt to this stack and achieve autonomous navigation capabilities. The navigation stack is limited to 2D navigation and needs major modifications to comply with the 3D environment. Though the next generation of ROS, i.e., ROS2 [67], was introduced a few years ago, the entire core of this research work will be based on ROS.

The ROS navigation stack requires the robot's odometry to track its pose and path. Additionally, it requires information about the environment that can be generated using perception sensors like Lidar or Camera. Each sensor has its own benefits depending on the research goal. Many popular techniques have been developed to support SLAM in ROS. For research focused on 2D navigation, the *Robot_localization* [43] package from ROS can be utilized to estimate the robot's pose on a map. The map can be either provided before starting navigation in the environment or can be created using SLAM algorithms. For 2D navigation, maps should be based on the occupancy grid map format. For 3D navigation, multiple options exist, like voxel-based maps, point cloud maps, and Octomaps [27].

Many quadruped robots have been developed using ROS. Some of the popular ones used in this research are ANYmal [28] from ETH Zürich, Cheetah [2], and Mini-Cheetah [33] from MIT. Apart from these, multiple commercial robots are available, such as the Spot

robot by Boston Dynamics and a range of different-sized quadruped robots from Unitree - A1, B1, Go1, and Aliengo.

The majority of these quadruped robots utilize ROS control packages to control their movement and path planning for goals. Some notable ROS control projects are OCS2 [16], Free Gait [15], WoLF [49], QuadSDK [45], TOWR [74]. These projects focus on controlling the joints and links in the quadruped robot's legs to navigate them on uneven terrains and at different speeds. *legged_control* [39] is an open source Nonlinear Model Predictive Control (NMPC) and Whole-body control (WBC) based framework built using ROS control and OCS2 libraries. This framework supports the control of Unitree robots and hence would be used extensively for simulating the control and dynamics of the Unitree Go1 robot.

3.2. ROS and BIM

A BIM model contains detailed semantic information about the 3D environment, and this information can be used to aid the autonomous navigation process. This information can be beneficial, especially in indoor areas with many dynamic objects and constantly changing environments. BIM can inform the robot about the static structures in such environments, which can also help track the robot's position. This research work focuses on integrating the information in a 3D BIM model into the ROS navigation stack.

In recent years, multiple research works have focused on integrating BIM models in their industries. In the Augmented Reality field, BIM models have been used to localize the AR devices [25][41] and support emergency response systems by generating user interfaces for indoor localization [17]. In the construction industry, the BIM models are used to automate the monitoring of construction progress [47][65] and cloud-based real-time construction management using RFID devices. Some projects have focused on creating semantic maps from BIM models to support indoor mapping [32] and visualizing obstacles [21][6]. Some deep learning-based methods have focused on improving the robot's localization in an indoor environment. [1][23][3][7].

3.3. ROS based SLAM

SLAM has been one of the key areas of research for the robotic community. Therefore, many popular SLAM solutions are available as ROS packages that work for both Lidar and Camera sensors. Gmapping [61] is one of the most popular 2D SLAM packages in ROS. It relies on odometry data and 2D scans of the environment from perception sensors and works well with a camera or lidar sensor. Hence, it performs poorly when the odometry data is of poor quality. Hector SLAM [35] is another 2D SLAM package in ROS that works quite well even without odometry data. Cartographer [26] is a ROS package that works for multiple varieties of sensors and performs well for both 2D and 3D navigation.

RTAB-Map [38] is a SLAM project initially developed to support visual SLAM based on camera data but now supports multiple types of lidar and camera sensors. Lidar Odometry and Mapping (LOAM) [76] is a popular lidar-based SLAM solution that generates odometry data from lidar scans and creates a map. Many improvements have been made over LOAM, and they are available as open source ROS packages [22][64]. Apart from SLAM solutions, some lidar-based ROS algorithms focus on improving the localization of a robot [5][73]. The most popular localization algorithm for 2D localization is AMCL [54]. This algorithm was the core package used in this research work to localize the robot in a 2D map.

Adaptive Monte-Carlo Localizer

AMCL algorithm

Algorithm Augmented MCL (X_{t-1}, u_t, z_t, m):
static w_{slow}, w_{fast}
$\bar{X}_t = X_t = \emptyset$
for $m = 1$ to M
  $x_t^{[m]} =$ **sample_motion_model**$(u_t, x_{t-1}^{[m]})$
  $w_t^{[m]} =$ **measurement_model**$(z_t, x_t^{[m]}, m)$
  $\bar{X}_t = \bar{X}_t + \left\langle x_t^{[m]}, w_t^{[m]} \right\rangle$
  $w_{avg} = w_{avg} + \frac{1}{M} w_t^{[m]}$
endfor
$w_{slow} = w_{slow} + \alpha_{\text{slow}}(w_{avg} - w_{slow})$
$w_{fast} = w_{fast} + \alpha_{\text{fast}}(w_{avg} - w_{fast})$
for $m = 1$ to M do
  with probability $\max(0.0, 1.0 - w_{fast}/w_{slow})$ do
    add random pose to X_t
  else
    draw $i \in \{1, \ldots, N\}$ with probability $\propto w_t^{[i]}$
    add $x_t^{[i]}$ to X_t
  endwith
endfor
return X_t

The AMCL algorithm is a computationally efficient adaptation of the Monte Carlo localization method that utilizes particle filter to estimate the pose of the robot[68]. In AMCL, the particles are adaptive, i.e., the algorithm utilizes fewer points when the pose is accurately identified but uses a large number of particles when the guessed estimate is poor. In the beginning, the robot's pose is represented by a large number of particles spread

throughout the input occupancy grid map. As the robot moves around and new measurements arrive, the particle filter estimates the likelihood of a particle representing the true position of the robot. The particles with low likelihood are removed in the update cycle, while those with high likelihood are preserved. More particles are added in the area around the most probable pose of the robot before the subsequent measurements arrive. After some time, the particle filter accumulates multiple particles around the true position of the robot. Algorithm 1 describes the particle filtering process in AMCL. The AMCL algorithm is available in ROS as a package and provide multiple parameters that can be configured to suit the need of different robots.

HDL Localization

HDL Graph SLAM[37] is a real-time SLAM algorithm that works well with 3D lidar sensors. It estimates the odometry of the robot by NDT-based scan matching and utilizes loop detection to provide a 3D graph slam solution. This algorithm provides a standalone HDL localization package that functions even in the absence of good odometry data from the IMU sensors of the robot. Robot pose tracking only requires a point cloud map of the environment and the initial position of the robot on the map.

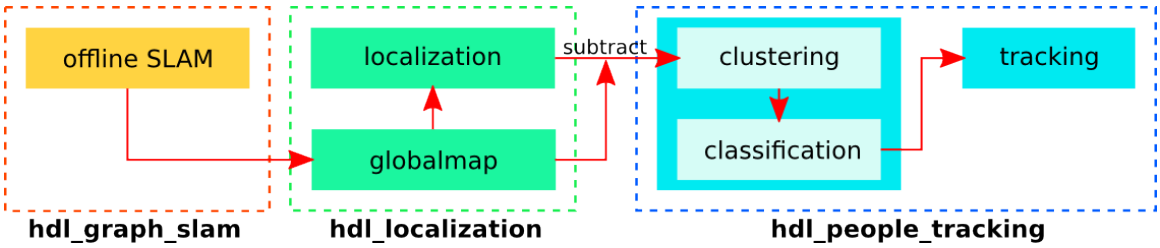


Figure 3.1.: HDL Graph SLAM [36]

3.4. Summary

The latest developments in quadruped robot control and localization methods were researched. The 2D map required for the ROS navigation stack can be generated by running a SLAM algorithm in ROS and navigating the environment. Alternatively, a pre-built map can be used directly to focus only on localization. A 3D BIM model contains vital information about indoor environments. Hence they will be used to generate this 2D map. The generated map will be compared with ROS-based techniques like Gmapping and Hector mapping. Localization in 3D point cloud maps will also be explored.

The OCS2-based open source project *legged_control* [39] can be used as a starting point for creating a gazebo simulation to replicate the real world dynamics of the Go1 quadruped robot. It provides multiple walking patterns or gaits, but this project will primarily use the standard stance and static-walk gait for simulation.

The control packages for the real world are available on Unitree's GitHub repository. However, they are defined in a primitive manner and require significant additional work to plan the robot control in two different states: high and low. The low-state mode is required to fine-tune the walking patterns of the Go1 robot. To avoid redeveloping the basic controls and odometry generation, closed-source packages from the robot supplier MyBotShop were used. They provide a good starting point for implementing the 2D navigation stack. More packages would need to be developed to program the 2D navigation stack fully, and the learnings from the simulation experiments will be added here. (@s: check this sentence)

4. Methodology

4.1. Hardware setup

This section describes the hardware and software used in this research work. A quadruped Unitree Go1 robot [20] is used to perform autonomous navigation in simulation and real world. The Go1 robot is powered by a rechargeable battery system and can easily move in most environments, including under-construction buildings. It supports wireless and wired control and is compatible with ROS based development. The entire framework is set up in ROS, and it runs on a small computer that can be mounted on the Go1 robot along with the Ouster lidar [29][Figure 4.1b], which provides point cloud information of the environment and Realsense camera sensors [51] [Figure 4.1c] for visual pointclouds. The Gazebo simulator and Rviz tool were used to simulate and visualize the framework.

(a) Unitree Go1 [20]

(b) Ouster OS1-32 [29]

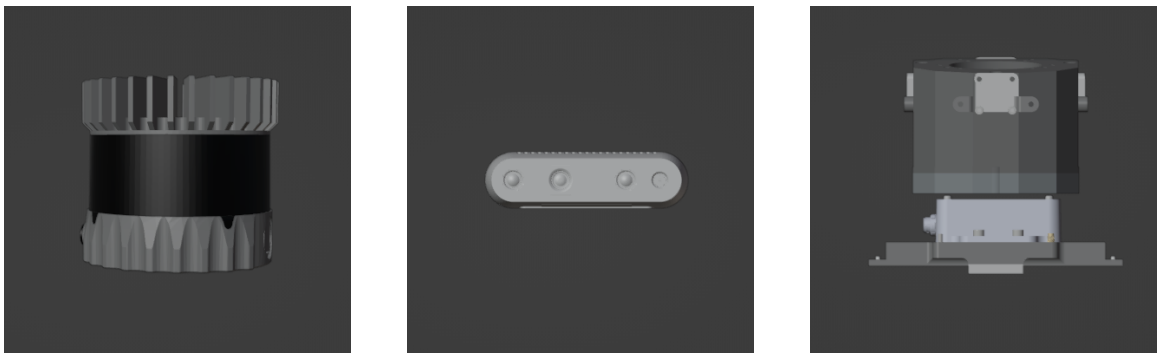
(c) Realsense D435i [51]

Figure 4.1.: Hardware setup

4.2. ROS Integration

A ROS based simulation framework is necessary before attempting to autonomously navigate the Go1 robot in the real world environment. As mentioned in Chapter 2 , ROS provides several simulation tools, among which Gazebo and Rviz were primarily used here. Unitree Go1 recommends using ROS melodic environment but also supports ROS noetic. Therefore, ROS noetic was chosen to test the autonomous navigation capabilities of the Go1 robot. Other ROS based tools, such as Rqt-based graphical tools, were used to send commands to the robot, visualize the flow of information and verify the transforms between different sensors.

4.2.1. Basic setup



(a) Ouster lidar (b) Realsense camera (c) Mounting frame

Figure 4.2.: 3D Meshes

```
$ rosrun xacro xacro go1_robot.xacro > go1_robot.urdf
```

Source Code 4.1.: Generate *URDF* file from *Xacro*

The most crucial requirement for setting up a ROS based framework is correctly defining the coordinate transforms between different sensors; otherwise, the incoming data will make little sense. For the Go1 robot, Unitree provides an XML-based *Xacro* file detailing the transforms and other information (weight, translation, rotation info) on its open source GitHub repository [53]. This *xacro* file was modified, and information about the mounting [71], Ouster lidar, and Realsense camera was added to this file. Once the final *xacro* file is configured, it can be converted into a *URDF* file by using Code 4.1. This *URDF* file can be used to visualize the entire system as a whole in Gazebo or Rviz. The transforms of

individual joints and links can be visualized as well, and any issues in transformation can be resolved.

4.2.2. OGM from BIM

As discussed in Chapter 2 , to create a map from the input BIM model, the model should represent the 3D environment as accurately as possible. The BIM model need not contain information about all the objects in the environment, but basic information like Floor and Wall structure should be present. The accuracy of the input model will decide the accuracy of localization in the 2D map. This algorithm was developed using Miguel Vega's work [69] [70] on creating 2D occupancy grid maps from a BIM model.

The input model is first converted into an *SVG* format file [11] and then processed using *OpenCV* library to generate distinct layers as *PNG* format files. These layers will then be further processed with *OpenCV* tools to create the final *PGM* map compatible with the ROS navigation stack. For compatibility, the 2D map should represent the outdoor area in gray color, the free space on the floor in white color, and the objects (including walls) that can cause a collision marked in black color. The process to generate this 2D map is detailed in this section.

Create SVG

```
$ # Command to generate SVG file using IfcConvert tool
$ IfcConvert <Input IFC filename> <Output SVG filename> <Geometry Options>

$ # Example
$ IfcConvert model.ifc model.svg --exclude entities IfcFloor IfcSpace
```

Source Code 4.2.: Convert *IFC* file to *SVG* file

The open source project *IfcOpenShell*[12] provides a tool *IfcConvert*[10] to convert a 3D BIM model in *IFC* format into a 2D *SVG* floor plan. The command to perform this conversion is listed in Code 4.2. The parameters required to perform this conversion are:

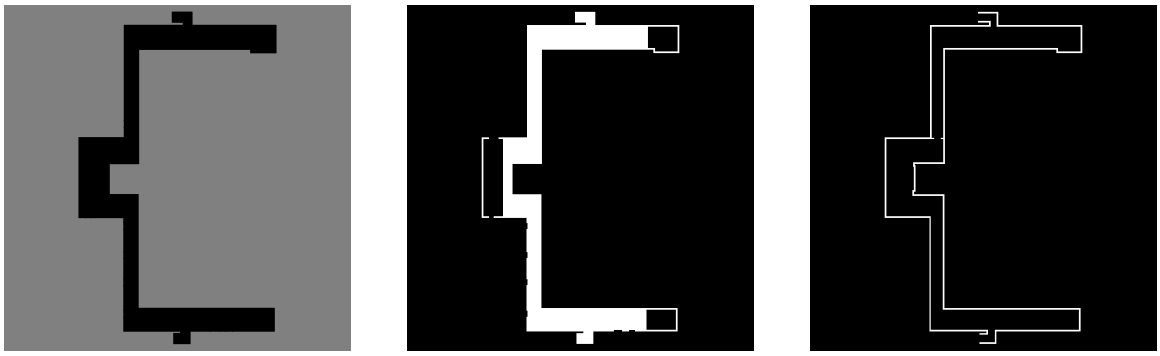
- *IfcConvert path:* Path of the folder where *IfcConvert* executable file is located
- *Input filename*: Path and filename of the input IFC file
- *Output path*: Path and filename of the output *SVG* file
- *Geometry Options:* These define the geometry of the output *SVG* files. For example, the option *-section-height* is used to specify the *z-coordinate* or height in the BIM model from where the 2D layer will be sliced

- *Entities*: Parts of the IFC file that should be included or excluded like *IfcWindow, IfcDoor, etc.*

Extract Layers

An *SVG* file is just a 2D layer of the 3D model; hence all the necessary information cannot be extracted with just one layer. Multiple such layers are necessary, which can then be superimposed on each other to create the final map containing information about the indoor area, outdoor area, and the borders separating them. These slices can be extracted using the *-section-height* option in *IfcConvert*. The height (*z-coordinate*) of the layer should be selected carefully, as a minor change could affect the final result.

For superimposing images and drawing on them, it is better to first convert the *SVG* files to other image formats like *PNG* files and then use the OpenCV library for processing them as discussed in Chapter 2 . The goal is to draw three different layers representing the indoor area where the robot can drive, the outdoor area where navigation is not possible, and the walls structure separating the two areas.



(a) Outdoor layer (b) Indoor layer (c) Wall layer

Figure 4.3.: Image layers

1. *Outdoor layer*: This layer should differentiate between outdoor and indoor areas. Accordingly, the 3D model can be drawn as a black colored silhouette on a gray colored canvas representing the outdoor area. To achieve this, the *z-coordinate* point should be chosen carefully to ensure the representation of the entire model floor area. The best option is to choose the point from the floor entity *IfcSlab* from the IFC file. Other IFC entities like *IfcSpace* and *IfcWindow* can be excluded for faster processing times. After selecting the *z-point* and creating an *SVG* file, it was converted into a *PNG* file for drawing the indoor and outdoor layers.

   The easiest way to differentiate between the indoor and outdoor areas in the image is to use the contour detection method from the OpenCV library to find the borders

of the 3D model. For this, the *PNG* image was first converted into a grayscale image, and *Binary thresholding* was applied to the image, resulting in a black and white colored image. Afterward, to detect the contours of this black and white image, the *findContours* method from OpenCV was used. All the contours detected in the image are not necessary for this layer. The focus is on the contour that separates the indoor and outdoor regions. Generally, the outermost contour represents the borders of the entire image. This contour can be used to draw a plain gray-colored canvas whose size is the same as the original image's size. The rest of the contours depend on the model's contents and represent the entire model's shape and shapes of individual rooms or objects. For this layer, information about the indoor areas is not required; hence, they can all be painted in a single color. OpenCV provides simple pixel-wise addition functions to superimpose images. Thus it would be better to paint the indoor area in black color (*pixel value* $= 0$) and later add colored layers (*pixel value* > 0) on this. Figure 4.3a shows an example image of this layer.

2. *Indoor layer*: This layer should highlight the entire floor area in the BIM model that the robot will access for navigation. For multi-story buildings, this layer should be chosen at the floor level on which the robot will move around. The movable floor area should be painted white, while the rest of the image can be painted black. The floors lower or higher than the one of interest will be ignored and painted black. The model's outline should be maintained, which requires that the walls should also be painted white. Similar to the previous layer, the best option is to choose the point from the floor entity *IfcSlab* from the IFC file. Other IFC entities like *IfcWall* and *IfcWindow* can be excluded for faster processing times.

   After selecting the *z-point* and creating an *SVG* file, it was converted into a *PNG* file for drawing the indoor and outdoor layers. Since this layer requires information about both walls and movable area, contour detection is not necessary as we do not want to paint the indoor area in a single color. Instead, a simple binary thresholding operation can be performed after converting the *SVG* file into a grayscale *PNG*. This results in the outdoor area being painted white and objects of interest (wall, floor) being painted black. However, we need them reversed, so another operation from OpenCV *bitwise_not* that inverts the colors of the binary image will be used to convert the outdoor area to black and objects of interest to white. Figure 4.3b shows an example of this layer.

3. *Intermediate layer*: The Outdoor and Indoor layers can now be added using the addition operation from OpenCV library to create an image that contains the outdoor area in gray color and the indoor movable area in white color. However, as it can be seen in Figure 4.3b, the *Indoor layer* is filled up with white color and cannot differentiate walls from free space on the floor. Thus, the resulting image will also be unable to differentiate between walls and movable areas. Information about walls is necessary to avoid collisions. Therefore, we need another layer that will add the wall

information distinctly to this image.

4. *Wall layer*: The wall layer contains the outlines of all walls that separate rooms on a single floor and define the borders of the model. For this layer, the z-coordinate is chosen such that it captures the information about all the walls in the model while ignoring the floors. In particular, the selected point must not be located along a window as it might be represented as a door in the extracted 2D SVG file and hence depict an opening where the robot can walk through. The *SVG* layer is created by excluding some entities - *IfcSpace*, *IfcOpeningElement*, *IfcDoor*, *IfcWindow* and *IfcSlab* for faster processing time and preserving only the information about walls.

   To add the wall information to the image generated after the addition of *Indoor* and *Outdoor* layers, a simple operation can be performed in OpenCV where the walls can be painted white in color (*pixel value* = 255), other areas painted black in color (*pixel value* = 0) and then this image is subtracted from the intermediate layer (Outdoor + Indoor layer). We do not want to alter the movable areas, so they should be represented by black color, which would not cause any change during subtraction.

   The generated SVG file contains walls in black color and other areas in white color. First, the *SVG* file is converted into a grayscale *PNG*, and then *Binary thresholding* is applied to it. The resulting image will have other areas painted white and the walls painted black. We need them reversed, so the pixel inversion operation *bitwise_not* from OpenCV is applied, which converts the outdoor area to black and the walls to white. Figure 4.3c shows an example of this layer.

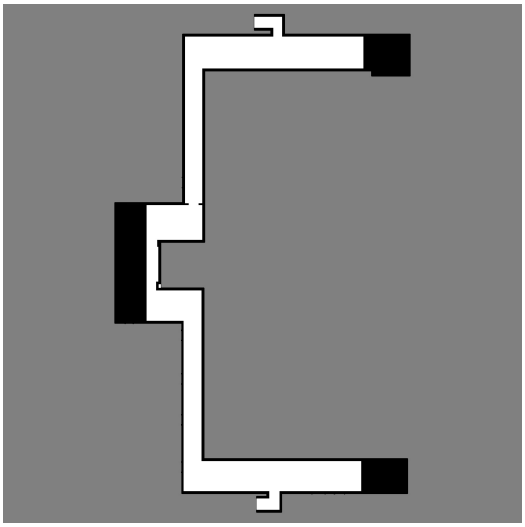


Figure 4.4.: Final *PGM* image

5. *Final image*: As discussed above, the walls layer can now be subtracted from the intermediate image obtained after adding indoor and outdoor layers. The resulting

image will contain all walls in black color, the movable floor area in white color, and the outdoor area in gray color. There are possibilities of mismatch in the alignment of the walls at border areas. The morphological transformation operation from OpenCV *morphologyEx* can be used to fill these gaps. This operation results in the desired final image that should be saved in *PGM* format as ROS expects occupancy grid maps. In addition, the respective *YAML* file that defines the resolution and origin coordinates of the model can be generated by extracting the origin coordinates from any one of the SVG files. Figure 4.4 shows an example of the final 2D map.

4.2.3. 3D Map from BIM

As discussed in Chapter 2 , multiple ROS based algorithms are available for robot localization based on input 3D perception data from a 3D lidar or camera. Most of these algorithms require a 3D point cloud (*PCD*) map of the environment. The 3D map can be created by the 3D visualization tool *CloudCompare*[9]. It provides an option to sample points from a 3D mesh. Therefore, the input BIM model was first converted into a Wavefront (*.obj*) file and then sampled in *CloudCompare*. Figure 4.5 shows an example of a 3D point cloud map created using *CloudCompare*.

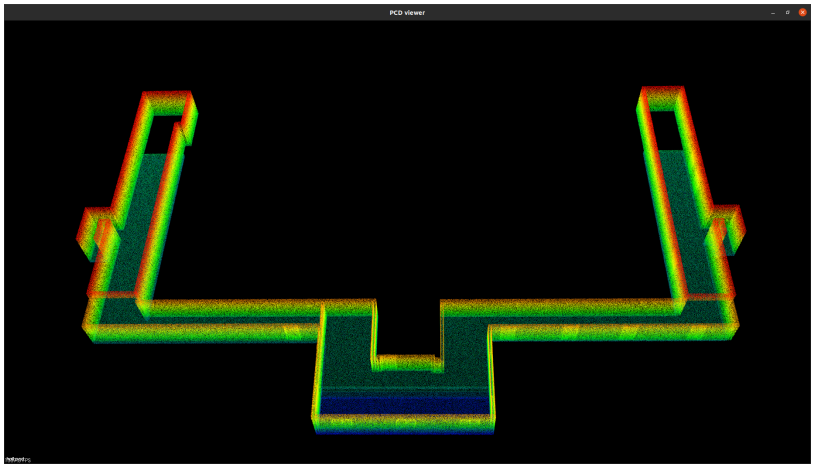


Figure 4.5.: 3D point cloud map

4.3. Simulation

This section discusses the implementation details of the ROS navigation stack for simulating autonomous navigation. This simulation setup was based on the open source project from Qiayuan Liao [39] that uses OCS2[16] and ROS control[8] libraries to operate a Unitree A1 quadruped robot in gazebo simulator and operating it in the real world. It provides

a simple mechanism to simulate various gaits for Unitree's quadruped robots and add customizations for navigation. The existing project was modified for this research work by adding the necessary Unitree Go1 configuration (xacro) files and additional sensors as discussed in Section 4.2.1 .

4.3.1. Gazebo environment

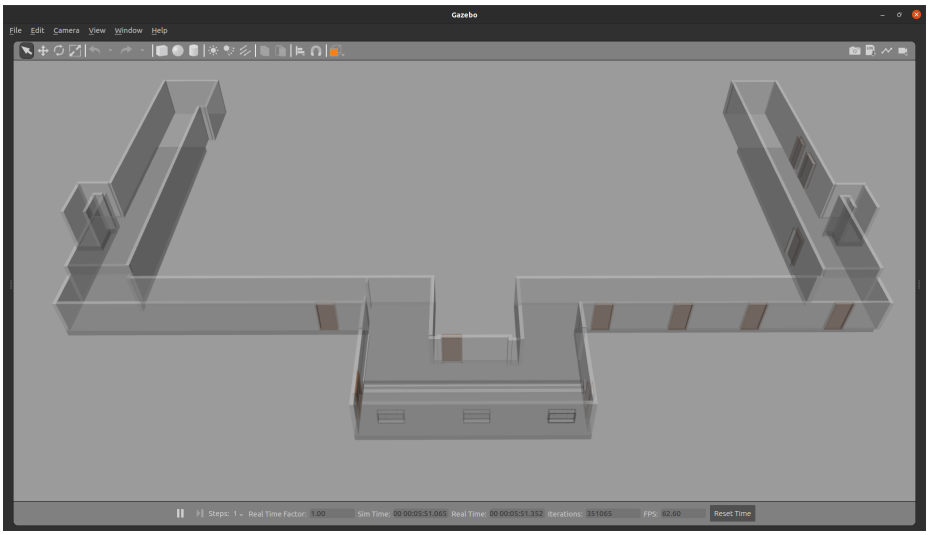


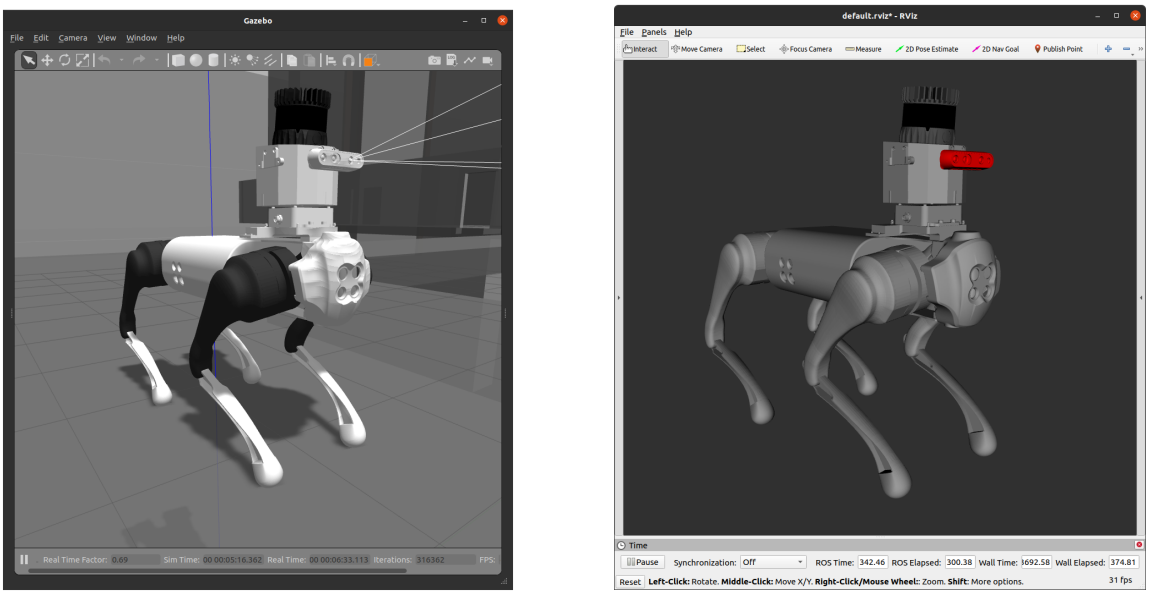
Figure 4.6.: Custom world in Gazebo

The first step is to create a custom simulation environment that accurately reflects the input BIM model, thus ensuring a true representation of the real world. Gazebo simulator is used in this research work to create the simulation environment. Gazebo requires the environment to be described and configured in a Simulation Description Format (SDF) file, an XML-based format that utilizes 3D meshes or nested models to create a simulation environment. Therefore, the input BIM model (.ifc) is first converted to a Collada mesh file (.dae), which can then be added and resourced by an SDF file. This SDF file, along with a basic configuration file can now be referenced by a Gazebo world file (.world) along with other additional models or resources (e.g., a ground plane or a light source). The resulting world file looks as shown in figure 4.6. This world file can now be loaded into a gazebo environment using a simple ROS launch file.

4.3.2. Configuration

The next step adds all the necessary configuration files and three-dimensional mesh files of the quadruped Go1 robot, lidar, and camera sensors through a URDF file.

- **Unitree Go1**:



(a) Gazebo (b) Rviz

Figure 4.7.: Visualization of Go1 robot

The configuration files and mesh representations in collada format (.dae) were imported from Unitree's ROS GitHub repository [53]. The sensors and the mounting system information were added to this URDF file. Further, minor modifications were made to match the requirements of Qiayuan Liao's [39] project.

- **Ouster Lidar** (*libgazebo_ros_ouster_laser.so*):

  For simulating the 3D point cloud lidar data, Wil Selby's blog[63] was followed. The necessary mesh file and plugin to simulate lidar in Gazebo were used from Wil Selby's[62] and Gepetto team's[18] git repositories.

- **RealSense Camera** (*librealsense_gazebo_plugin.so*):

  For simulating the RealSense D435i camera, the gazebo plugin was imported from Pal-Robotics Github repository[66]. The 3D mesh files were used from Intel's ROS Github repository for RealSense cameras[50].

- **ROS control** (*liblegged_hw_sim.so, libgazebo_ros_p3d.so*):

  The plugins for controlling the quadruped robot were used without any modifications from Qiayuan Liao's [39] project.

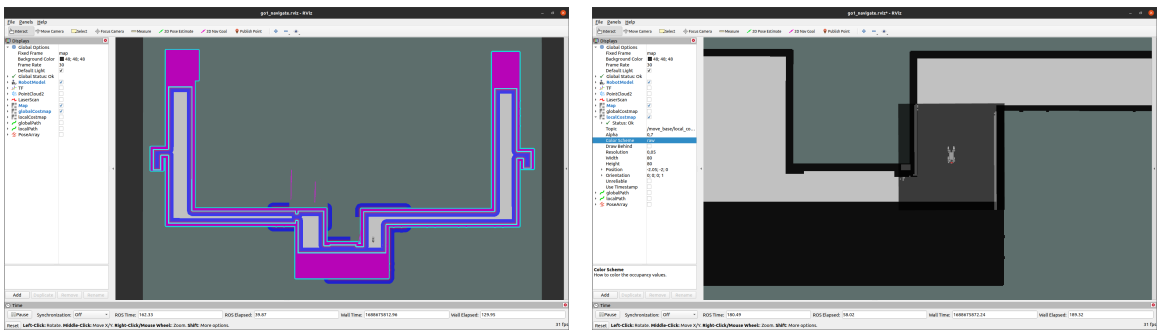
Figure 4.7 shows the visualization of the Go1 robot with a mounting frame on which Ouster Lidar and Realsense camera are mounted.

4.3.3. Manual Control

Before setting up the planners and costmaps for autonomous navigation, it should be verified if the quadruped robot can move around in the custom gazebo world based on user-input velocity commands. After loading the custom world in the Gazebo simulator, the quadruped robot and its controller should be started using the Rqt-based graphical tool *rqt_controller_manager*. This tool allows the user to switch on the robot's movement, i.e., from a static stance to walking or running. After the controllers are launched, the robot can be given velocity commands either through the command line interface or by using the Rqt-based graphical tool *rqt_robot_steering*.

4.3.4. Costmaps and Planners

As discussed in section 4.2.2, to create the costmaps for navigation and obstacle avoidance, the map generated from the BIM model will be used. For the 2D obstacle layer in the costmaps, a 2D laser scan of type *sensor_msgs/LaserScan* is required. Therefore, a *pointcloud_to_laserscan* node [A.2.2] was used to convert the 3D point cloud from Ouster lidar to a 2D laser scan of type *sensor_msgs/LaserScan*. For the 3D obstacle layer, a source that provides messages of type *sensor_msgs/PointCloud* is required. Hence, either the point cloud from the Ouster lidar or the Realsense camera can be used. The global planner was set up using the default Dijkstra's algorithm. For the local planner, DWAPlannerROS [58] algorithm was used. The costmaps are visible on the 2D map as soon as the 2D laser scan and map topics are available. The local and global plans can be seen by setting a target goal point where the robot has to navigate autonomously. This goal can be easily set using the *2d Nav Goal* option in Rviz. Figure 4.8 displays the global and local costmaps drawn over the input BIM-based map.



(a) Global Costmap (b) Local Costmap

Figure 4.8.: Costmaps

4.3.5. Localization

Another prerequisite for 2D autonomous navigation is the robot's initial position in the input map. The initial pose can be set using the *2D Pose Estimate* option in Rviz or by including the information in a ROS launch file. After setting the initial pose, the robot can be given a target goal to navigate to, but this also requires that the robot knows where it is moving on the map. The odometry frame provides this information but is prone to errors or drift during long-term navigation. As discussed in Chapter 2 , this issue can be fixed by using the AMCL algorithm for localization that requires only a 2D laser scan of the environment and is capable of scan matching, i.e., matching edges and contours of the environment.

4.3.6. Summary

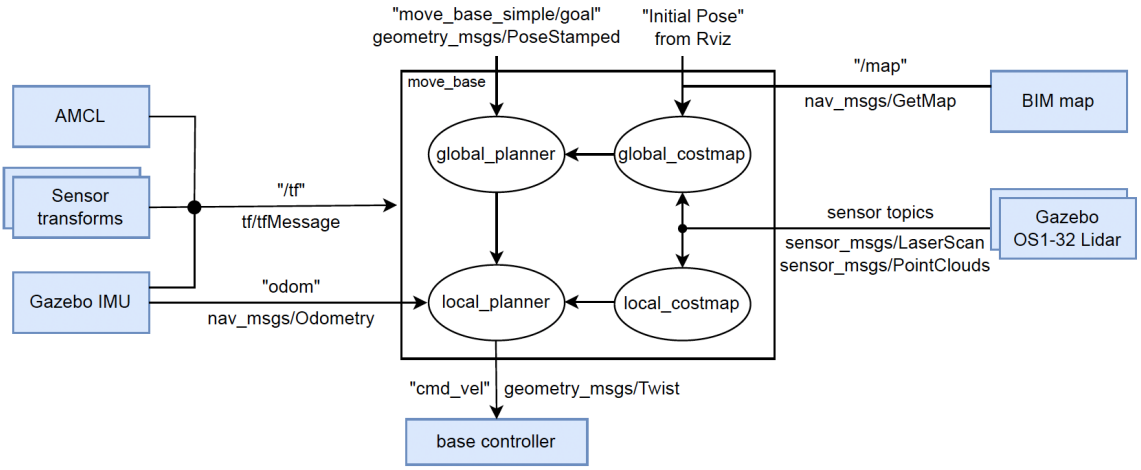


Figure 4.9.: Simulation Navigation Stack

In brief, the simulation process for autonomously navigating the Go1 robot in a BIM based world can be summarized as below:

1. Launch the custom environment using the Gazebo world file.
2. Launch the ROS controllers to start up the robot in the simulation environment.
3. Launch the pointcloud to laser scan node to convert 3D lidar data to 2D scans.
4. Launch the map server node to load the BIM based occupancy gridmap.
5. Launch the AMCL node to localize the robot on the input map.
6. Launch the move base node to load the costmaps and planners.

7. Set the inital pose of the robot on the map and provide a target goal via Rviz.

4.4. Real world autonomous navigation

This section explains the ROS-based framework required for autonomously navigating Unitree Go1 in the real world. This framework is developed using real-time development based on live data and recorded *rosbag* files. *Rosbag* files are better for tuning parameters and verifying the implementation.

Figure 4.10.: Real world environment

4.4.1. Remote control

One of the main aims of this project is to perform autonomous navigation with the Go1 robot by controlling it remotely through wireless control. A remote connection is required from the computer mounted on the Unitree Go1 robot to another computer system to operate the robot wirelessly. One convenient option is to use a Remote Desktop Protocol (RDP) based tool. For this project, *Microsoft Remote Desktop* tool was used as it provides a fast remote connection and accurately replicates the remote system's graphics. However, it requires that both devices should be on the same network. The network was constructed

by creating a Mobile WiFi hotspot from a cellular device, and then this network connection was added to both computers.

4.4.2. Power supply

As mentioned in Section 4.1 , the Unitree Go1 robot can be powered on via its rechargeable battery, and for the sensors and the computer, a set of small rechargeable batteries were used. The Go1 robot and Ouster lidar were connected to the onboard computer via ethernet cables, and the Realsense camera was connected using a *USB Type-C to Type-C* cable [72]. After switching on all the devices, a remote connection is established between the devices. Then the ROS system can be launched. Figure 4.11 displays the setup on a Go1 robot before and after switching it on.

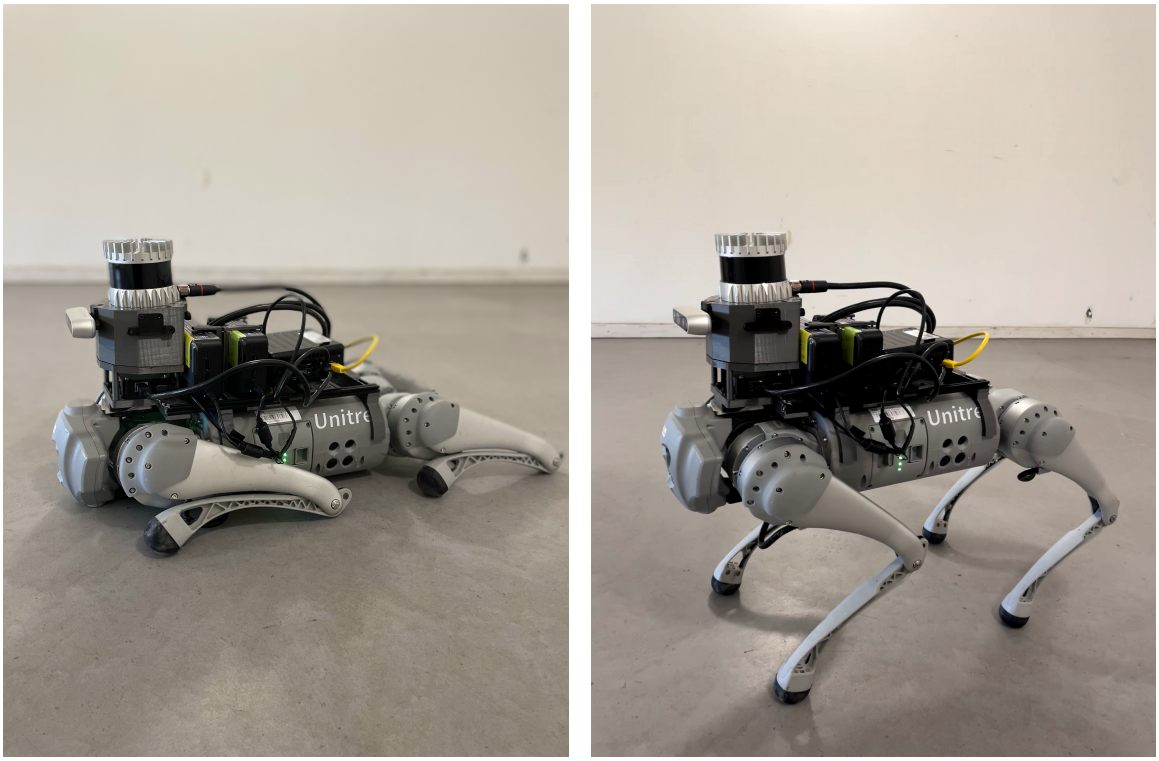


Figure 4.11.: Unitree Go1 setup

4.4.3. Basic configuration

The URDF files from Section 4.2.1 and mesh files from section 4.3.2 were used for the basic configuration. As discussed in section 4.2.2, the map generated from the BIM model was used for real world 2D navigation. Additionally, the private project from the robot supplier

MYBOTSHOP [44] was used to connect the Unitree Go1's controller system to the onboard computer and send velocity commands through the *cmd_vel* topic. The velocity commands can be either sent by a keyboard, *rqt* based plugins, or the navigation stack. This project also provides an Extended Kalman Filter based robot localization node to localize the robot in the map using Unitree Go1's odometry. Additionally, it provides launch files to generate the odometry of the Go1 robot from its IMU sensor data.

4.4.4. 2D scan

Ouster Lidar

Similar to simulation, the ROS navigation stack requires a 2D laser scan in real life. The ROS drivers provided by the Ouster ROS project[30] were used to generate the 3D point cloud data from the Ouster lidar. A simple launch file *sensor.launch* is provided as part of the driver package to generate the point cloud data. The launch file requires the Ouster sensor name (unique for each product) and an ethernet connection between the Ouster lidar and the computer. Thus, the sequence is first to launch the *sensor.launch* file and then use *pointcloud_to_laserscan.launch* from Section 4.3.4 to generate the 2D laser scan.

Realsense Camera

Intel Realsense also provides ROS drivers to operate the D435i camera and visualize its data [50]. It provides a simple launch file *rs_camera.launch* that can automatically detect the sensor type and create ROS topics that can be visualized in Rviz or used for further processing. A 2D laser scan can also be extracted from the 3D camera data using the *depthimage_to_laserscan*[56] node from ROS. A launch file *depthcamera_to_laser.launch* was created to perform this conversion in real-time.

4.4.5. Simulation Packages

The move base node configuration files to load costmaps and planners on the input BIM based 2D map were taken over from the simulation setup. Additionally, the launch files to load the AMCL node was also taken over from simulation setup.

4.4.6. 3D Localization

To supplement the 2D navigation some ROS packages for 3D localization that directly use 3D point cloud from lidar were also implemented.

hdl_localization

The hdl_localization for real-time 3D localization was also implemented. As detailed on *koide3/hdl_localization*[36], the *hdl_localization.launch* has to be configured to receive the odom-

etry and 3D point cloud data from the sensors. Then, this launch file can be executed in ROS with ouster lidar, and the data can be visualized in Rviz. Initially, the *aligned_points* topic might be misaligned with the input point cloud map, but this can be easily aligned by using the *2D Pose Estimate* option to set the initial pose of the robot in the input point cloud map.

4.4.7. Summary

The steps required to autonomously navigate the Go1 robot in real world environment are summarized as:

1. Power on all the devices and add the wired connections.
2. Launch Unitree Go1 description and control files to generate odometry data and enable remote control via ROS.
3. Launch the perception sensors (lidar/camera) nodes and convert 3D point cloud data to 2D scans.
4. Launch the move base node and map server node to load the BIM based 2D map, costmaps and planners.
5. Launch the AMCL node for 2D localization.
6. Launch Rviz, set initial pose of the robot on the map and provide target goals.

5. Testing and Validation

In this chapter, the experiments performed during the development of this research work and their results are discussed. The Gazebo simulator and Rviz tool were extensively used to visualize the configuration process and verify if the results are as expected. Other tools used in this work are detailed in Section A.1 .

5.1. Map creation

A 2D map is an essential component of the ROS navigation stack. As discussed in the previous section, the 2D maps were generated using the *IfcConvert* tool, and the OpenCV library was used to paint the maps to distinguish between different layers. To extract the 2D layer slices for the map generation process, the possible range of *z-coordinate* points on the input BIM model were chosen by identifying a range of points using the 3D visualization tool CloudCompare[9]. A simple trial and error method was used to select the height from this range and then visually verify the generated *SVG* files.

5.1.1. Single-floor BIM

The map generation process from the input BIM model in Figure 5.1 is described here. Table 5.1 lists the required input parameters to the map generation algorithm.

1. *Outdoor layer*: Since the input BIM model has only one floor, the model's outline separating the outdoor and indoor areas can be visualized by this one floor. The lowest and highest points on the floor are at $z = -4.15$ and $z = -4.0$ respectively. To extract a silhouette, the z-coordinate was selected from this range $[-4.15, -4]$. Figure

Parameter	Value
contourHeight	-4
overlayHeight	-4
wallsHeight	-3.16
bounds	600x600
scale	50

Table 5.1.: Input parameters

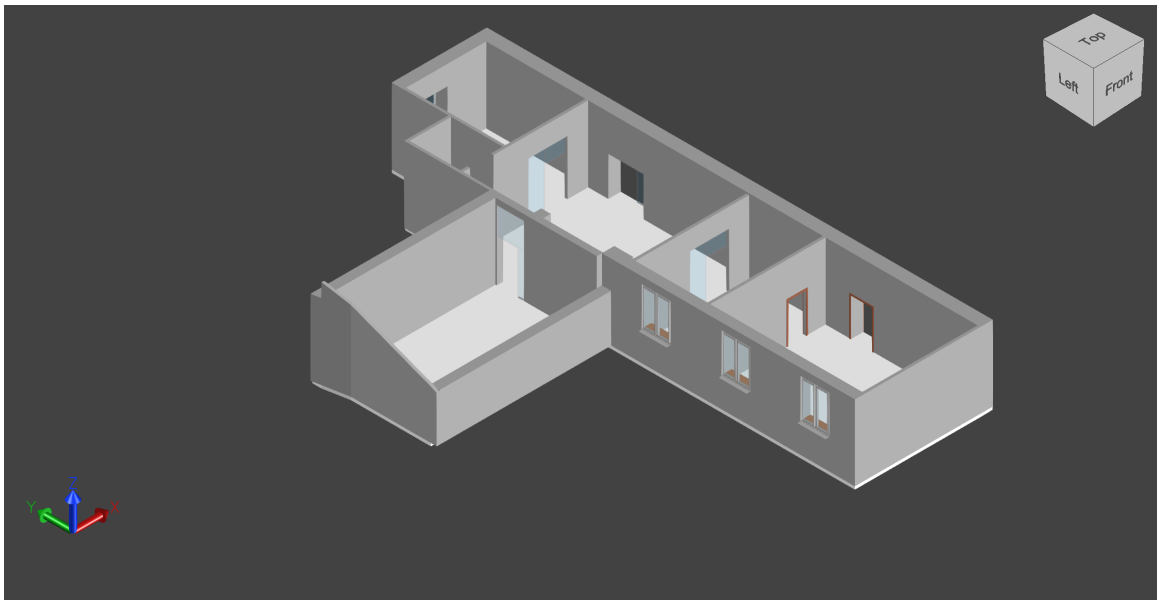


Figure 5.1.: Single-floor BIM [Image generated using Open IFC Viewer[46]]

5.2a shows the *SVG* file generated at $z = -4$ and figure 5.2b shows the generated *PNG* file after processing the *SVG* file with OpenCV.

2. *Indoor layer*: The entire area of the model on which the robot navigates can be visualized by extracting the single floor from the BIM model. To extract the indoor layer, the z-coordinate was selected from the floor's height range $[-4.15, -4]$. Figure 5.3a shows the *SVG* file generated at $z = -4$ and Figure 5.3b shows the generated *PNG* file after processing the *SVG* file with OpenCV.

3. *Border layer*: The input BIM model contains walls of varying heights, but their base is the same floor, i.e., the lowest z-coordinate is the same for all walls. To choose a z-coordinate that represents all walls, the height range should be from the highest point on the shortest wall to the lowest point on the walls. Also, the windows have to be excluded. This range was identified as (-4, -3.15) to avoid doors and windows (around $z = -3.15$). Figure 5.4a shows the *SVG* file generated at $z = -3.16$ and Figure 5.4b shows the generated *PNG* file after processing the *SVG* file with OpenCV.

(a) Outdoors layer SVG (b) Outdoors layer PNG

Figure 5.2.: Outdoor Layer

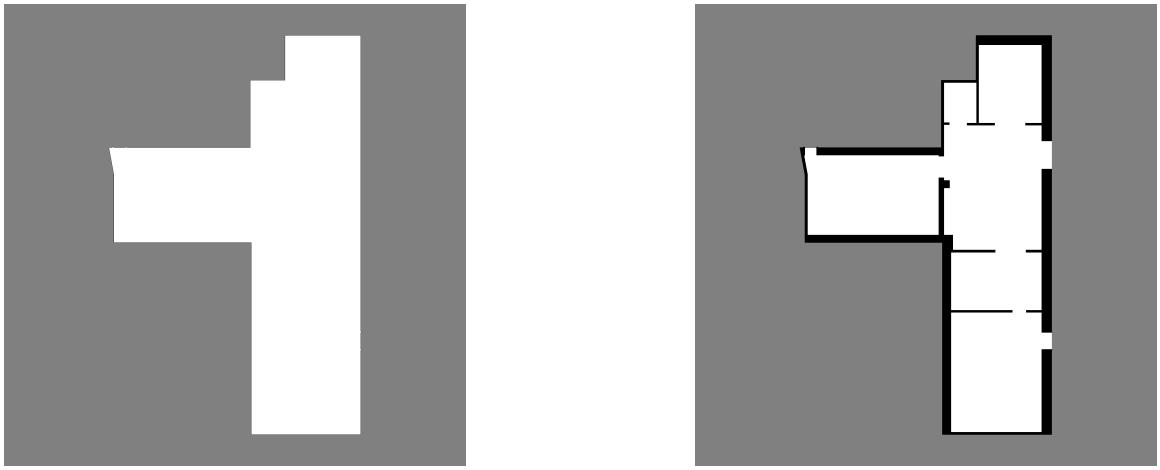
(a) Indoors layer SVG (b) Indoors layer PNG

Figure 5.3.: Indoor Layer

(a) Wall layer SVG (b) Wall layer PNG

Figure 5.4.: Wall Layer

Intermediate and Final images



(a) Intermediate = Outdoor + Indoor

(b) Final = Intermediate - Walls

Figure 5.5.: Final image

After obtaining the three layers, an intermediate layer was generated by adding the Outdoor and Indoor layers using OpenCV. Figure 5.5a shows the intermediate layer. Later, using OpenCV the wall layer was subtracted from the intermediate layer to generate the final image layer. This layer was saved as an occupancy grid map. Figure 5.5b displays the occupancy grid map generated from the BIM model in Figure 5.1 .

5.1.2. Multi-floor BIM

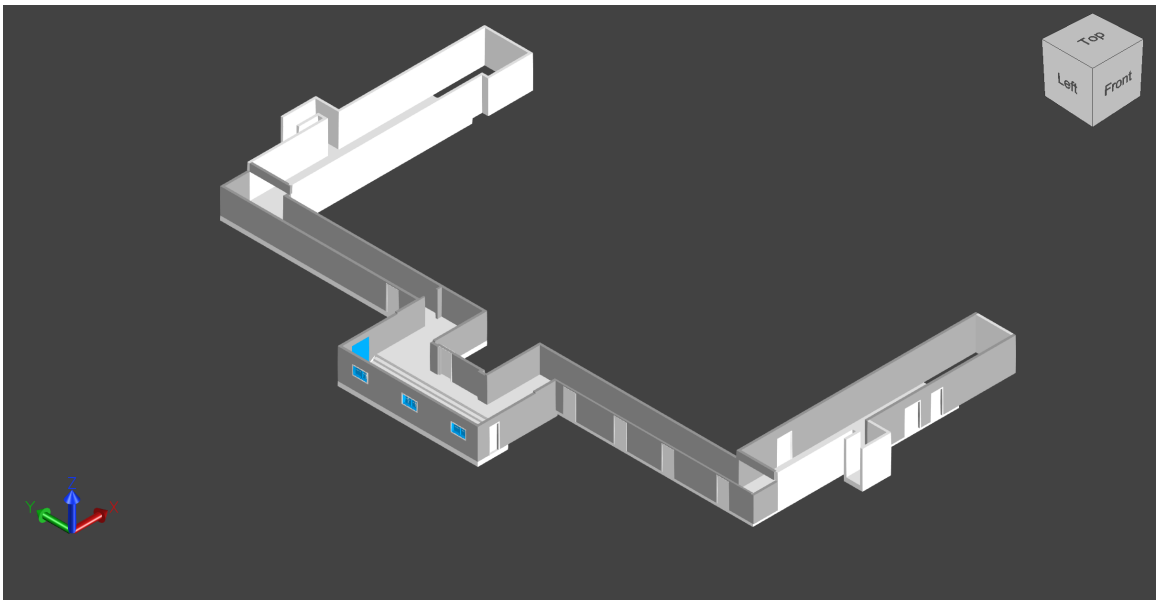


Figure 5.6.: Multi-floor BIM [Image generated using Open IFC Viewer[46]]

Parameter	Value
contourHeight	-3.5
overlayHeight	-3.5
wallsHeight	-1.501
bounds	500x500
scale	100

Table 5.2.: Input parameters

Figure 5.6 describes the map generation process from the input BIM model. Table 5.2 lists the required input parameters to the map generation algorithm.

1. *Outdoor layer*: Since the input BIM model has multiple floors, a map of only one floor is extracted. The outline of the model separating the outdoor and indoor areas can be visualized using the points on this floor. To extract a silhouette, the z-coordinate was selected at $z = -3.5$. Figure 5.2a shows the *SVG* file generated at $z = -3.5$ and Figure 5.2b shows the generated *PNG* file after processing the *SVG* file with OpenCV. As seen, the outline of the entire BIM model is captured in a black silhouette.

2. *Indoor layer*: The entire area of the model on which the robot can navigate can be visualized by using a point from the height of the floor. To extract the indoor layer

(a) Outdoor layer SVG (b) Outdoor layer PNG

Figure 5.7.: Outdoor Layer

the z-coordinate was selected at $z = -3.5$. Figure 5.3a shows the *SVG* file generated at $z = -3.5$ and Figure 5.3b shows the generated *PNG* file after processing the *SVG* file with OpenCV. As seen, only the area of interest is highlighted in white color and other floors are represented as black color obstacles to avoid robot movement.

(a) Indoor layer SVG (b) Indoor layer PNG

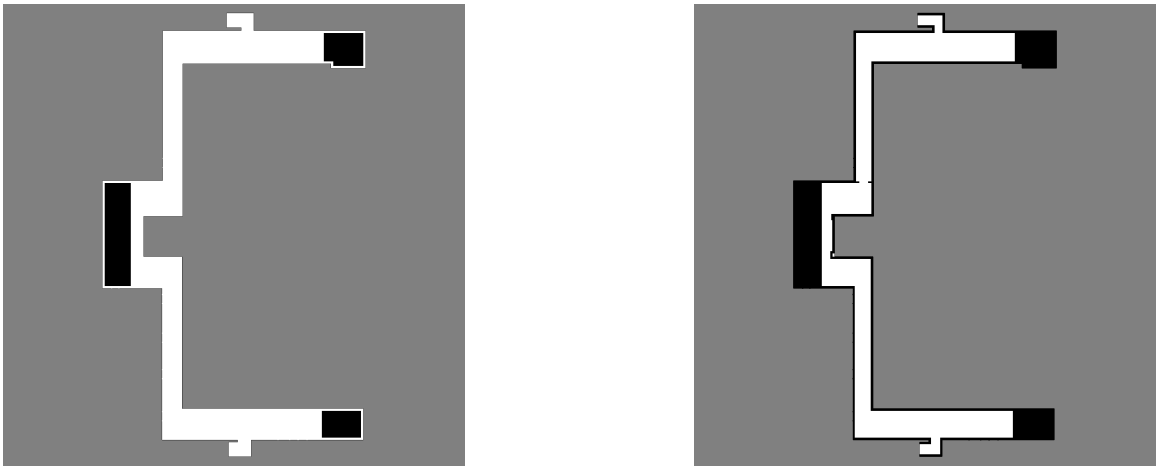
Figure 5.8.: Indoor Layer

3. *Border layer*: The input BIM model contains walls of varying heights and multiple floors. The z-coordinate should be carefully chosen such that it represents the walls around the area of interest accurately. Also, the windows have to be excluded. The z-coordinate that captures all the relevant walls was identified as $z = -1.501$ to avoid doors and windows (around $z = -1.5$). Figure 5.4a shows the *SVG* file generated at $z = -3.16$ and Figure 5.4b shows the generated *PNG* file after processing the *SVG* file with OpenCV. As seen, all the walls have been generated in white color on a black canvas as expected.

(a) Wall layer SVG (b) Walls

Figure 5.9.: Wall Layer PNG

Intermediate and Final images



(a) Intermediate = Outdoor + Indoor (b) Final = Intermediate - Walls

Figure 5.10.: Final image

After obtaining the three layers, an intermediate layer was generated by adding the Outdoor and Indoor layers using OpenCV. Figure 5.5a shows the intermediate layer. Afterward, the wall layer was subtracted from the intermediate layer to generate the final image layer using OpenCV. This layer was saved as an occupancy grid map. Figure 5.5b displays the occupancy grid map generated from the BIM model in Figure 5.1 . As seen, the navigatable area for the robot is highlighted in white, other floors are highlighted in black, and the walls are colored black as well.

5.2. Simulation experiments

This section discusses the results from the Simulation of the Go1 robot in a custom designed Gazebo world. The Gazebo world file was created using the input BIM model, and the Go1 robot and its sensors were loaded into the environment. It is expected that the Go1 robot navigates in this environment as it would in the real world.

5.2.1. Basic configuration

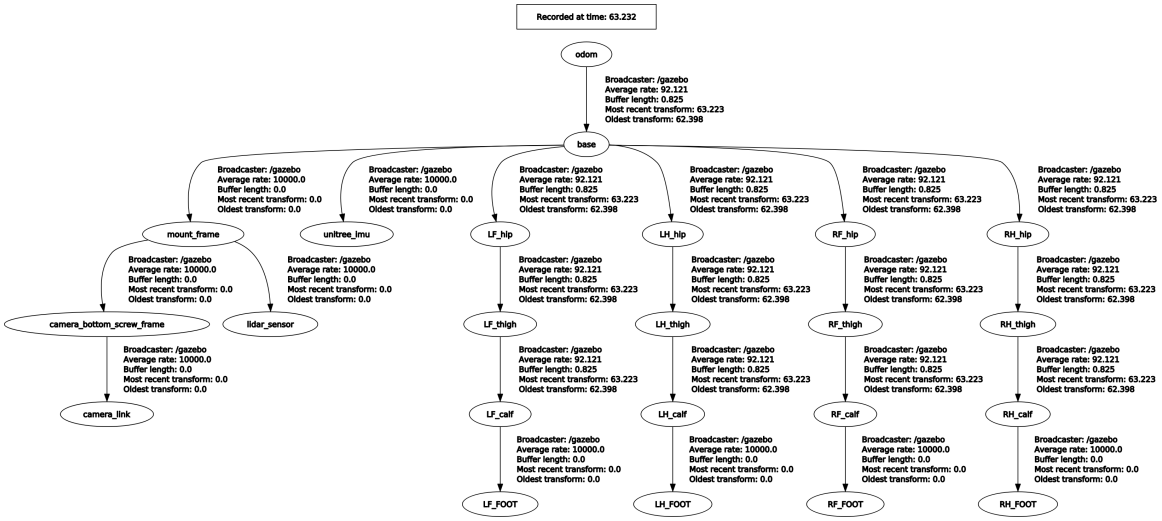
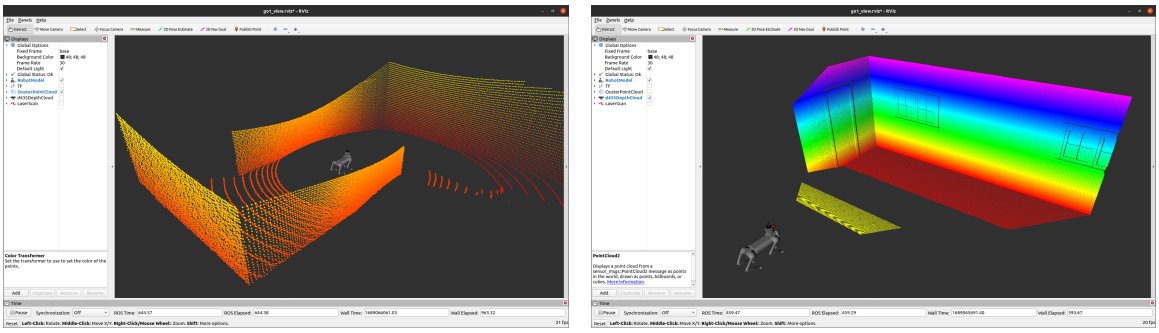


Figure 5.11.: Transform tree of Go1 robot

As discussed in Chapter 2 , the most important requirement is to set up the transfer tree properly,i.e., defining the flow of information and their respective coordinate transforms. After configuring the xacro files and generating a URDF, the Go1 robot was loaded into the Gazebo world. Figure 5.11 displays the transform tree of the Go1 robot in the Rqt-based graphical tool *rqt_tf_tree*. As seen in the figure, the Lidar and camera sensors are linked to the *mount_frame*, which is the frame defined for the mounting system on which sensors are placed. This *mount_frame* is linked to the *base* frame of Go1, where all the other frames supply their data to. An *imu_sensor* is also present that helps generate the odometry of the Go1 robot.

5.2.2. Sensor Data

After the transforms are set up, the sensors should be checked to see if they produce any data and if that data accurately reflects the simulation world. As discussed in Chapter 4 , Gazebo requires a plugin to simulate the sensor data. The plugins used here were



(a) Ouster pointcloud in rviz

(b) Realsense Real data

Figure 5.12.: Sensor data

libgazebo_ros_ouster_laser.so to simulate the Ouster lidar data, which can be configured according to the type of Ouster sensor available (match parameters). For RealSense, d435 was used *librealsense_gazebo_plugin.so*, which can also be configured for range and other parameters.

Figure 5.12a displays the Ouster point cloud in the input BIM model and Figure 5.12a displays the Realsense camera point cloud in the input BIM model.

Convert to 2D laser scan

ROS navigation stack requires that the sensor data should be provided as 2D laser scan instead of 3D point clouds. The *pointcloud_to_laserscan* was used to conveet the 3D lidar data into 2D laser scans. The ROS launch file to perform this conversion is listed in Appendix A.2.2. The *cloud_in* topic was set to the ouster lidar pointcloud topic and the target frame to transform the data was set as the ouster sensors frame. Figure 5.13 shows the lidar sensor data in multiple colors and the 2D scan as white colored points. It can be observed that the walls are detected properly in the generated 2D scan.

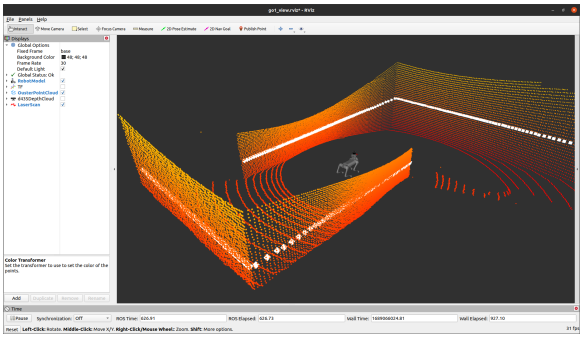


Figure 5.13.: 3D point cloud and 2D laser scan

5.2.3. Manual Navigation

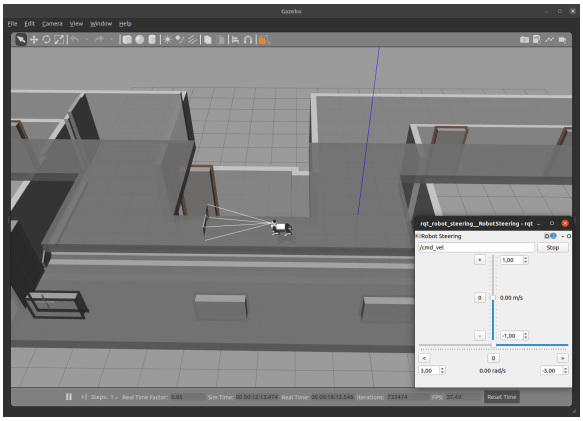


Figure 5.14.: Manually controlling Go1 robot

To check the physical dynamics of the simulation setup, it is required to check if the Unitree go1 robot could move in the Gazebo world. For this, it should be ensured that while launching the go1 robot in the gazebo world, it is initially placed on the floor on which we would like to navigate autonomously. Scenarios of climbing stairs or moving in external areas were not considered for this work.

Figure 5.14 shows the Go1 robot in the custom-designed Gazebo world. As discussed in Chapter 4 , the ROS control-based controllers for the robot were started to move the robot from a static stance to a trot stance. Then, longitudinal and lateral velocity commands were given using the Rqt-based graphical tool *rqt_robot_steering* on the topic *cmd_vel*. It was observed that the Go1 robot responds well to the velocity commands on the *cmd_vel* topic and moves in the Gazebo world as expected. Special care should be taken while moving the robot in the simulation world. If the robot collides with the wall or steps on staircases, it might fall down or display unrealistic behavior, which would end the current simulation session, and the environment would need to be relaunched to start from the beginning.

5.2.4. 2D Localization

After ensuring the robot responds to velocity commands, it should be checked how it localizes itself on the map and how good the alignment is. The AMCL node needs the *scan* topic on which the 2D laser scan data from the lidar sensor will arrive. It also needs the robot's initial position in the input 2D map. The initial pose can either be included in the launch file or set using the *2D Pose Estimate* option in Rviz. After launching the AMCL node, it crashed with an error message caused due to the tilt in the ouster lidar placement. AMCL requires the lidar sensor to be mounted planar to the floor. To solve this issue, the mounting position of the Ouster lidar was changed in the URDF file, and the tilt angle was

set to 0 degrees. This solved the issue, and the AMCL ran without problems. After the AMCL node starts up, the initial pose was set using *2D Pose Estimate* option in Rviz. It was observed that the AMCL node localizes the robot accurately and tracks its movement accurately when it is moved.

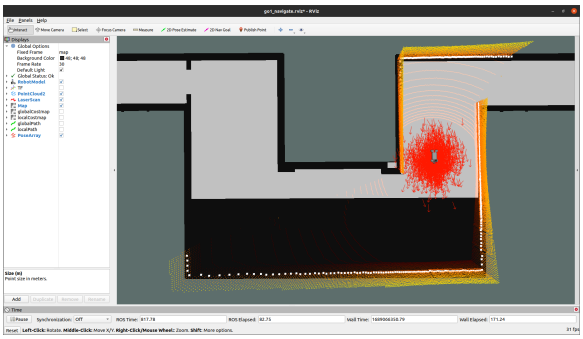


Figure 5.15.: AMCL localization

Figure 5.15 shows the output visualization after setting the initial pose of the Go1 robot in the BIM-based 2D map. The arrows in red surrounding the Go1 robot represent the pose array of AMCL which calculate the probability of the robot's pose on the map. As seen, the robot pose is determined accurately, and the lidar sensor detections match the collision walls of the 2D map.

5.2.5. Path Planning and Autonomous navigation

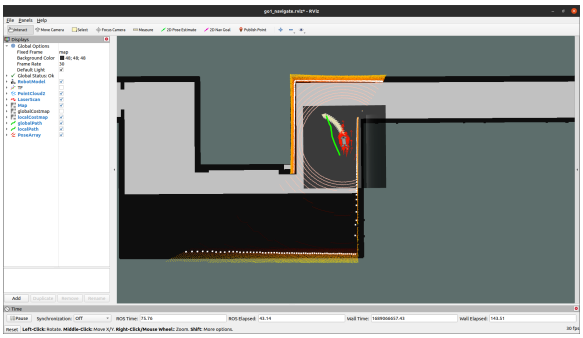


Figure 5.16.: Short goal

As discussed in Chapter 4 , the planners and costmaps were implemented after setting up the lidar scans and AMCL node. To test autonomous navigation, the robot was given target goals via the *2D Nav Goal* option in Rviz. Figure 5.16 shows the Go1 robot on the 2D map surrounded by AMCL pose array, the local costmap in black, the Ouster 3D point cloud, and 2D laser scans. The robot was given a short goal just behind its back.

This results in two plans: a global plan in green and a local path in yellow. It was visually observed that both plans were accurate. The local plan tries to move the robot toward the global plan and align as much as possible. When the endpoint in the global map is reached, the robot stops moving. This was the expected result for simulating 2D autonomous navigation.

There were also some issues with 2D autonomous navigation in the simulation environment. Figure 5.17 displays one such event. When the robot is given a long-distance goal, it tends to lose its laser scan-matching alignment in long corridors. This can be caused due to poor performance of AMCL in long corridors. AMCL needs edge features in the 2D scan to align itself over time. Straight long corridors mean no edge features are available, and this causes a drift in the alignment over time. This issue should be further investigated to avoid colliding with walls.

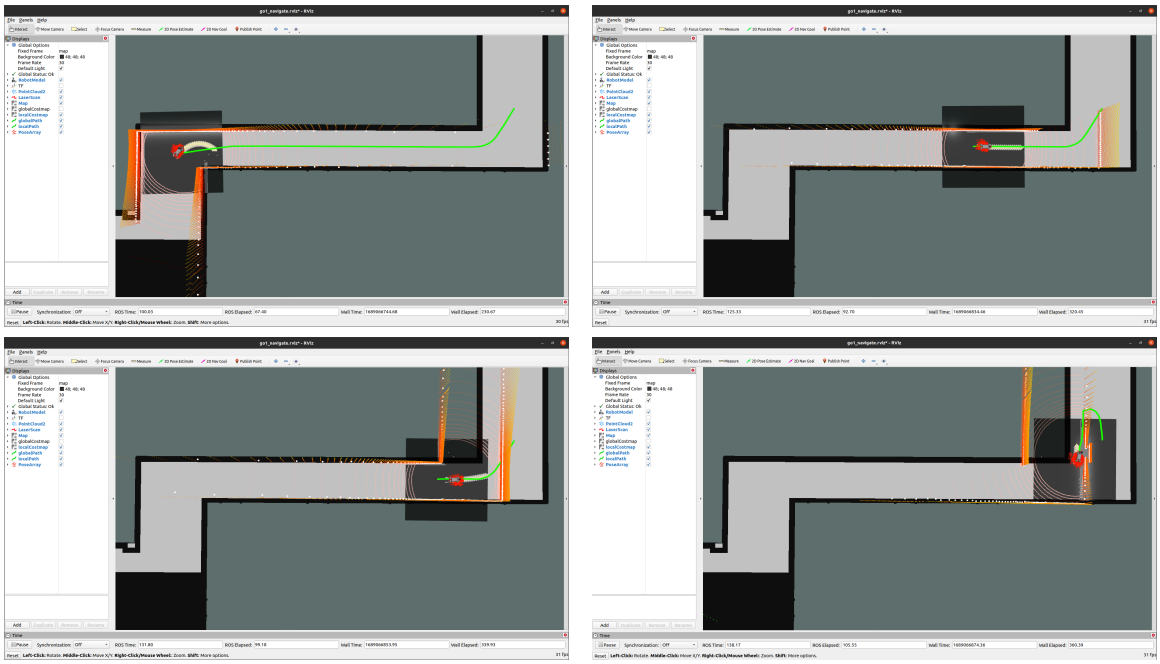


Figure 5.17.: Long goal

5.3. Real World Experiments

This section discusses results from the autonomous navigation setup for the Go1 robot in real world scenarios. A BIM model of the real world environment was used to generate a map used in the ROS navigation stack. It is expected that the Go1 robot can navigate autonomously in this environment based on input target goals.

Figure 5.18.: Real world environment for Autonomous Navigation test

5.3.1. Basic configuration

Similar to the simulation environment experiments, the first step is to evaluate if all the transforms are correctly set up between the frames. It should then be verified if the sensor data reflects the real world. Errors in the coordinate transformation of sensor data can lead to disastrous consequences because of collisions or breakdowns.

As seen in Figure 5.19 , all the joints and links of the Go1 robot are correctly set up, and data is being published as expected on the *trunk* frame. The *trunk* frame, in turn, forwards the data to the *base* frame of the robot. As defined in the modified URDF file for the Go1 robot, the lidar and camera sensors transform the data to the parent *mounting_frame*, transforming it to the robot's *trunk* frame. All these transforms are published by the *robot_state_publisher*. The *base* to *odom* frame transform is provided by the Extended Kalman Filter (EKF) based robot localization node. The inputs to the EKF are the Go1 robot's Pose (x, y, z) and Euler angles (Roll, Pitch, Yaw) which are generated by the Go1 robot in *HighState* [52]. The input BIM-based 2D map is published on the *map* frame, and as

discussed in Chapter 4 , the AMCL algorithm provides the *map* to *odom* frame transform.

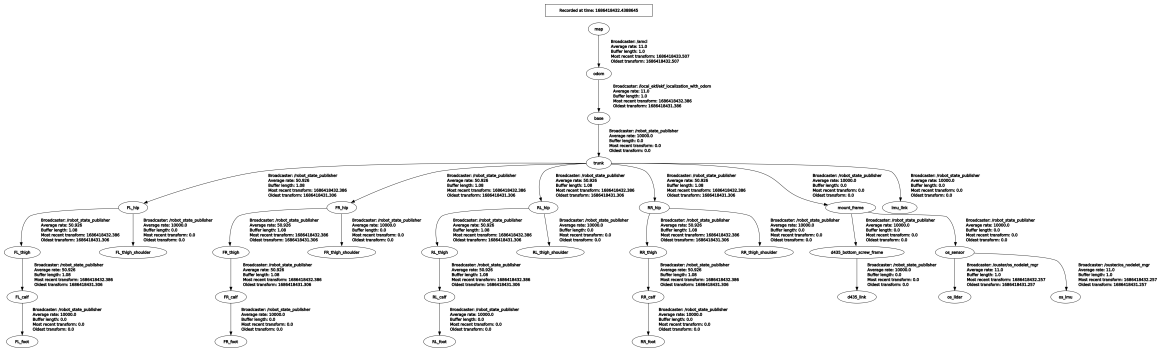
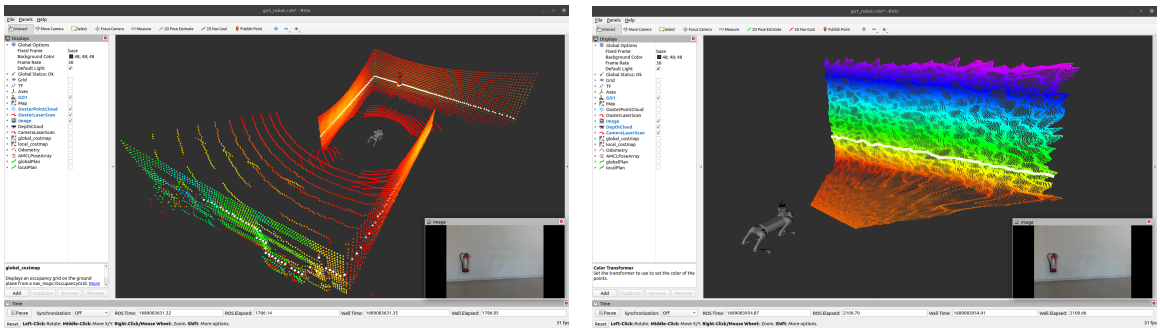


Figure 5.19.: Transform tree of Go1 robot

5.3.2. Sensor data

After verifying the transforms, the incoming data was visualized using the Rviz tool. With Rviz, the 3D point cloud option can be used to visualize the incoming data on the lidar and camera topics, i.e., */ouster/points* and */camera/depth/image_rect_raw* and verify the alignment of the sensors. If the configuration files contain incorrect geometry information, the sensor point cloud data will have misalignments with the real world. Before visualizing the data, the lidar and camera sensor drivers for ROS should be started using their launch files,i.e., *ouster_ros/sensor.launch* and *realsense2_camera/rs_camera.launch*.



(a) Ouster lidar (b) Realsense camera

Figure 5.20.: Sensor data

The ROS navigation stack requires the sensor data to be provided as 2D scans. Therefore the *pointcloud_to_laserscan.launch* file was used to convert the 3D point cloud from Ouster lidar into 2D laser scans. Figure 5.20a shows the ouster pointcloud in color on the *z-axis* and the 2D laser scan in white points. During the initial tests, it was observed that the

pointcloud_to_laserscan node provides the scan which is not exactly on the horizontal plane but rather with a 5-degree tilt due to the mounting position of the Ouster sensor on the Go1 robot. In a long corridor, the floor is detected as a wall behind the robot and identified as an obstacle. This, in turn, causes problems for scan matching algorithms like AMCL and results in poor tracking. Therefore, the URDF file for the Ouster lidar sensor was modified by removing the 5-degree tilt from the *z-axis*, which means the ROS framework will think that the lidar is mounted parallel to the floor and has a 0-degree tilt on the *z-axis*. Further, the *pointcloud_to_laserscan* node's parameters were updated by repeatedly changing them and observing the output laser scan in Rviz. Finally, the *pointcloud_to_laserscan* node outputs a 2D scan which detects the walls as shown in Figure 5.20a .

Realsense camera also provides a 3D image and depth point cloud, which are much easier to visually verify as they show the environment exactly as it is and have a fast update rate. Besides providing visual images for observing the robot's path and tracking objects, it can also be used to produce a 2D laser scan similar to the *pointcloud_to_laserscan* node. As discussed in Chapter 4 , this is done using the *depthcamera_to_laserscan.launch* file. Figure 5.20b shows the camera depth point cloud in color on the *z-axis* and 2D laser scan in white points. It also shows an image of the environment from the camera *image* topic. Compared with the 2D laser scan from lidar, the 2D scan from the camera sensor fluctuates more on the *z-axis*. However, it is still useful as a redundant data source in case the lidar sensor runs out of power. Both *pointcloud_to_laserscan* and *depthcamera_to_laserscan* nodes were used as inputs to AMCL algorithm for experiments in this research work.

5.3.3. 2D Localization

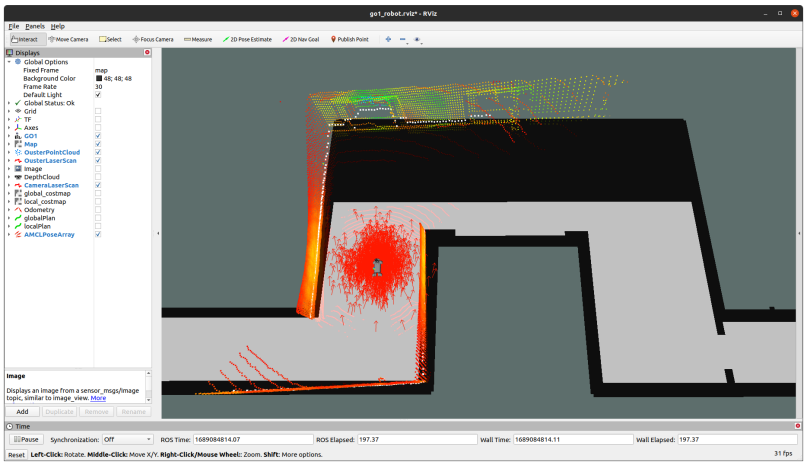


Figure 5.21.: AMCL localization

After verifying the input sensor data, the localization node should be verified as it provides the important transformation between the fixed *map* frame which represents the en-

vironment around the robot, and the *odom* frame that provides the odometry information of the robot. The *map* to *odom* frame transform provides information of the movement of a robot from its inital position to a target position. The AMCL node created for simulation was used for real world navigation. The AMCL node works for both lidar and camera-based 2D laser scans. It was observed that the AMCL node works well when the robot's initial pose is set on the map using Rviz. The AMCL pose array produces good results if the robot walks straight, but as soon as the robot takes a turn, the AMCL node corrupts and produces junk data. This leads to bad localization on the map, affecting path planning.

Figure 5.21 shows the lidar sensor data and AMCL's pose array when the Go1 robot's initial pose is set through Rviz. The correctness of the robot in the map can be verified by observing the 2D and 3D lidar scans matching the walls on the *PGM* map.

5.3.4. Autonomous Navigation

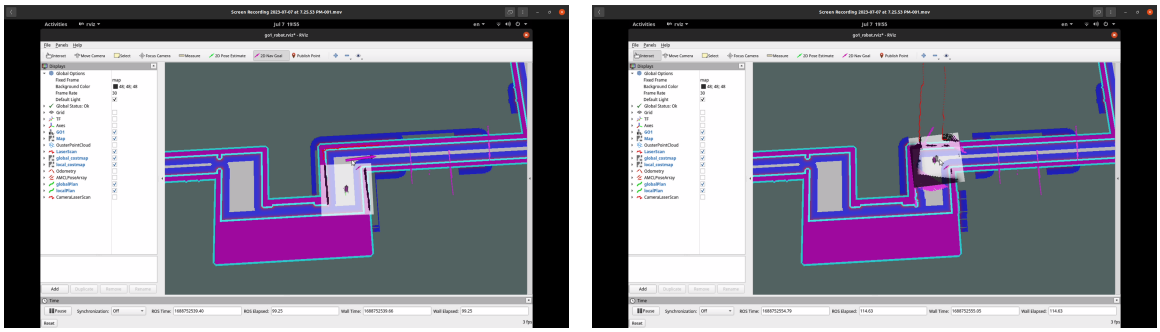


Figure 5.22.: Autonomous navigation

The costmaps and planers developed for simulation were reused for real world navigation. The Global planner was set up using Dijkstra's algorithm, and the local planner was set up using the DWA planner. It was expected that the robot would be able to move to target goals autonomously for both long-distance and short-distance targets. The entire navigation stack was visualized on Rviz after running the prerequisite launch files for the Go1 robot and perception sensors. The costmaps were built as expected, and the AMCL was able to localize the robot based on the initial pose. It was observed that the robot could autonomously navigate to short goals in straight lines, but it struggled to navigate to long-distance goals. As soon as the robot rotates following the planner's turn command, the AMCL pose array produces junk data, and the laser scans mismatch with the real world. Figure 5.22 shows an example of such an event.

As seen in Figure 5.22 , the Go1 robot is given a target goal at a corner in the corridor using the *2D Nav Goal* option in Rviz. Though the planners produce accurate global and local paths, the Go1 robot's turning maneuver causes the input sensor scans to mismatch and produce incorrect local costmaps. This leads to incorrect recognition of the real world

environment; thus, the robot starts performing random incorrect maneuvers. To fix the issue, the depth cloud from the camera was used to produce 2D scans, but it did not improve the autonomous navigation performance. One of the causes for this error can be incorrect odometry generation leading to erroneous transform of *odom* to *base* frame. This issue can be resolved using the raw IMU data from the Go1 robot to produce odometry or adding a high-quality IMU sensor to the payload system mounted on the Go1 robot. This will be further discussed in the upcoming Chapter 6 .

5.3.5. 3D Localization

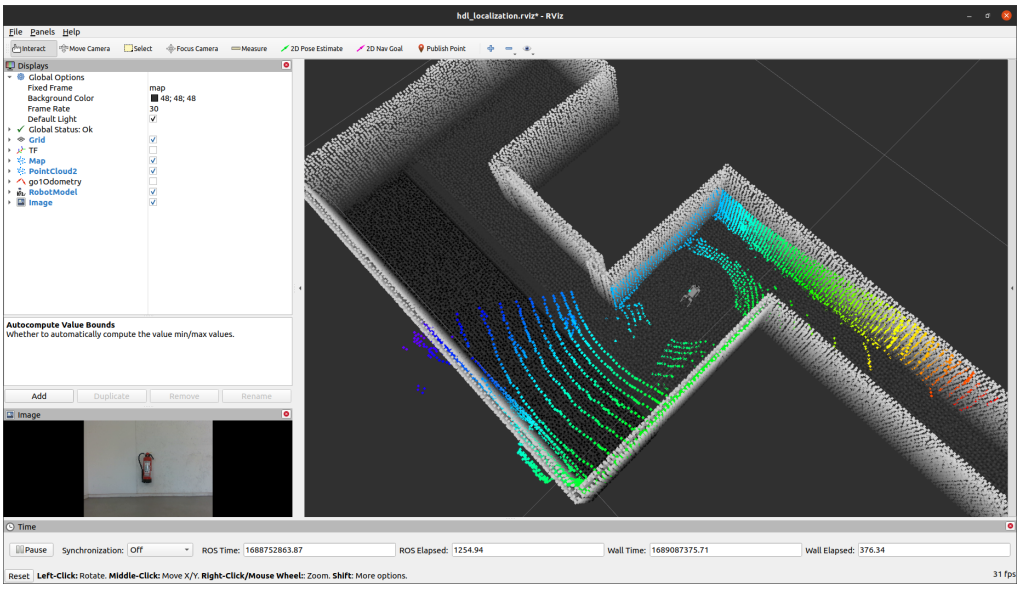


Figure 5.23.: HDL Graph SLAM based localization

Since 2D localization with AMCL had poor results, multiple 3D localization algorithms were explored. Most algorithms struggled to produce good results at startup without accurate input IMU data. However, the *hdl_localization* algorithm from *hdl_graph_slam* works well even in the absence of IMU data. It requires an input 3D map as a *.pcd* format file, which represents the environment as a point cloud. As discussed in Section 4.2.3 , the point cloud map was extracted from the input BIM model and used as an input to the *hdl_localization* node. This node also requires the initial pose of the Go1 robot in the 3D environment. It can be set in the input 3D map by using the Rviz tool. Initially, the *hdl_localization* node will take a few seconds to match the point cloud but once the match is established it provides good tracking of the robot in the 3D environment.

Figure 5.23 displays the BIM based 3D pointcloud map in white color and the generated aligned points from *hdl_localization* node in color. It is observed that the pointcloud matching has high accuracy.

6. Conclusion and Future scope

The core aim of this research work was to set up a basic framework for the autonomous navigation of a quadruped robot in simulation environments and real world environments while utilizing the information about these environments from their corresponding BIM models. This chapter discussed the findings and future scope of extending this work.

6.1. Conclusions

In this section, the objectives set in Section 1.2 will be addressed to draw conclusions.

1. **Create the basic setup to integrate a quadruped robot in ROS**

   This research work used the Go1 quadruped robot from Unitree and the requisite *xacro* and *URDF* configuration files imported from Unitree's open source project. In addition to the quadruped robot, an Ouster lidar, and a Realsense camera were used for autonomous navigation. Thus, their information was defined in ROS compliant configuration files and added to Unitree's configuration files. 3D meshes were created for the payload mounting system on the Go1 robot, and transforms were set up to provide accurate real world data.

2. **Setup a basic simulation framework in ROS**

   A basic simulation framework was created for simulating the Unitree Go1 robot in the Gazebo simulator. A process was defined to create a 3D gazebo *worlds* from 3D BIM models of a facility. The 3D models were first converted into a wavefront (*.obj*) 3D mesh format file using the open source tool IfcConvert. These meshes were then added to *SDF* files required to create a world file in Gazebo. An open source ROS control-based project was used to simulate the movement of the quadruped Go1 robot in the virtual gazebo environment. Further, all the nodes necessary for the ROS navigation stack were created and configured to provide accurate results. The simulation framework was majorly tested in one custom-made 3D environment but can be easily extended to other 3D environment models. Multiple ROS based SLAM packages were explored to verify if the simulation framework can simulate autonomous navigation properly in Gazebo simulator.

3. **Extract a map from BIM**

   A Python based algorithm was developed to generate a 2D occupancy grid map from a 3D BIM model of a facility. The map was constructed from three distinct layers,

each containing vital information about the environment, i.e., information about the outdoor environment, indoor environment, and collision objects. The focus was on processing one BIM model at a time. The algorithm can be automated for processing multiple models in a batch but needs user input for specifying the "z-coordinate" to slice the three layers from each model. In terms of computational time, the whole map generation process is quite fast, thus enabling a quick trial-and-error method to check the quality of the output visually. Multiple ROS based map creation methods were also explored to compare the performance of this algorithm [75]. The BIM model was also utilized to create 3D maps based on point clouds or other efficient voxel-based approaches. This 3D map was used to explore 3D localization methods in ROS.

4. **Simulation to real world transition**

   The simulation framework was used to understand the capabilities of the quadruped robot and to set up the prerequisite packages required for the ROS navigation stack in the real world. A basic autonomous navigation framework was created and tested in the real world. Before operating the Go1 robot in the real world, the environment was simulated in Gazebo, and comparisons were made on how accurately the simulation and real world match. It was observed that the robot could not navigate with full autonomy in the real world due to localization issues on the 2D map. Some potential solutions to resolve this were explored, but it was concluded that this would not be possible in a short time frame. Another conclusion was that the full capabilities of the 3D Ouster lidar should be utilized for localization as the 3D point cloud to 2D laser scan conversion operation artificially reduces the input sensor data. Multiple ROS based 3D localization algorithms were also explored to understand the performance of localization in an input 3D BIM-based point cloud map.

6.2. Limitations

In this section, the limitations of this research work are discussed.

- In the 3D BIM model to 2D map generation process, it is difficult to visually identify the accuracy of the final map. In some testing scenarios, it was observed that the final map might have a different resolution than the actual environment. During the map generation process, the coordinates of the origin point of the 3D BIM model are preserved in the generated SVG files. These coordinates were configured using a *YAML* file for the generated occupancy grid map. But it was found that these coordinates were not correct, and the resolution issue persists. A potential solution is to use different tools for generating the SVG files and then converting them to PNG files because there might be a loss of information in this process.

- Another limitation of the map generation process is that a batch job of multiple input BIM files is not possible. To utilize the Python script for multiple BIM models, it is recommended to extract the 2D layer height from user input through the command line interface.
- The AMCL node for localization provides poor performance in certain areas that do not contain unique features like long corridors. Even if walls have unique features like windows, the performance is still dependent on the height at which the 2D laser scan is extracted from the 3D point cloud. The AMCL node is generally designed for differential drive-based robots and should be tuned for better performance in quadruped robots.
- Certain 3D localization algorithms that were explored during this research work failed to provide good performance at startup due to a lack of accurate odometry information. Further investigation is required to answer if the performance of these algorithms can be improved. And the more important question would be: Are these algorithms required to achieve the end goal of autonomous navigation because other options exist such as HDL Localization that does not depend on accurate odometry information.
- During the development of the ROS navigation stack for real world autonomous navigation, it was observed that the Go1 hardware did not respond to velocity commands from the ROS framework. An outdated firmware version was identified as the root cause, so a new firmware was installed on the Go1 robot; however, that made the Go1 robot totally unresponsive, even to the remote controller's commands. This issue was later fixed by replacing the main board of the Go1 robot, and it began responding to ROS velocity commands. In the future, if more control is required on the individual hardware joints and internal camera sensors of the Go1 robot, then it is recommended that custom nodes should be designed to extract the raw data from these sensors using the onboard Raspberry Pis.
- The mobile-device based *hotspot* network connection has limitations of range and is prone to poor connectivity in indoor areas. Another wireless network connection method should be explored.

6.3. Discussion

The primary contribution of this work is the quadruped robot based framework for autonomously operating the Go1 robot in various indoor environments based on a 2D map generated from a 3D BIM model. The map creation method results in a good initial map that can be used for localization at startup. If needed, the map can be redrawn and updated over time based on the difference between the real world and its BIM model representation. It is difficult to visually estimate the accuracy of the generated map, especially for

multi-story BIM models. The obtained map might be prone to resolution or projection errors. During the simulation experiment, two methods were used to verify the quality of the map. The first method was to visually check the alignment of detected edge features like walls, corners, or windows in the gazebo lidar scan with edge features in the 2D map from the BIM model. The second method focused on generating a 2D sliced map from a gazebo model by using the gazebo collision detection plugins.

The entire framework was designed in *ROS Noetic* but can be adapted to the latest releases of *ROS2* as well. The custom gazebo world creation process is rather straightforward and requires only a 3D mesh file. The xacro configuration files for the Go1 robot are customizable, and additional sensors or payload systems can be easily added. In this research work, an Ouster lidar and a Realsense camera sensor were simulated through their gazebo plugins. While exploring the potential computational resources that would be required for a multi-camera sensor system, it was observed that simulating four Realsense cameras together with an Ouster lidar slowed down the entire simulation significantly. To work with the 2D navigation stack, a 2D laser scan was extracted from a 3D point cloud using *pointcloud-to-laserscan* node in ROS.

The simulation framework was then used to create the ROS navigation stack for real world autonomous navigation. Though both frameworks used a similar implementation, autonomous navigation did not work as expected, and the robot faced issues in navigating to target goals. This project can be used as a starting point to tune the navigation stack for real world development. The simulation framework is customizable and allows to perform diverse experiments in a shorter time compared to direct real world development. The simulation framework also supports multiple gaits for Unitree robots, allowing them to climb stairs. The ROS navigation tuning guide [77] can be used as an initial verification method to identify the root causes of the failure and to fix them. One of the options to solve the localization issue is by using 3D maps to localize the robot and provide the pose to the 2D map. However, this requires a static transform should be established from the the 3D map to the corresponding 2D map of the floor on which the robot moves. This can be an area of further research.

6.4. Future scope

This section describes some of the future research topics based on this thesis.

- The *SVG* file creation process from the BIM model can be tried with other *SVG* file processing tools. The origin coordinates information from the BIM model should be utilized in the final 2D map to avoid the manual setting of resolution and coordinates.

- The simulation framework designed in this work can be extended to work in different kinds of environments by adding static and dynamic objects to the Gazebo environment. It can also be extended to work in large-scale environments. The sim-

ulation framework can be customized to work for ROS2[34] and Nav2[40], which is the successor of the ROS navigation stack.

- Instead of using odometry messages from the Go1 robot, raw IMU sensor data can be extracted from the Go1 robot[19]. If the quality of the IMU sensor data is not satisfactory, then a high-quality IMU sensor should be added to the payload system mounted on the Go1 robot to generate high-quality odometry.
- Additional camera sensors can be added to visualize a 360-degree view around the Go1 robot. This will lead to an increase in computational resource requirements. In that case, a GPU should also be added.
- It was observed that the Ouster sensor heats up quickly in the existing mounting frame. For long-term usage, heat sinks should be provided for the Ouster lidar. Additionally, it should be mounted parallel to the ground if 2D navigation is the main goal. For 3D navigation, this setup might work with an inclined lidar; however, it is recommended to place it parallelly.
- The ouster lidar continuously provides high-quality point cloud data of the environment. This data can be captured through another node and saved to compare with the input BIM model.

Appendix

A. Hardware And Technical Setup

A.1. Tools

The hardware setup used in this research work consists of the following:

- Unitree Go1: A quadruped robot from Unitree robotics
- OS1-32: A 3D lidar sensor from Ouster Inc.
- RealSense D435i: A depth camera from Intel Realsense
- Mini PC: A portable size computer running Ubuntu 20.04 and ROS Noetic
- Payload: A 3D printed lightweight mounting system on Go1 robot for perception sensors and the computer
- Power supply: 2 rechargeable batteries to run mini pc and Ouster lidar

The computer is connected to the Go1 robot and Ouster lidar using ethernet cables, while the Realsense camera sensor can be connected using a *USB Type-C to Type-C* cable. A pack of rechargeable batteries is used to supply power to the Ouster lidar and the computer. The camera sensor is powered by the mini computer through the cable connection itself and does not need any other power souce. The Unitree Go1 sensor also utilizes a rechargeable battery system for power supply.

A.2. ROS launch files

A.2.1. Launch Custom World

```
<launch>
    <arg name="robot_type" default="$(env ROBOT_TYPE)" doc="Robot type: [a1, aliengo, go1, laikago]"/>
    <arg name="rviz"       default="true"/>

    <rosparam file="$(find legged_gazebo)/config/default.yaml" command="load"/>

    <param name="legged_robot_description" command="$(find xacro)/xacro $(find legged_unitree_description)/urdf/robot.xacro
       robot_type:=$(arg robot_type)
    "/>

    <!-- We resume the logic in empty_world.launch, changing only the name of the world to be launched -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find legged_gazebo)/worlds/hall.world"/>
    </include>

    <!-- push robot_description to factory and spawn robot in gazebo -->
    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" clear_params="true"
          args="-z 1.5 -param legged_robot_description -urdf -model $(arg robot_type)" output="screen"/>

    <!-- Launch rviz -->
    <node if="$(arg rviz)" name="rviz" pkg="rviz" type="rviz"
        args="-d $(find legged_unitree_description)/rviz/go1_view.rviz"
        output="screen"/>
</launch>
```

A.2.2. Launch Pointcloud to Laserscan Node

```
<launch>
    <node pkg="pointcloud_to_laserscan" type="pointcloud_to_laserscan_node"
    ↪  name="pointcloud_to_laserscan">

      <remap from="cloud_in" to="/ouster/points"/>
      <remap from="scan" to="/scan" />
      <rosparam>
        target_frame: os_sensor
        transform_tolerance: 0.01

        angle_min: -3.14  <!-- -PI, minimum scan angle in radians -->
        angle_max: 3.14   <!-- +PI, maximum scan angle in radians -->

        range_min: 0.3
        range_max: 70.0
      </rosparam>
    </node>
</launch>
```

A.2.3. Launch AMCL node

```
<launch>
    <arg name="frame_prefix" default=""/>
    <arg name="scan_topic" default="/scan" />

    <node pkg="amcl" type="amcl" name="amcl" output="screen">
      <remap from="scan" to="$(arg scan_topic)"/>

      <param name="use_map_topic" value="false"/>
      <param name="base_frame_id" value="$(arg frame_prefix)base"/>
      <param name="odom_frame_id" value="$(arg frame_prefix)odom"/>
      <param name="global_frame_id" value="$(arg frame_prefix)map"/>

      <param name="odom_model_type" value="omni"/>
      <param name="odom_alpha5" value="0.1"/>
      <param name="gui_publish_rate" value="100.0"/>
      <param name="laser_max_beams" value="1000"/>
      <param name="laser_max_range" value="70.0"/>
      <param name="min_particles" value="500"/>
      <param name="max_particles" value="20000"/>
      <param name="kld_err" value="0.05"/>
      <param name="kld_z" value="0.99"/>
      <param name="odom_alpha1" value="0.2"/>
      <param name="odom_alpha2" value="0.2"/>

      <param name="odom_alpha3" value="0.2"/>
      <param name="odom_alpha4" value="0.2"/>
      <param name="laser_z_hit" value="0.5"/>
      <param name="laser_z_short" value="0.05"/>
      <param name="laser_z_max" value="0.05"/>
      <param name="laser_z_rand" value="0.5"/>
      <param name="laser_sigma_hit" value="0.2"/>
      <param name="laser_lambda_short" value="0.1"/>
      <param name="laser_model_type" value="likelihood_field"/>
      <param name="laser_likelihood_max_dist" value="2.0"/>
      <param name="update_min_d" value="0.25"/>
      <param name="update_min_a" value="0.2"/>
      <param name="resample_interval" value="1"/>

      <param name="transform_tolerance" value="1.0"/>
      <param name="recovery_alpha_slow" value="0.0"/>
      <param name="recovery_alpha_fast" value="0.0"/>

      <!-- <param name="initial_pose_a" value="0.0"/> -->
    </node>
</launch>
```

A.2.4. Launch Autonomous Navigation

```
<launch>
  <arg name="rviz" default="true"/>

  <!-- Map server -->
  <arg name="map_file"
  default="$(find legged_unitree_description)/maps/map.yaml"/>
  <node pkg="map_server" name="map_server" type="map_server"
        args="$(arg map_file)">
  <param name="frame_id" value="/map" />
  </node>

  <!-- Convert 3D point cloud to 2D laser -->
  <include file="$(find legged_unitree_description)
  /launch/include/pointcloud_to_laserscan.launch"/>

  <!-- AMCL used for localization -->
  <include
  file="$(find legged_unitree_description)/launch/include/amcl.launch"/>

  <!-- Calls navigation stack -->
  <include
  file="$(find
↪  legged_unitree_description)/launch/include/move_base.launch"/>

  <node if="$(arg rviz)" name="rviz" pkg="rviz" type="rviz"
  args="-d $(find legged_unitree_description)/rviz/go1_navigate.rviz -f
↪  /map"
  output="screen"/>
</launch>
```

Bibliography

[1] Debaditya Acharya, Kourosh Khoshelham, and Stephan Winter. Bim-posenet: Indoor camera localisation using a 3d indoor model and deep learning from synthetic images. *ISPRS Journal of Photogrammetry and Remote Sensing*, 150:245–258, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0924271619300589, doi:https://doi.org/10.1016/j.isprsjprs.2019.02.020.

[2] Gerardo Bledt, Matthew J Powell, Benjamin Katz, Jared Di Carlo, Patrick M Wensing, and Sangbae Kim. Mit cheetah 3: Design and control of a robust, dynamic quadruped robot. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2245–2252. IEEE, 2018.

[3] F. J. Bot, Pirouz Nourian, and Edward Verbree. A graph-matching approach to indoor localization using a mobile device and a reference bim. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2019.

[4] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[5] Fernando Caballero and Luis Merino. Dll: Direct lidar localization. a map-based localization approach for aerial robots. In *2021 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 5491–5498. IEEE, 2021.

[6] Do Jeong Chan and Lee Wang Heon. Ros based remote control of an autonomous mobile robot and visual slam for parking lot management. *Journal of Institute of Control Robotics and Systems*, 24:1088–1093, 2018.

[7] Junjie Chen, Shuai Li, and Weisheng Lu. Align to locate: Registering photogrammetric point clouds to bim for robust indoor localization. *Building and Environment*, 209:108675, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0360132321010659, doi:https://doi.org/10.1016/j.buildenv.2021.108675.

[8] Sachin Chitta, Eitan Marder-Eppstein, Wim Meeussen, Vijay Pradeep, Adolfo Rodríguez Tsouroukdissian, Jonathan Bohren, David Coleman, Bence Magyar, Gennaro Raiola, Mathias Lüdtke, and Enrique Fernández Perdomo. ros_control: A generic and simple control framework for ros. *The Journal of Open Source Software*, 2017. URL: http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf, doi:10.21105/joss.00456.

[9] CloudCompare. 3d point cloud and mesh processing software. [Software]. URL: http://www.cloudcompare.org/.

[10] IfcOpenShell Contributors. Ifcconvert: An application for converting ifc geometry into several file formats. [Software]. URL: https://ifcopenshell.sourceforge.net/ifcconvert.html.

[11] IfcOpenShell Contributors. Ifcconvert documentation. [Online]. URL: https://blenderbim.org/docs-python/ifcconvert/usage.html.

[12] IfcOpenShell Contributors. Ifcopenshell: The open source ifc toolkit and geometry engine. [Online]. URL: https://ifcopenshell.org/.

[13] Francisco A. X. Da Mota, Matheus Xavier Rocha, Joel J. P. C. Rodrigues, Victor Hugo C. De Albuquerque, and Auzuir Ripardo De Alexandria. Localization and navigation for autonomous mobile robots using petri nets in indoor environments. *IEEE Access*, 6:31665–31676, 2018. doi:10.1109/ACCESS.2018.2846554.

[14] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs:(numerische mathematik, 1 (1959), p 269-271). 1959.

[15] Péter Fankhauser, C. Dario Bellicoso, Christian Gehring, Renaud Dubé, Abel Gawel, and Marco Hutter. Free Gait – An Architecture for the Versatile Control of Legged Robots. In *IEEE-RAS International Conference on Humanoid Robots*, 2016.

[16] Farbod Farshidian et al. OCS2: An open source library for optimal control of switched systems. [Online]. URL: https://github.com/leggedrobotics/ocs2.

[17] Yanxiao Feng, Julian Wang, H. Howard Fan, and Ce Gao. Bimil: Automatic generation of bim-based indoor localization user interface for emergency response. In *Interacción*, 2020.

[18] LAAS-CNRS Gepetto team. Package for the simulation of the ouster os1-64 with ros and gazebo. [Online]. URL: https://gepgitlab.laas.fr/gepetto/ouster-gazebo-simulation.

[19] AATB GmbH. Publish camera/imu/odometry as ros topics on the unitree go1 dogs. [Online]. URL: https://github.com/aatb-ch/go1_republisher.

[20] Unitree Go1. A quadruped robot from unitreerobotics. [Online]. URL: https://shop.unitree.com/products/unitreeyushutechnologydog-artificial-intelligence-companion-bionic-companion-intelligent-robot-go1-quadruped-robot-dog.

[21] Muhammad Gopee, Samuel Prieto, and Borja García de Soto. Ifc-based generation of semantic obstacle maps for autonomous robotic systems. 07 2022. doi:10.35490/EC3.2022.161.

[22] HKUST Aerial Robotics Group. Advanced implementation of loam. [Online]. URL: https://github.com/HKUST-Aerial-Robotics/A-LOAM.

[23] In Young Ha, Hongjo Kim, Somin Park, and Hyoungkwan Kim. Image-based indoor localization using bim and features of cnn. *Proceedings of the 35th International Symposium on Automation and Robotics in Construction (ISARC)*, 2018.

[24] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[25] Patricia M. Herbers and Markus König. Indoor localization for augmented reality devices using bim, point clouds, and template matching. *Applied Sciences*, 2019.

[26] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-time loop closure in 2d lidar slam. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1271–1278, 2016.

[27] Armin Hornung, Kai M Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous robots*, 34:189–206, 2013.

[28] Marco Hutter, Christian Gehring, Dominic Jud, Andreas Lauber, C Dario Bellicoso, Vassilios Tsounis, Jemin Hwangbo, Karen Bodie, Peter Fankhauser, Michael Bloesch, et al. Anymal-a highly mobile and dynamic quadrupedal robot. In *2016 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 38–44. IEEE, 2016.

[29] Ouster Inc. Introducing the os1-32, the lowest cost 32-channel sensor ever. [Online]. URL: https://ouster.com/blog/os1-32-high-resolution-low-cost-lidar-sensor/.

[30] Ouster Inc. Official ros drivers for ouster sensors. [Online]. URL: https://github.com/ouster-lidar/ouster-ros.

[31] ISO Central Secretary. *Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries*. International Organization for Standardization, Geneva, CH, ISO 16739-1:2018 edition, 2018. URL: https://www.iso.org/standard/70303.html.

[32] Jaehoon Jung, Sanghyun Yoon, Stachniss Cyrill, and Joon Heo. A study on 3d indoor mapping for as-built bim creation by using graph-based slam. *Korean Journal of Construction Engineering and Management*, 17:32–42, 2016.

[33] Benjamin Katz, Jared Di Carlo, and Sangbae Kim. Mini cheetah: A platform for pushing the limits of dynamic quadruped control. In *2019 international conference on robotics and automation (ICRA)*, pages 6295–6301. IEEE, 2019.

[34] Soo Young Kim. Unitree go1 ros 2 api and examples. [Online]. URL: https://github.com/kimsooyoung/go1_ros2.

[35] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE, November 2011.

[36] Kenji Koide. koide3/hdl_localization, real-time 3d localization using a (velodyne) 3d lidar. URL: https://github.com/koide3/hdl_localization.

[37] Kenji Koide, Jun Miura, and Emanuele Menegatti. A portable three-dimensional lidar-based system for long-term and wide-area people behavior measurement. *International Journal of Advanced Robotic Systems*, 16, 02 2019. doi:10.1177/1729881419841532.

[38] Mathieu Labbé and François Michaud. Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 36(2):416–446, 2019.

[39] Qiayuan Liao. Nonlinear mpc and wbc framework for legged robot based on ocs2 and ros-controls, legged_control. [Online]. URL: https://github.com/qiayuanliao/legged_control.

[40] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2718–2725. IEEE, 2020.

[41] Bilawal Mahmood, SangUk Han, and Dong-Eun Lee. Bim-based registration and localization of 3d point clouds of indoor scenes using geometric features for augmented reality. *Remote. Sens.*, 12:2302, 2020.

[42] Eitan Marder-Eppstein, Eric Berger, Tully Foote, Brian Gerkey, and Kurt Konolige. The office marathon: Robust navigation in an indoor office environment. In *2010 IEEE International Conference on Robotics and Automation*, pages 300–307, 2010. doi:10.1109/ROBOT.2010.5509725.

[43] Thomas Moore and Daniel Stouch. A generalized extended kalman filter implementation for the robot operating system. In *Intelligent Autonomous Systems 13: Proceedings of the 13th International Conference IAS-13*, pages 335–348. Springer, 2016.

[44] MYBOTSHOP. Mybotshop/qre_go1, quadruped robot go1 ros wrapper. URL: https://github.com/MYBOTSHOP/qre_go1.

[45] Michael Neunert, Markus Stäuble, Markus Giftthaler, Carmine D Bellicoso, Jan Carius, Christian Gehring, Marco Hutter, and Jonas Buchli. Whole-body nonlinear model predictive control through contacts for quadrupeds. *IEEE Robotics and Automation Letters*, 3(3):1458–1465, 2018.

[46] Open Design Alliance (ODA). Open ifc viewer: A professional-grade viewer for ifc files. [Software]. URL: https://openifcviewer.com/documentation#about-open-ifc-viewer.

[47] Aritra Pal, Jacob J. Lin, and Shang-Hsien Hsieh. *A Framework for Automated Daily Construction Progress Monitoring Leveraging Unordered Site Photographs*, pages 538–545. URL: https://ascelibrary.org/doi/abs/10.1061/9780784483893.067, arXiv:https://ascelibrary.org/doi/pdf/10.1061/9780784483893.067, doi:10.1061/9780784483893.067.

[48] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[49] Gennaro Raiola, Michele Focchi, and Enrico Mingo Hoffman. Wolf: the whole-body locomotion framework for quadruped robots. *arXiv preprint arXiv:2205.06526*, 2022.

[50] Intel® RealSense™. Intel(r) realsense(tm) ros wrapper for depth camera. [Online]. URL: https://github.com/IntelRealSense/realsense-ros.

[51] Intel® RealSense™. Intel® realsense™ depth camera d435i combines the robust depth sensing capabilities of the d435 with the addition of an inertial measurement unit (imu). [Online]. URL: https://www.intelrealsense.com/depth-camera-d435i/.

[52] Unitree Robotics. Ros packages for unitree robots. [Online]. URL: https://github.com/unitreerobotics/unitree_ros/blob/master/robots/go1_description/.

[53] Unitree Robotics. Unitree docs. [Online]. URL: https://unitree-docs.readthedocs.io/en/latest/get_started/Go1_Edu.html.

[54] ROS.org. Amcl: Probabilistic localization algorithm for a robot moving in 2d. [Online]. URL: http://wiki.ros.org/amcl.

[55] ROS.org. Converts a 3d point cloud into a 2d laser scan. [Online]. URL: http://wiki.ros.org/pointcloud_to_laserscan.

[56] ROS.org. Converts a depth image to a laser scan for use with navigation and localization. [Online]. URL: http://wiki.ros.org/depthimage_to_laserscan.

[57] ROS.org. Documentation for sensor_msgs. [Online]. URL: http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/PointCloud2.html.

[58] ROS.org. An implementation of the dynamic window approach to local robot navigation on a plane. [Online]. URL: http://wiki.ros.org/dwa_local_planner.

[59] ROS.org. Implementations of the trajectory rollout and dynamic window approaches to local robot navigation on a plane. [Online]. URL: http://wiki.ros.org/base_local_planner.

[60] ROS.org. An optimal trajectory planner considering distinctive topologies for mobile robots based on timed-elastic-bands. [Online]. URL: http://wiki.ros.org/teb_local_planner.

[61] ROS.org. Ros wrapper for openslam's gmapping. [Online]. URL: http://wiki.ros.org/gmapping.

[62] Wil Selby. Ouster sample code. [Online]. URL: https://github.com/wilselby/ouster_example.

[63] Wil Selby. Simulating an ouster os-1 lidar sensor in ros gazebo and rviz, May 2019. URL: https://wilselby.com/2019/05/simulating-an-ouster-os-1-lidar-sensor-in-ros-gazebo-and-rviz/.

[64] Tixiao Shan and Brendan Englot. Lego-loam: Lightweight and ground-optimized lidar odometry and mapping on variable terrain. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4758–4765, 2018. doi:10.1109/IROS.2018.8594299.

[65] Hao Shen, Xiang Li, Xin Jiang, and Yunhui Liu. Automatic scan planning and construction progress monitoring in unknown building scene. *2021 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 1617–1622, 2021.

[66] PAL Robotics S.L. Pal-robotics/realsense_gazebo_plugin, Jan 2019. URL: https://github.com/pal-robotics/realsense_gazebo_plugin.

[67] Dirk Thomas, William Woodall, and Esteve Fernandez. Next-generation ros: Building on dds. *ROSCon Chicago*, 2014, 2014.

[68] Sebastian Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):52–57, 2002.

[69] M.A. Vega Torres. Occupancy Grid Map to Pose Graph-based Map for long-term 2D LiDAR-based localization, November 2022. doi:10.5281/zenodo.7330270.

[70] M.A. Vega Torres, A. Braun, and A. Borrmann. Occupancy grid map to pose graph-based map: Robust bim-based 2d- lidar localization for lifelong indoor navigation in changing and dynamic environments. In *eWork and eBusiness in Architecture, Engineering and Construction: ECPPM 2022*. CRC Press, Sep 2022. URL: https://publications.cms.bgu.tum.de/2022_ECPPM_Vega.pdf, doi:10.1201/9781003354222-72.

[71] M.A. Vega Torres, A. Braun, and A. Borrmann. Bim-slam: Integrating bim models in multi-session slam for lifelong mapping using 3d lidar. In *Proc. of the 40th International Symposium on Automation and Robotics in Construction (ISARC 2023)*, Jul 2023.

[72] M.A. Vega Torres and F. Pfitzner. Investigating robot dogs for construction monitoring: A comparative analysis of specifications and on-site requirements. In *Proc. of the 34th Forum Bauinformatik*, Sep 2023.

[73] Ignacio Vizzo, Tiziano Guadagnino, Benedikt Mersch, Louis Wiesmann, Jens Behley, and Cyrill Stachniss. KISS-ICP: In Defense of Point-to-Point ICP – Simple, Accurate, and Robust Registration If Done the Right Way. *IEEE Robotics and Automation Letters (RA-L)*, 8(2):1029–1036, 2023. doi:10.1109/LRA.2023.3236571.

[74] Alexander W Winkler, Dario C Bellicoso, Marco Hutter, and Jonas Buchli. Gait and trajectory optimization for legged systems through phase-based end-effector parameterization. *IEEE Robotics and Automation Letters (RA-L)*, 3:1560–1567, July 2018. doi:10.1109/LRA.2018.2798285.

[75] Hao Yang. Create pgm map from gazebo world file for ros localization. [Online]. URL: https://github.com/hyfan1116/pgm_map_creator.

[76] Ji Zhang and Sanjiv Singh. Loam: Lidar odometry and mapping in real-time. In *Robotics: Science and Systems*, volume 2, pages 1–9. Berkeley, CA.

[77] Kaiyu Zheng. Ros navigation tuning guide. *Robot Operating System (ROS) The Complete Reference (Volume 6)*, pages 197–226, 2021.

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

14. July 2023 Anuj Berwal