Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня
магістра
(назва освітньо-кваліфікаційного рівня)

студента *Панасейко Ганни Миколаївни*
(ПІБ)

академічної групи *121М-19-1*
(шифр)

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

на тему: *Моделі, алгоритми та програмне забезпечення для планування шляху для навігації мобільних роботів з уникненням перешкод на основі дерева октантів*

____________________ *Г.М. Панасейко*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	Доц. Сироткіна О.І.			
економічний	Доц. Касьяненко Л.В.			

Рецензент				

Нормоконтролер	Доц. Сироткіна О.І.			

Дніпро
2020

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп’ютерних систем
(повна назва)
_______________ І.М. Удовик
(підпис) (прізвище, ініціали)
« » ____________ 20 20 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студенту *121М-19-1* *Панасейко Ганні Миколаївні*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Моделі, алгоритми та програмне забезпечення для планування шляху для навігації мобільних роботів з уникненням перешкод на основі дерева октантів*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 22.10.2020 р. № 888-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об’єкт досліджень – процес оптимізації та покращення точності руху та уникнення перешкод для навігації мобільних роботів.

Предмет досліджень – моделі та методи виявлення перешкод та навігації з метою уникнення виявлених перешкод.

Мета роботи – підвищення ефективності системи розпізнавання перешкод мобільними роботами для навігації у середовищі, використовуючи датчики для забезпечення дороги без зіткнень з об’єктами, які не знаходяться на одному рівні з лазерами.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна полягає у тому, що удосконалено методи системи планування шляху для навігації мобільних роботів на основі дерева октантів для якісного та точного шляху від початкової точки до заданої у просторі.

Практична цінність результатів полягає у тому, що запропонована система навігації та обходження перешкод дозволяє помічати перешкоди які знаходяться не на одному рівні з рівнем лазерів мобільного робота.

4 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз теми та постановка задачі	12.09.2020-30.09.2020
Побудова системи розпізнання перешкод у тривимірному просторі засобами ROS, Octomap, teb_local_planner для удосконалення планування шляху для навігації мобільних роботів.	01.10.2020-31.10.2020
Удосконалення автоматизованої системи та її тестування для вирішення задачі розпізнання та обходження перешкод мобільним роботом у просторі.	01.11.2020-07.12.2020

Завдання видав	____________ (підпис)	*Сироткіна О.І.* (прізвище, ініціали)
Завдання прийняв до виконання	____________ (підпис)	*Панасейко Г.М.* (прізвище, ініціали)

Дата видачі завдання: *12.09.2020 р.*

Термін подання кваліфікаційної роботи до ЕК *10.12.2020*

РЕФЕРАТ

Пояснювальна записка: 121 стор., 47 рис., 3 таблиці, 6 додатка, 60 джерел.

Об'єкт дослідження: процес оптимізації та покращення точності руху та уникнення перешкод для навігації мобільних роботів.

Предмет дослідження: моделі та методи виявлення перешкод та навігації з метою уникнення виявлених перешкод.

Мета магістерської роботи: підвищення ефективності системи розпізнавання перешкод мобільними роботами для навігації у середовищі, використовуючи датчики для забезпечення дороги без зіткнень з об’єктами, які не знаходяться на одному рівні з лазерами.

Методи дослідження. Для вирішення поставлених задач використані методи: пошуку шляхів, порогового значення, обробки хмари точок, генерації дерева октантів.

Наукова новизна полягає у тому, що удосконалено методи системи планування шляху для навігації мобільних роботів на основі дерева октантів для якісного та точного шляху від початкової точки до заданої у просторі.

Практична цінність результатів полягає у тому, що запропонована система навігації та обходження перешкод дозволяє помічати перешкоди які знаходяться не на одному рівні з рівнем лазерів мобільного робота.

Список ключових слів: ROS, навігація, мобільні роботи, уникнення перешкод, дерево октантів, лазер, хмара точок.

ABSTRACT

Explanatory note: 121 pages, 47 figures, 3 tables, 6 applications, 60 sources.

Object of research: the process of optimizing and improving the accuracy of movement and avoiding obstacles to the navigation of mobile robots.

Subject of research: models and methods of obstacle detection and navigation avoiding detected obstacles.

Purpose of master's thesis: increase the efficiency of the obstacle recognition system by mobile robots for navigation in the environment, using sensors to ensure the road without collisions with objects that are not on the same level with the lasers.

Research methods. Methods of path finding, thresholding, point cloud processing, octree generation were used to perform the task.

Originality of research is determined by the improved methods of path planning system for navigation of mobile robots based on the octant tree for a high-quality and accurate path from the starting point to a given in space.

Practical value of the results is that the proposed system of navigation and bypassing of obstacles allows you to notice obstacles that are not on the same level with the level of lasers of the mobile robot.

In the Economics section we calculated the complexity of software development, the cost of creating software and the duration of its development, as well as marketing research of the market for the created software product.

Keywords: ROS, navigation, mobile robots, obstacle avoidance, octree, laser, point cloud.

CONTENTS

LIST OF ACRONYMS

2D	Two-Dimensional
3D	Three-Dimensional
CPP	Classical Path Planning
GPP	Global Path Planner
GPS	Global Positioning System
OS	Operating System
PC	Personal Computer
ROS	Robot Operating System
RST	Robotics System Toolbox
SLAM	Simultaneous Localization and Mapping
TEB	Timed Elastic Band

INTRODUCTION

The relevance of research. Robotics develops more and more every year. New approaches are being created to solve the problems of motion, localization, and automation of robots. Many models have made great strides in solving various problems. A lot of technical complexes are created for military purposes: target detection, its elimination. Robot firefighters are being created; rescue robots capable of getting people out of the water, from the rubble of collapsed buildings. One of the many robotics trends is the transition from remote-controlled systems, which require constant human participation to perform all the robot's actions, to autonomous systems in which the operator only indicates the final and intermediate goals. This is convenient for carrying out alien research, where a large signal delay does not allow remote control.

Mobile robotic systems are used today in a wide variety of industries. The broader the scope of their application, the more stringent the requirements for their performance for specific tasks become. One of the most urgent such requirements relates to the robot's autonomy and navigation capabilities.

With the development of such areas of science as the theory of automatic control and artificial intelligence, a modern robot has learned not only to carry out the program laid down for it but also, based on a variety of sensors and related systems of all kinds of information, to independently make decisions and minimize errors when performing tasks assigned to it.

The research is now progressing towards autonomous robots that will assist us in our daily lives. One of the enabling technologies is navigation, and navigation is the subject of this thesis.

Navigation remains the main problem of all currently existing mobile devices moving independently. For successful navigation in space, the robot's on-board system must be able to plan a path, as a strategy to find a path towards a goal location, correctly interpret information about the world around it received from the sensors. Moreover, the robot should control movement parameters, its own coordinates and be able to adapt to environmental changes.

One of the most challenging aspects of autonomous outdoor mobile robot navigation is reliability. That is, a mobile robot must be able to reach its destination safely, every single time, not only avoiding collisions to obstacles and humans around it but also successfully driving through difficult paths such as slopes, bumps, or potholes.

The most crucial factor is to provide the robot with the necessary sensors or subsystems to determine the presence, type, and distance of an obstacle. It is also essential to form accurate and timely actions to change the trajectory of movement and make other motor decisions.

The purpose of the research is to create an updated system for planning a mobile robot's path by avoiding obstacles using the octree. The aim is to add sensors to ensure the road of the path without collisions with objects that are not on the same level as the lasers. The goal is to explore recent studies about the 2D obstacle avoidance for a mobile robot. Develop an improved method for obstacle detection in a dynamic environment, plan the route for a mobile robot with avoidance of detected obstacles. The developed solution should take point cloud as input data. The algorithm is based on processing point cloud data to an octree, building projection, and converting it to obstacles.

Tasks of research. For the achievement of the set purpose in work, the following tasks are formulated and solved:

1. Consider the basic principles of navigation.

2. Consider the existing sensors and choose the most suitable for this mobile robot.

3. Consider the necessary ROS elements for navigation.

4. Use the method of reducing the resolution of the point cloud and cutting off excess elements using PCL.

5. Set up a system of continuous generation of octree from a point cloud using octomap, followed by the projection of occupied cells on the floor.

6. Develop a package to convert the resulting projection into polygons and then send them to the topic of obstacles for teb_local_planner.

Object of research: Process of mobile robot navigation among 3D obstacles using octrees.

Subject of research: Models and methods of obstacle detection and navigation avoiding detected obstacles.

Research methods. Methods of path finding, thresholding, point cloud processing, octree generation were used to perform the task.

Originality of research. The scientific novelty of the thesis results is determined by the development of a new, improved path planning system for navigation of mobile robots based on the octant tree for a high-quality and accurate path from the starting point to a given in space.

After analyzing modern work in the field of mobile robot navigation and avoiding obstacles in real-time, it was found that in contemporary control theory, research in robotics, the most promising areas of development are the use of algorithms based on data processing from sensors. Some research focus on the internal system of localization of the robot on basis of ultrasound. One of the simplest means of movement on the specified route is odometry, which is based on establishing the route of movement of the robot by determining the movement of the robot wheels. The strong disadvantage of this technique is the accumulation of error during movement, so it is advisable to use it with other means of navigation, to correct this error.

Personal contribution of the author:

1. Scientific results of the work are obtained by the author independently.
2. Choice of research methods and implementation technologies.
3. Development of the theoretical part of the work, which explores and systematizes knowledge of existing approaches of path planning for mobile robot navigation.
4. Development of the new system for mobile robot navigation and obstacle avoidance using octree.
5. Testing and evaluation of the results.

Structure and scope of work. The work consists of an introduction, four sections and conclusions. It contains 121 pages, including 100 pages of text of the main part with 47 figures, a list of used sources with 60 items on 6 pages, 4 appendices on 15 pages.

SECTION 1
ANALYSIS OF MOBILE ROBOT NAVIGATION AND LOCALIZATION

1.1. Mobile Robot Navigation

Mobile robotics is currently one of the fastest-growing areas of scientific research. This solutions-oriented branch of industry, merging interdisciplinary sciences and technologies such as Engineering, Computer Science, Cognitive Sciences, Artificial Intelligence, and Mechanics. They have become more evident in commercial and industrial conditions.

The mobile robots should have navigational skills to work harmoniously within an integrated environment with humans. Humans expect that robots will be able to operate and navigate in their environments without collisions or interference.

Modern robotics began its rapid development in the 1970s. At that time, the first models appeared that effectively reproduced the essential functions of a person. Models of the first generation were very popular and found wide application in the scientific and industrial fields. Over time, adaptive robotics appeared - mobile robots on a new generation platform. Improved modifications could build an optimal route of movement, if necessary, make adjustments to the approved trajectory of movement.

A mobile robot moves to solve specific problems, receives data from external sensors, and must continuously process information in order to control its movement. All these processes take place continuously and are closely interconnected with each other. The use of vision in mobile robotics has become widespread.

Mobile robots with an autonomous navigation system could navigate in space due to mounted scanning sensors. Unique computer technology received a signal that made it possible to make the right decision on the way along the route. Navigation served as the coordinator of the movement of robotic equipment. Over time, more advanced robot navigation has emerged using upgraded sensors, gyroscopes, satellite navigation, lasers, and ultrasonic devices.

Thanks to their abilities, mobile robots can replace humans in many areas. Applications include Surveillance, Planetary Exploration, Patrolling, Rescue Operations, Exploration, Petrochemical Applications, Industrial Automation, Construction, Entertainment, Museum Guides, Personal Services, Extreme Environment Intervention, Transportation, Medical Services, etc., and much more other industrial and non-industrial applications. Most of them are already available on the market.

Mobile robots can move in various environments: in water, air, on the ground, in space. Movement in each environment has its own characteristics associated with their different physical properties.

A mobile robot's components are a controller (embedded microcontroller or a computer), different sensors, which depend upon the robot`s requirements, actuators, and power systems. An actuator is a component that is responsible for moving and controlling. They refer to the motors that move the robot can be wheeled or legged. Usually, DC power supply is used instead of AC to power a mobile robot.

To control a mobile robot in the event of movement along a trajectory, a position and orientation system in space is required. Currently widely used [2]: encoders, interparticle systems, GPS, and rangefinders. The growth in embedded computing solutions' productivity allows processing video data of the robot’s onboard vision system in real-time [3] and using them to solve more and more complex problems, incl. and for navigation.

The visual odometry method [3-6] is based on measuring the displacement of crucial points in space [7], information about which is obtained from the analysis of the sequence of stereo images of the technical vision system. Such a system can work in a non-deterministic environment; it allows solving related tasks, such as, for example, building a room map.

Several approaches can be used to classify robots - for example, by scope, purpose, method of movement, etc. By the main application's scope, industrial robots, research robots, robots used in training, and unique robots can be distinguished. The most important classes of general-purpose robots are manipulation and mobile robots.

A manipulation robot is an automatic machine (stationary or mobile), consisting of an executive device in the form of a manipulator with several degrees of mobility and a program control device, which serves to perform motor and control functions in the production process.

A mobile robot is an automatic machine that has a moving chassis with automatically controlled drives.

1.2. Concepts and Algorithms of Mobile Robot Navigation

Robot navigation means the robot’s ability to determine its own position in its frame of reference and then to plan a path towards some goal location. The mobile robot requires a map of the environment and the ability to interpret that representation.

Navigation can be defined as the combination of the three fundamental competencies:

1. Self-localization.
2. Path planning.
3. Map-building and map interpretation.

Talking about the navigation of mobile robots, there are three main types of systems - global, local and personal. The global type defines the absolute coordinates of the robot when moving over large areas and objects. Local navigation fixes coordinate following the specified parameters. For this type of navigation, the zone limits must already be known. Personal navigation implies positioning, considering objects that are nearby. Typically, the presented modification of the mobile robot is equipped with a special manipulator.

It is believed that the larger the device, the higher the importance of global navigation for it and the lower the personal one.

Navigation systems can be passive and active. A passive navigation system implies receiving information about one’s own coordinates and other characteristics of its movement from external sources, while an active one is designed to determine the location only on its own. As a rule, all global navigation schemes are passive, local ones are both, and personal schemes are always active.

Navigation without a map (for example, the DistBug algorithm) is often used in a constantly changing environment or when the calculated path only needs to be taken once and therefore does not have to be optimal. If you have a map, you can use the Dijkstra algorithm or the A * algorithm, which allows you to determine the shortest path before the robot starts moving. Navigation without a map is carried out while the robot is moving in direct interaction with its sensors. Map-enabled navigation algorithms rely on a nodal distance graph, which must either be provided in advance or built based on environmental analysis (for example, using a quadtree) [8].

Below will be presented and discussed various methods and algorithms of navigation used to operate mobile robots.

The potential field method is an algorithm for global trajectory construction on the map using virtual forces. This method requires knowing the start and endpoints of the route and the positions of all obstacles and walls.

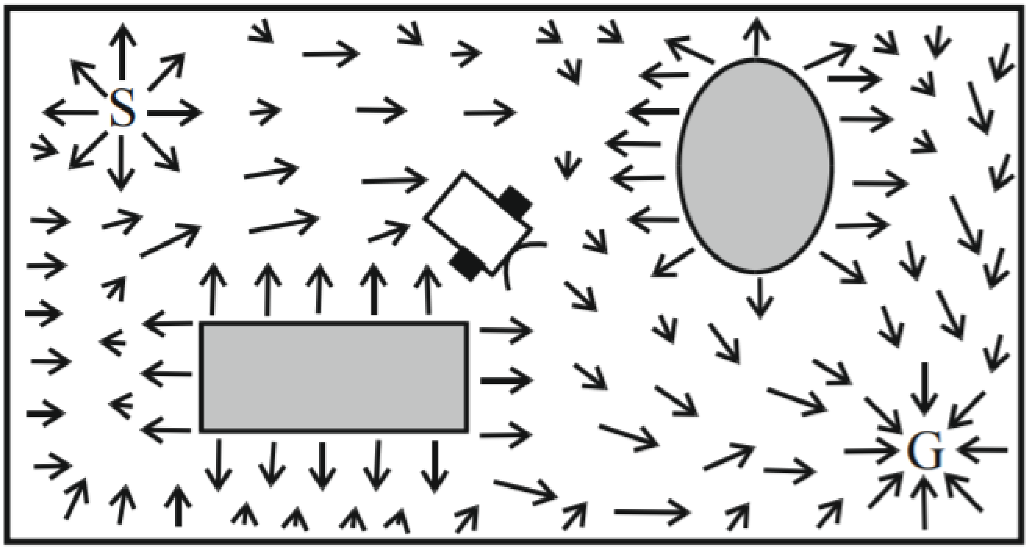


Fig. 1.1. Potential field

Fig. 1.1 shows precisely how the force field is formed: repulsive forces emerge from obstacles and walls, the preposition of which determines the direction of the potential field from the starting point of the route to the final one.

The trajectory is built using virtual forces of attraction and repulsion. The starting point, all obstacles, and boundary walls repel the robot, and the target point attracts it. The magnitude of each force is inversely proportional to the distance to the object. The robot simply moves in a force field.

The wandering point algorithm is a local trajectory planning algorithm. Its implementation requires the use of a local distance sensor.

The algorithm is to try to reach the target route point in a straight line. When an obstacle is detected, it is necessary to measure the angles of rotation to the right and left, ensure the avoidance of the obstacle, and turn to a smaller angle. Next, you should move along the obstacle's border until it stops interfering with the passage to the target point.

Fig. 1.2 shows the successive positions (1 ... 6) of the robot when it moves from the starting point to the target point. The target is not visible from the starting point; therefore, the robot switches to the mode of avoiding obstacles along the contour and moves to point 1, from which it can again move to the target without hindrance. At point 2, the robot again encounters an obstacle and begins to go around it in the kennel. Finally, he gets to point 6, from which he can reach the goal in a straight line, without encountering obstacles in his path.

In real conditions, the robot moves not strictly along the found route points but along an approximating curve.

The disadvantage of this method is that with specific placement of obstacles, the robot can get stuck in its movement. In this case, he will endlessly bend around obstacles and never reach the goal [9].

Bug family algorithms solve the problem of local traffic planning and guarantee its convergence. If the trajectory exists, then it will be found, and if not, then the algorithm will determine that the goal is unattainable.

The algorithms of the Bug family use as input the coordinates of the robot (distance traveled), the coordinates of the target point, and the readings of a contact sensor (for Bug1 and Bug2) or a rangefinder (for DistBug).

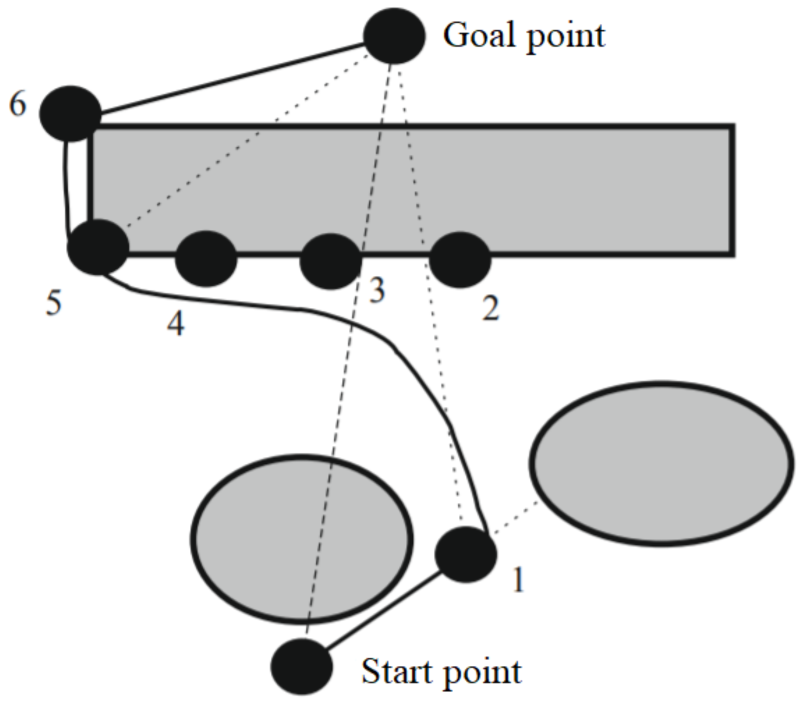


Fig. 1.2. The wandering point algorithm

The essence of the algorithms of the Bug family is as follows:

Bug 1: Move straight to the target before hitting an obstacle (to the meeting point). Bypass the contour's obstacle entirely, registering the shortest distance to the target (at the vanishing point). Upon re-reaching, the meeting point, return to the vanishing point and continue the execution of the algorithm [10].

Bug 2: Draw an imaginary line M from the starting point to the target point. Move along M until it collides with an obstacle (to the meeting point). Follow the object's boundary to the point of intersection with the line M, which is closest to the target (vanishing point). Continue execution of the algorithm [10].

DistBug: Move straight to the target before hitting an obstacle (to the meeting point). Follow the boundary of the obstacle, registering the shortest distance to the target. If the target becomes visible or enough free space in its direction, continue the algorithm from this point (vanishing point). If the robot returns to the previous meeting point, then the goal is unattainable [11].

This algorithm is poorly implemented in practice since real conditions do not provide the robot's required positioning accuracy and the readings of its rangefinder. Almost all modifications of the DistBug algorithm are sensitive to the noise of sensor data and errors in the robot's movement.

Currently, many algorithmic methods for localization, navigation, and mapping have been developed, and most often, probabilistic methods that minimize uncertainty are used to solve localization problems comprehensively.

1.3. Sensors

Mobile robots need information about the world around them so they can relate to their environment. Therefore, they rely on sensory devices to convert external stimuli into electrical signals. These signals are electrical data representing the state of the surrounding world and must be interpreted by the robot to achieve its goal. There is a wide range of sensors used to this end.

Sensors play one of the essential roles in robotics. With the help of various sensors, the robot “feels” himself and the world around him. These are the senses - eyes, ears, skin for robots.

A mobile robot is an “intelligent” self-propelled mechanism. For it to be able to provide meaningful behavior and perform functionally complete work with the help of its “intellectual” capabilities, it must be equipped with various sensory systems that allow the robot to perceive the environment and navigate in it. As with humans and most animals, the most important and significant of these systems in the robot's movement is the visual sensor.

The robot moves blindly without environmental sensors. External sensors are involved in:

– recognition of places and objects that have already been encountered;

– determining free space and planning movement in it in order to avoid collisions with obstacles;

– interaction with objects, ice, and animals;

– creating a general idea of the environment around the robot.

Sensors play one of the most critical roles in robotics. With the help of various sensors, the robot “feels” himself and the world around him. These are the senses - eyes, ears, skin for robots.

A different amount of data with a specified frequency sends, depending on the sensor's sort and its goal. However, an individual microprocessor has its own limit of information that can be received every time.

When dealing with Robot Navigation, sensors are usually used for positioning and obstacle avoidance. In terms of positioning, sensors can be divided into relative and absolute. Relative positioning sensors include odometry and inertial navigation, which are methods that measure a robot’s position concerning the robot’s starting point and movement. In particular, absolute positioning sensors recognize structures in the environment whose position is known, which allows the robot to estimate its own position.

The camera can be represented as the eyes of the robot, and the images taken from the camera are useful for recognizing the environment around the robot. For example, object recognition using a camera image, facial recognition, a distance value obtained from the difference between two different images using two cameras (stereo camera), mono camera visual SLAM, color recognition using information obtained from an image, and object tracking are very useful.

Map-based positioning is a technique in which robots use their sensors to create a local map of the environment. This local map is compared to a known global map stored in the robot’s memory. If the local map matches a part of the global map, the robot will be able to estimate its position.

Laser-based distance sensors with 1D or 2D sensors do not transmit large data. At the same time, the problem is more associated with the cameras. They address enough data and expect excellent processing power, which is challenging for a microprocessor.

There are several sensor packages offered by the sensor available on ROS. Sensors are classified into 1D rangefinders (Infrared distance sensors for low-cost robots), 2D range finders (LDS is frequently used in navigation as in the algorithm presented in chapter 4), 3D Sensors (such as Intel’s RealSense, Microsoft’s Kinect are needed for 3D measurements), Pose Estimation (GPS + IMU), Cameras (that are commonly utilized for object and gesture recognition, face recognition and 3D SLAM), Audio/Speech Recognition and many other sensors.

Laser Distance Sensors (LDS) consist of a distant sensor such as Light Detection and Ranging (LiDAR), Laser Range Finder (LRF), and Laser Scanner. LDS sensor finds the distance to an object and uses a laser in origin. This type of sensor ensures great performance and speed with data collection in real-time. LDS is frequently used in all systems where distance evaluation is compulsory. Technology is widely applied in robotics, where it is one of the fundamental sensors for distance recognition.

LDS usually includes a single laser source, a motor, and a reflective mirror. The motor rotates the inner mirror while it is scanning using the laser. The range of the LDS goes from 180° to 360°.

LIDAR is also known as Light Imaging Detection and Ranging. It is a technology that detects objects on the surface, as well as their size and exact disposition. The device emits laser pulses that move outwards in various directions until the signals reach an object and then reflect and return to the receiver.

Modern laser devices can generate a scanner in several hundred thousand seconds and use a system of movable mirrors or the scanner body itself. As a result of such measurements or “scanning”, we obtain three-dimensional points in a short time, with a large and complete described object.

The main result of laser scanning - be it ground, air, or mobile - is a cloud of three-dimensional points describing the surveyed object's geometric parameters with varying accuracy. The number of laser reflections obtained when photographing a surveyed object is often hundreds of millions and even billions. The processing of such data arrays and the formation based on final products for users in various fields of activity is today the most laborious component of laser technology.

The invention of this technology has had a tremendous impact on the development of the automotive industry. Self-driving and driverless cars use LIDAR to scan surroundings and plan a car’s behavior in order to avoid collisions with obstacles.

1.4. SLAM

One of the main tasks of control systems for autonomous robots is the navigation task, and its subtask is the localization of the robot in space.

In automated navigation of mobile robots, simultaneous localization, and mapping (SLAM) technology is a computational scheme for building or updating a map of an unknown environment while simultaneously tracking an agent's location within it.

Visual SLAM approaches are usually divided into two main branches: smoothing approaches based on bundle adjustment and filtering approaches based on probabilistic filters. The latter is divided into three main classes: dense, sparse, and semantic approaches. Dense approaches are able to build dense maps of the environment, which make the algorithms more robust but at the same time heavy in terms of computational requirements; indeed, most of these approaches can work in real-time only when dedicated hardware is used. Sparse approaches address the problem of computational requirements by the map; obviously, this choice impacts the robustness of the solution. These algorithms require less computational effort because they try to allocate in memory only the most significant vital points representing the map; for this reason, they are natural candidates for a real-time Visual SLAM implementation. Semantic approaches extract higher-level semantic information from the environment in order to build a more robust and compact map.

To date, there is already a large number of algorithms for the implementation of SLAM technology with a known, at least approximate, solution. Such techniques can include superior filtering — an advanced Kalman filter or particle filter — and static data estimation. These SLAM algorithms are essentially geared towards the available hardware resources and operational compliance with those resources' constraints and associated computational efficiency.

According to the above scheme, the algorithms implementing the SLAM technology use various computational schemes to solve the localization and mapping problem based on data obtained from various sensors.

The primary (original) SLAM algorithms include Kalman filters, particle filters, and estimates using various interval calculations. They provide an estimate of the above probability function for the robot pose and the map parameters.

A popular technique in SLAM technology is an equation based on the use of images from image sensors. The adjustment is performed to eliminate residuals due to the presence of errors in excessively measured values. Moreover, it is performed to determine the most probable values of unknowns close to these probable ones. During the adjustment process, this is achieved by determining corrections to the measured values (angles, directions, line lengths or elevations). Often the adjustment is performed using the least-squares method to minimize residuals.

SLAM alignment allows poses and feature positions of landmarks to be jointly estimated, increasing map accuracy, and is used in many new systems incorporating SLAM technology.

New algorithms for SLAM technology are still an active area of research, which is carried out according to various requirements and assumptions about the types of maps, sensors, and processing models. Many of the modern systems that implement SLAM technology are not based on a combination of different approaches.

In general, the SLAM algorithm can be described as a repeating sequence of steps:

1. Scanning the surrounding area.
2. Determination of displacement based on comparison of the current frame with the previous one.
3. Selection of mark features on the current frame.
4. Comparison of the marks of the current frame with the marks obtained for the entire history of observations.
5. Updating information about the position of the robot for the entire history of observations.

6. Checking for loops - does the robot pass through the same part of the terrain repeatedly.

7. Alignment of the general map of the world (starting from the position of the marks and the robot in the entire history of observations).

At each step of the algorithm, the robot has assumptions about the world's structure and the history of the movement in it.

Standard SLAM methods will be described below.

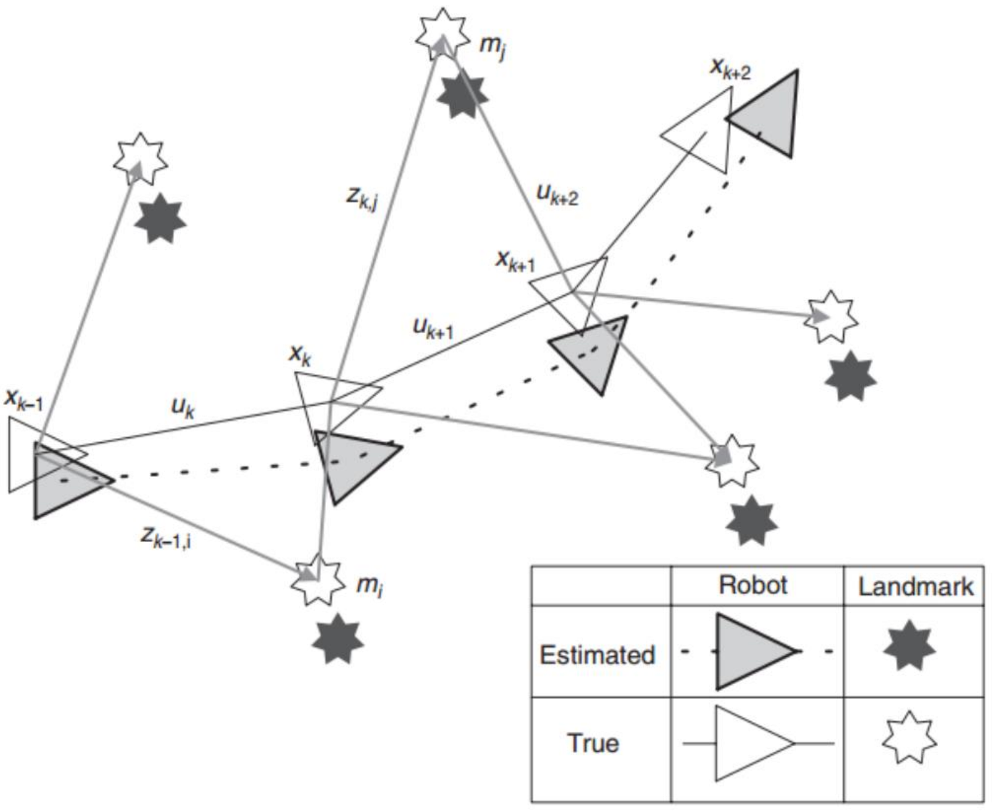


Fig. 1.3. SLAM concept diagram.

Basic SLAM will be called the initial versions and schemes of SLAM algorithms. A popular method for solving the SLAM problem has been the extended Kalman filter. At each step, the robot has a set of previously received landmarks and received data. Based on the new and previous frames, it is possible to determine the robot's displacement and predict the new position of the robot. In the same place, from the new frame, you can select the robot's position relative to them. Based on the difference between these two estimates of the positions of the robot, the probabilities/weights for all landmarks are updated, and the poses - the trajectory of the robot - is adjusted. A typical SLAM layout is shown in Fig. 1.3.

It shows the real trajectory and real positions of landmarks around the robot and the same characteristics obtained as estimates from the available measurements of the robot’s sensors. The goal of navigation algorithms is obviously to keep the data from diverging too much.

Monocular SLAM has received much attention in the last years.

In this approach, the map and the camera pose are stored at stochastic variables, and the system evolution is estimated by an incremental Extended Kalman Filter (EKF). Inverse depth parametrization can be used to represent point features, which permits the efficient and accurate representation of point features, which permits the efficient and accurate representation of uncertainties.

MonoSLAM is an algorithm that uses data from a single monocular camera on a robot. This is the difference between this method and the methods using laser scanners. As in classical laser scanner-based SLAM approaches, it is assumed that the robot’s pose is described as a stochastic variable with a Gaussian distribution, and the environment map is not overly saturated. The environment is described by a limited set of geometric features, i.e., has geometrically measurable characteristics. Features are also described as Gaussian variables. The method builds a probabilistic 3D map of the robot’s environment and estimates the position of the robot in it.

The system state determined by the robot determines the pose of the robot and is plotted on a map represented at any time $t = kt$, where t – is the time since the previous step; it is also specified as a stochastic variable with a Gaussian distribution.

Information regarding the movement of the camera at any time is encoded in the vector x_k, built as the trajectory of the robot.

FastSLAM is an efficient SLAM algorithm. Using this filter is computationally more efficient than using the classical Kalman filter. This method is based on the idea of splitting a large-dimension filter into a set of low-dimension filters, which makes it possible to reduce the computational complexity of the problem and accordingly speed up the calculations. In the case of large K, the problem's dimension increases proportionally, which complicates the application of traditional methods. Particle filters are also useful for small dimensions but problematic multidimensional areas as long as the number of particles is not increased. Practically direct implementation of these methods allows you to process maps with several hundred landmarks efficiently, but thousands of landmarks do not allow real-time processing.

The FastSLAM method is based on the fact that the individual measurements of different landmarks' positions are independent of each other. Between the measurements of different landmarks, a probabilistic dependence arises through the parameters of the robot's posture - errors in measuring other landmarks introduce errors in the determined pose and through them, affect the errors in measuring other landmarks. However, these estimates will be conditionally independent if we consider the conditional distributions of estimates of the positions of landmarks relative to a given distribution of the measured posture of the robot.

In addition, as an additional advantage of this method, it should be noted that it is more resistant to identification errors. Identification in SLAM tasks is understood as the identification of landmarks between measurements, that these two observations at different points in time refer to the same landmark. In the classical method, an identification error can lead to "catastrophic consequences". In the case of the FastSLAM method, instead of a single filter, M filters are used, tied to individual particles, and therefore it is possible to update these filter banks in different ways based on different hypotheses about identifying landmarks. In the future, particles with maps based on incorrect identification attached to them will lose the accuracy of identification measurements, lose the accuracy of measurements, and will be eliminated by the particle filter, which will allow the method to "straighten" and restore accuracy after incorrect identification.

In a sense, visual SLAM summarizes the way monoSLAM works.

SLAM uses camera images to plan the robot's position in a new environment. This method works by tracking features in images between camera frames and determining the robot's pose and position of features in the world based on their relative movement. This actually provides an additional odometer source that is useful for locating the robot. Since camera sensors are generally less expensive than laser sensors, this can be a useful alternative or addition to laser-based localization techniques.

Processing information from cameras consists in finding prominent details on them - landmarks and analyzing the displacement of these landmarks between different frames taken by the camera. In general, the camera position is described by six parameters, and matching three landmarks is required to determine all six. However, the determination of two angles by the tilt sensor makes it possible to determine the remaining parameters only by two landmarks. After the frame is normalized, the scale and rotation adjustments determine the displacement along the horizontal axes.

1.5. Point cloud

A point cloud is the most accurate digital record of an object or space, stored as a vast number of points covering the object's surface. Technically, a point cloud is the most accurate digital record of an object or space, stored as a considerable number of points covering the object's surface.

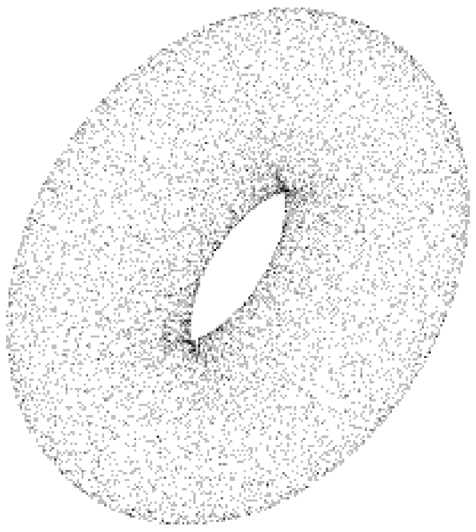


Fig. 1.4. Point cloud example

Points represent a three-dimensional shape or object. Each point has its own set of X, Y, and Z coordinates. All the coordinates relating to such an image are included in a concept called a point cloud. Point clouds represent all the coordinates that help illustrate the external surface of a three-dimensional object. Point clouds are usually created using 3D scanners or photogrammetry software that measures many points on the outer surfaces of objects around them.

Although point clouds can be directly visualized and inspected, they are generally not used directly in most 3D applications, and therefore tend to be converted to a polygon mesh or voxels.

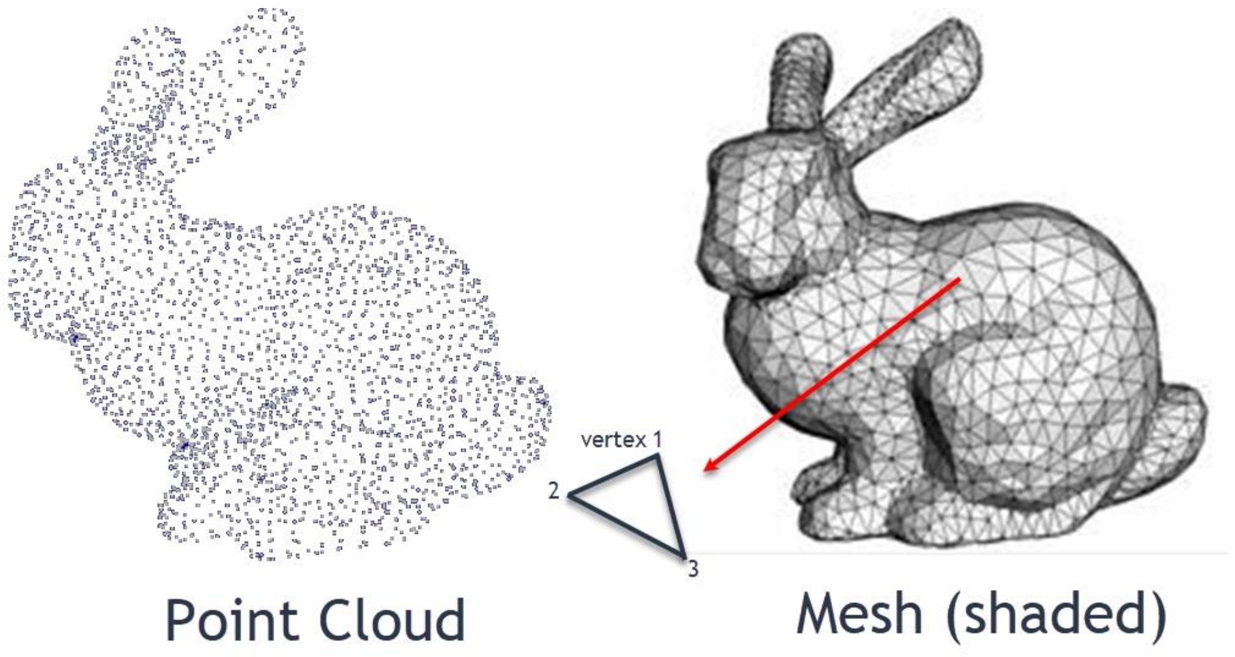


Fig. 1.5. From Point Cloud to Mesh

It is essential to understand that a point cloud is a collection of separate, unrelated points with a specific position and color. This makes it easy to edit, display, and filter point clouds.

Using separate, unrelated points is key to the practical use of point clouds since points are the easiest objects to process. The computer does not have to worry about the scale, rotation, and relation of points to other objects; only position and color are essential for the calculation.

Visible access to the scanned surfaces is a crucial factor in obtaining point cloud data. Regardless of how (scanner or photo) the survey is carried out, it will not be able to get points on surfaces that are not visible from the position chosen for data collection. This means that, more often than not, covering the entire object requires obtaining data from several different positions and then combining them.

The term “density” is used to describe the resolution of the collected dataset, which usually means the distance from the point to point. Less dense point clouds are obviously caught much faster.

Most point cloud databases contain not only information about the position of points but also a description of their visual properties, such as colors or reflectivity. All this may or may not be included in the point cloud, which is an additional factor affecting the speed of the survey and the processing of the obtained data.

Since a point cloud is a spatial dimension, it is possible to quickly measure something without contacting an object.

– Informative - you are not limited by drawings and see everything in 3D from all sides.

– Short production time - getting a point cloud faster than a 3D object model.

– Easy to use - you do not need expensive software.

– Laser scanning is a young technology that makes it possible to master a new product that is actively used in the world.

– 3D data is always more informative than 2D drawings.

– Any measurements available - you can take measurements of any beam on the ceiling, any inaccessible structural element using the cloud.

1.6. Voxel

The name voxel comes from “volumetric elements,” and it represents the generalization in 3D of the pixels. Voxels represent the traditional way to store volumetric data. They are organized in axis-aligned grids subdividing and structuring space regularly. Voxel representations offer a promising way to store volumetric data in order to unify texture and geometrical representations while simplifying filtering. The significant advantage of voxels is the richness of this representation and the very regular structure, which makes it easy to manipulate. That makes voxels an excellent candidate to address aliasing issues that are hard to deal with in triangulated models.

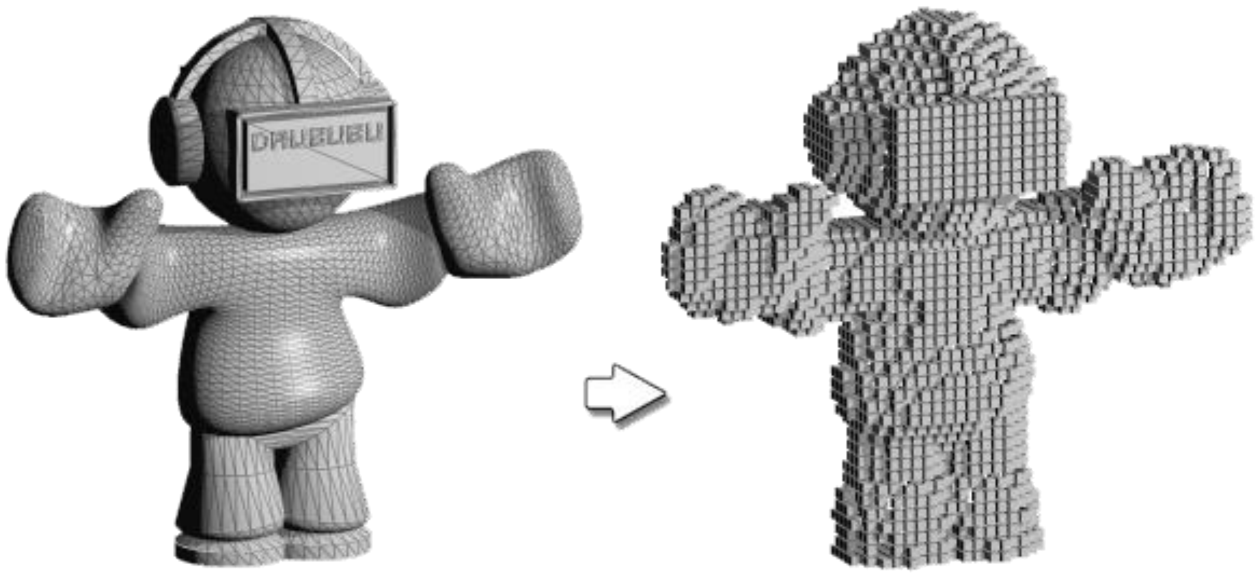


Fig. 1.6. Converting polygonal graphics to voxels

Voxels are often used to visualize and analyze medical and scientific data (such as GIS). Some 3D displays use voxels to describe their resolution.

In 3D computer graphics, a voxel is a value on a regular grid in three-dimensional space. As with pixels in a 2D bitmap, voxels themselves usually do not have their position (coordinates) explicitly encoded by their values. Instead, rendering systems determine the position of a voxel-based on its position relative to other voxels (that is, its position in the data structure that makes up a single volumetric image). The word voxel is similar to the word “pixel”, where “vo” represents “volume” and “el” represents “element”; similar formations with “el” for “element” include the words “pixel” and “texel”.

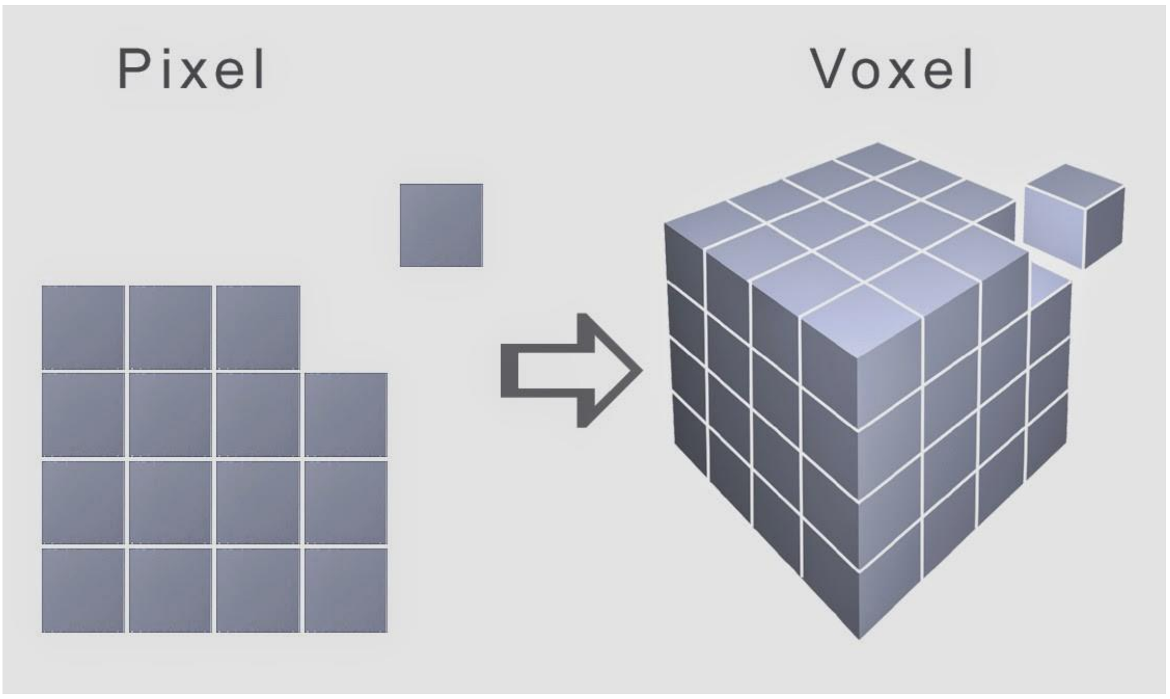


Fig. 1.7. Converting pixels to voxels

A voxel is a single sample or data point on an evenly spaced 3D grid. This data point can consist of a single piece of data, such as opacity, or multiple pieces of data, such as color, in addition to opacity. The voxel represents only one point on this grid, not a volume; the space between each voxel is not represented in the voxel dataset. Depending on the type of data and the intended use of the dataset, this missing information can be reconstructed and/or approximated, for example, using interpolation.

If a fixed voxel shape is used throughout the model, it is much easier to work with voxel anchor points, that is, with three coordinates of this point. But there is also a simple form of notation - the indices of the elements in the model set, that is, integer coordinates. The elements of the model set in this case are the state parameters indicating whether the voxel belongs to the modeled object or its separate parts, including their surfaces.

Voxels can contain multiple scalar values, mostly vector (tensor) data; in the case of ultrasound scans with B-mode and Doppler data, the density and volumetric flow rate are captured as separate data channels referring to the same voxel positions. While voxels provide precision and depth to reality, they tend to be large datasets and are difficult to manage, given the bandwidth of conventional computers. However, by efficiently compressing and processing large data files, interactive rendering can be enabled on consumer computers.

While voxels provide accuracy and depth to reality, they tend to be large datasets and are difficult to manage, given the bandwidth of conventional computers. However, by efficiently compressing and processing large data files, interactive rendering can be enabled on consumer computers.

Voxels are commonly used to create 3D images in medicine and represent terrain in games and simulations. Voxel terrain is used in place of a heightmap because of its ability to display ledges, caves, arches, and other 3D terrain features. These concave features cannot be represented on the heightmap because only the top "layer" of data is represented, and everything below it remains filled (volume that would otherwise be the interior of caves or the underside of arches or ledges).

Using voxels to construct 3D objects can be very beneficial in some situations. The peculiarity of voxel graphics is that the density of detail is uniform over the entire volume of the modeled object, which gives us a uniform data density. Voxel processing is more comfortable compared to polygon processing. Building a voxel representation of the environment from a point cloud is much easier than building a polygon representation. In the context of environmental recognition for a mobile robot, when receiving a point cloud at the input, it is easy to create voxels for further processing.

1.7. Octree

An octree is a tree in which each vertex has eight children. Octal trees are most often used to divide three-dimensional space by recursive partitioning into octants. Octrees are three-dimensional counterparts to quadtrees. Octrees are often used in 3D computer graphics and 3D game engines.

In the octant tree, each node divides the space into eight new octants. At a regional point in the octree, a node retains an exact three-dimensional point that is the “center” of the space division for that node. This point defines one of the corners of each of the eight child spaces.

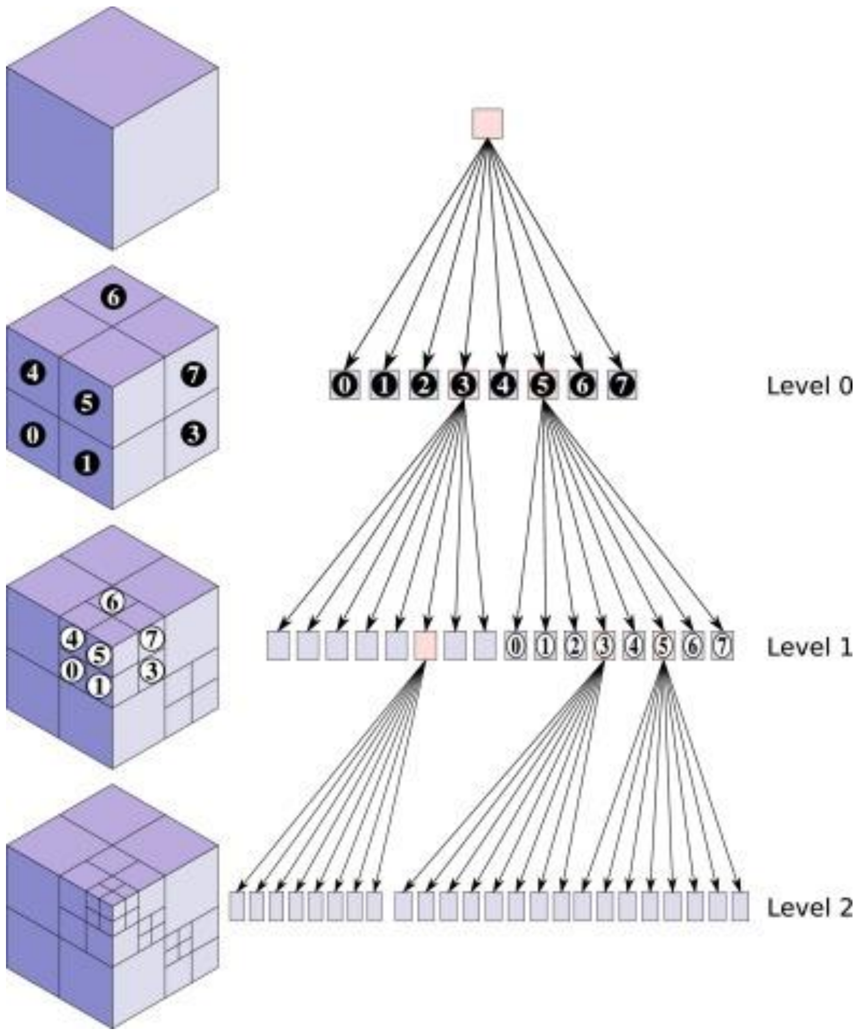


Fig. 1.8. Hierarchy of octants

Octrees are hierarchical tree structures that describe each region of 3D space as nodes. When compared with the necessary voxel representation, octrees reduce storage requirements for 3D objects. It also provides a convenient representation for storing information about object interiors. Octree encoding procedure is an extension of the quadtree encoding of 2D images.

The first node of the tree, the root, is a cube. Each node has either eight children or no children. The eight children form a 2x2x2 regular subdivision of the parent node. A node with children is called an internal node. A node without children is called a leaf.

Figure 1.9. shows an octree surrounding a 3D model where the nodes that have the bunny`s surface inside them have been refined, and empty nodes have been left as leaves.

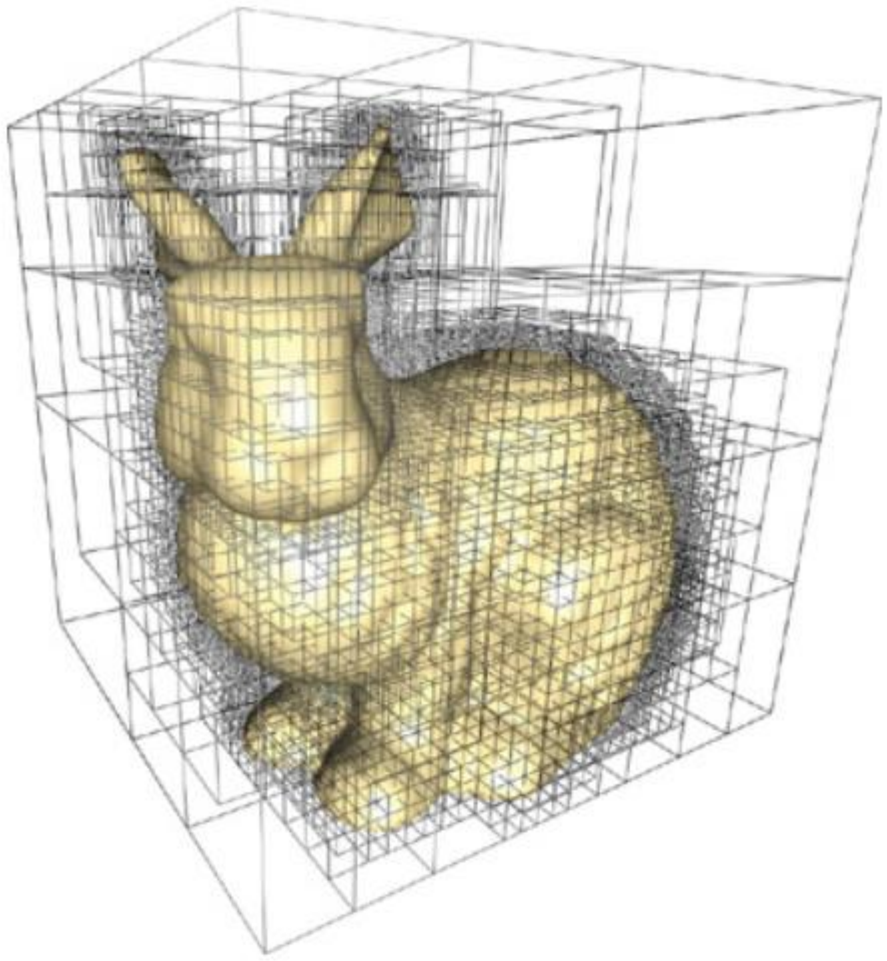


Fig. 1.9. An Octree Surrounding a 3D Model

Octrees are used to represent solid objects. They are particularly useful in applications that require cross-sectional views, for example, medical applications. Octrees are typically used when the interior of objects is important.

Each node is designated as:

– Black. If the cube is fully owned by the object.

– White. If the cube has no intersections with the object, that is, it belongs to the background.

– Gray. If the cube is partially owned by an object. In this case, the node has 8 descendants (octants), which are 8 cubes of the same size.

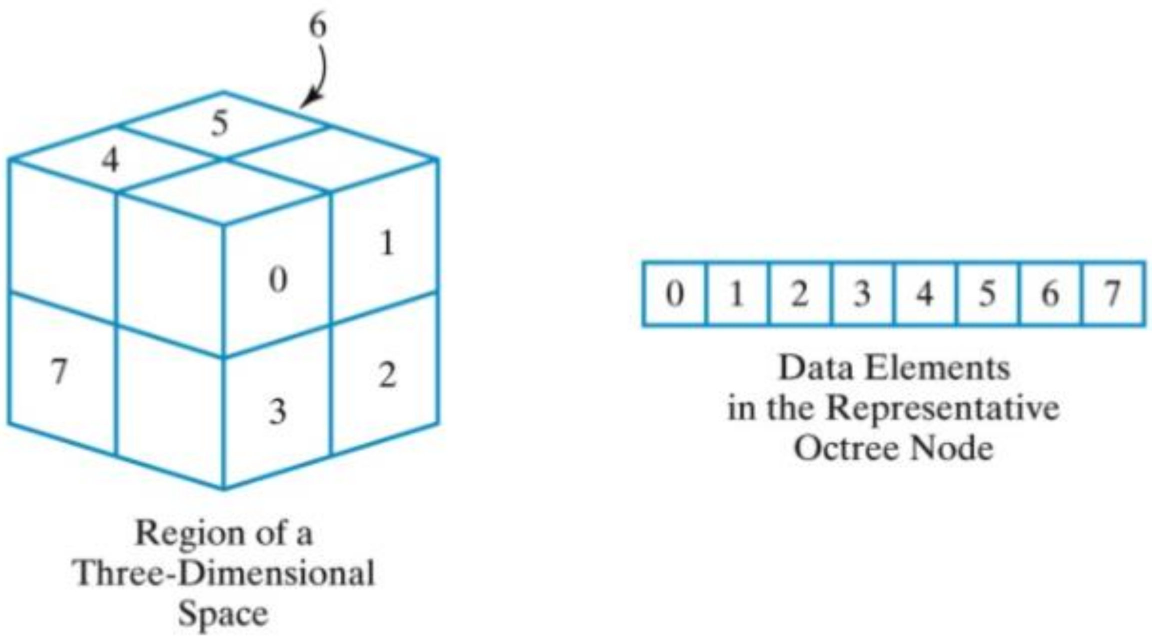

6
5
4
1
0
7
2
3
Region of a Three-Dimensional Space
0 1 2 3 4 5 6 7
Data Elements in the Representative Octree Node

Fig. 1.10. Data representation in memory

Octrees are based on a two-dimensional representation scheme called quadtree encoding, which divides a square region of space into four equal areas until homogeneous regions are found. These regions can be arranged in a tree.

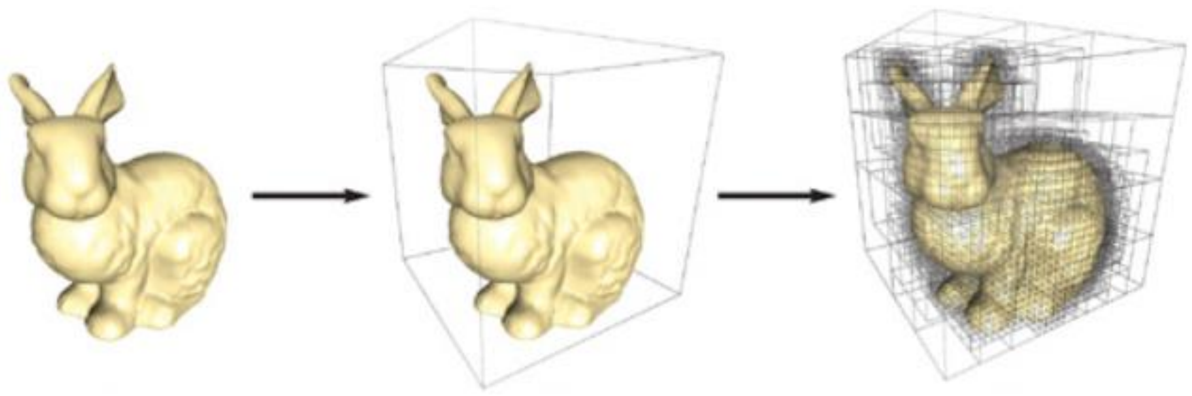


Fig. 1.11. Building an Octree Around a Mesh Surface

Quadtree encodings provide considerable savings in storage when large color areas exist in a region of space. An octree takes the same approach as quadtrees but divides a cube region of 3D space into octants. Each region within an octree is referred to as a volume element or voxel. The division is continued until homogeneous regions are discovered.

In 3 dimensions, regions can be considered homogeneous in terms of color, material type, density, or physical characteristics. Voxels also have the unique possibility of being empty.

Octrees cannot be considered k-trees because k-trees are split along the dimension, and octrees are split around a point. Moreover, k-trees are always binary, which is not valid for octrees.

Octrees are very generic data structures, widely used in computer science. They are a convenient way of storing information on unparameterized meshes, and more generally, in space.

Octrees can make the execution of queries faster and save on memory. Octrees can assist with modeling in many areas of earth science (such as geographical terrain, mineral deposits, and geological data), industrial modeling, robotics, pattern recognition, computer vision, and even medical imaging. Specifically, octrees can be used for spatial indexing, collision detection, and view frustum culling [3].

1.8. Conclusions to the first section

The first theoretical section is devoted to the analysis of mobile robots and localization. The basic concepts and algorithms of navigation, existing sensors, methods of data processing and storage were considered.

Talking about the navigation of mobile robots, there are three main types of systems - global, local and personal. The global type defines the absolute coordinates of the robot when moving over large areas and objects. Local navigation fixes coordinate following the specified parameters. For this type of navigation, the zone limits must already be known. Personal navigation implies positioning, considering objects that are nearby. Typically, the presented modification of the mobile robot is equipped with a special manipulator.

It is believed that the larger the device, the higher the importance of global navigation for it and the lower the personal one.

A mobile robot is an “intelligent” self-propelled mechanism. For it to be able to provide meaningful behavior and perform functionally complete work with the help of its “intellectual” capabilities, it must be equipped with various sensory systems that allow the robot to perceive the environment and navigate in it. As with humans and most animals, the most important and significant of these systems in the robot's movement is the visual sensor.

Point cloud is a collection of separate, unrelated points with a specific position and color. This makes it easy to edit, display, and filter point clouds.

A voxel is a single sample or data point on an evenly spaced 3D grid. This data point can consist of a single piece of data, such as opacity, or multiple pieces of data, such as color, in addition to opacity. The voxel represents only one point on this grid, not a volume; the space between each voxel is not represented in the voxel dataset. Depending on the type of data and the intended use of the dataset, this missing information can be reconstructed and/or approximated, for example, using interpolation.

Octrees are hierarchical tree structures that describe each region of 3D space as nodes. When compared with the necessary voxel representation, octrees reduce storage requirements for 3D objects. It also provides a convenient representation for storing information about object interiors. Octree encoding procedure is an extension of the quadtree encoding of 2D images.

SECTION 2
DEVELOPMENT OF THE PATH PLANNING ALGORITHM

The main structure of this solution is shown in Fig. 2.1.

Where the first step is to obtain a cloud of raw points; since the cloud is too dense, processing will be slow. There is also no need to use such a large number of points. Therefore, using the PCL library, the resolution of row data has been reduced. Moreover, using PCL, redundant objects have been removed, such as robot parts, floor, and ceiling.

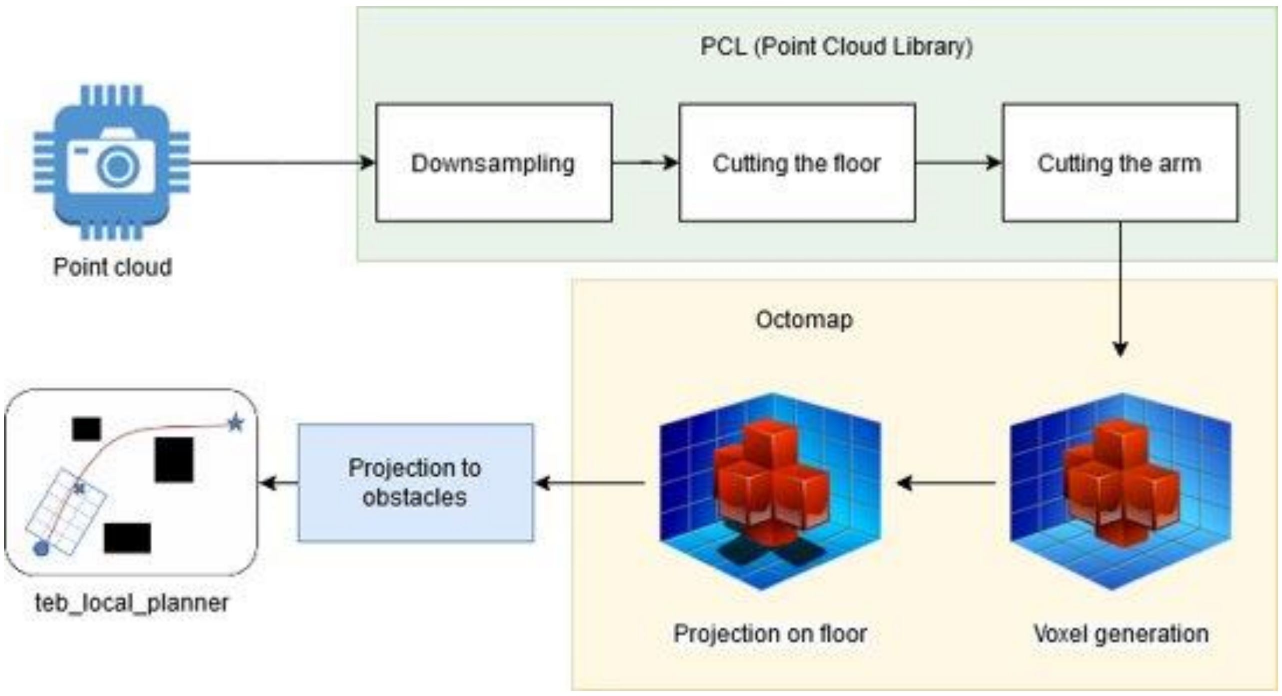


Fig. 2.1. Octree navigation pipeline

After the above processing, the point cloud is sent to Octomap, where an octree is created from it, and an octree is projected onto the floor.

Further, using the developed software, the projection is converted into obstacles suitable for sending to teb_local_planner. In this section, the above structure will be described in more detail along with a description of the hardware and software platform.

2.1. Robot

The research was carried out on the mobile robot “Leonart” of the RT Lions team at Reutlingen University. The mobile robot was created to educate students, with practical purposes, and for participation in RoboCup competitions.

The base of a mobile robot is the mobile platform MPO-700 from Neobotix. This platform is presented in Fig. 2.2. It is an autonomous robot vehicle for a wide range of applications. The robot starts after turning clockwise. Its four Omni-Drive-Modules enable it to move exceptionally smoothly in any direction. This robot is even capable of rotating freely while driving to its destination.

The robot has fully omnidirectional maneuverability, very steady movements, high stability, and payload. Moreover, it is compact and has easily integrated drive units. This makes the MPO-700 a high-performance alternative for applications that require omnidirectional movements without the limitations of traditional kinematics.

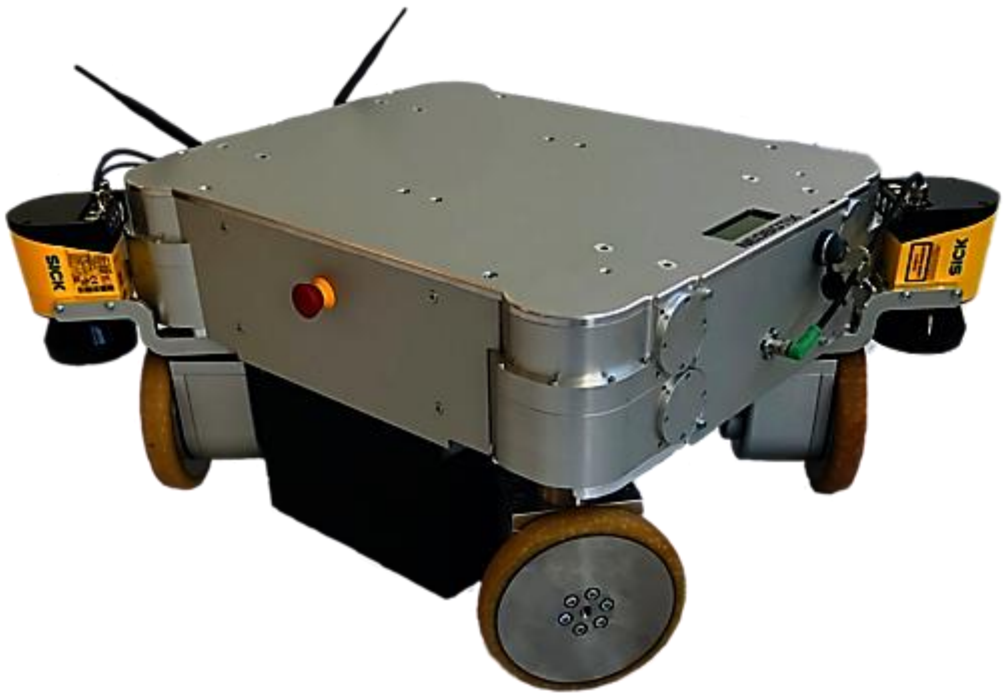


Fig. 2.2. Neobotix MPO-700

Neobotix platform MPO-700 has an emergency stop button; when this button is pressed, the robot is immediately set to an emergency stop. All drives are disconnected from the power supply, and the fall-safe brakes are engaged. This state can be reset by unlocking the emergency stop buttons and turning the key switch clockwise to position II for a few seconds.

LC-Display indicated the current state of the robot. It shows different indicators such as status information, messages, battery charge level, temperature, uptime since startup. LC-Display also presents information about the version, status, and serial number.

The mobile robot "Leonart", which is presented in Fig. 2.3 and Fig. 2.4, has three computers. One of them is from Neobotics, the second one for the "Leonart" and the third one is for the Sawyer robot from Rethink Robotics. The Sawyer from Rethink Robotics is a high-performance collaborative robot that performs automation tasks where traditional industrial robots reach their limits. Computers have Ubuntu system and ROS Kinetic.

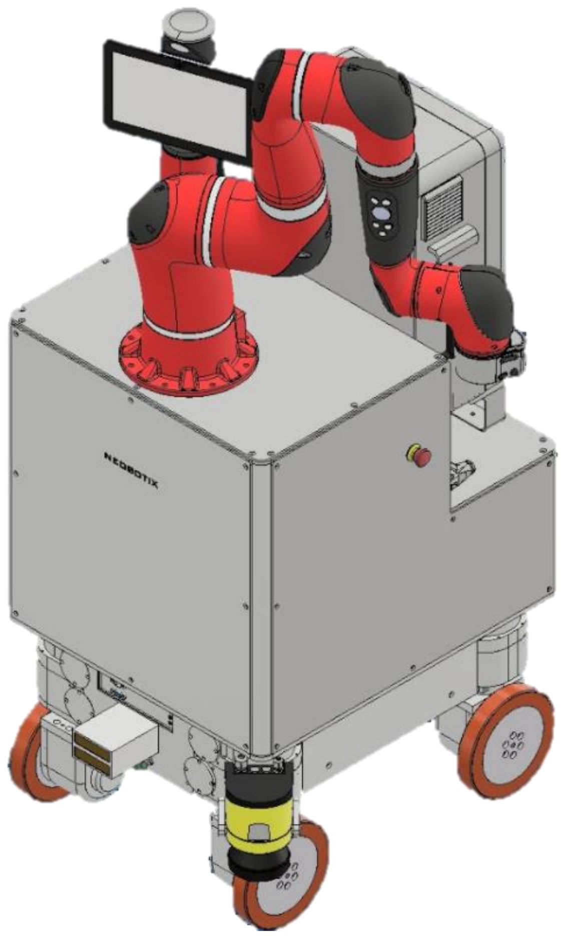


Fig. 2.3. Mobile robot "Leonart" (first view)

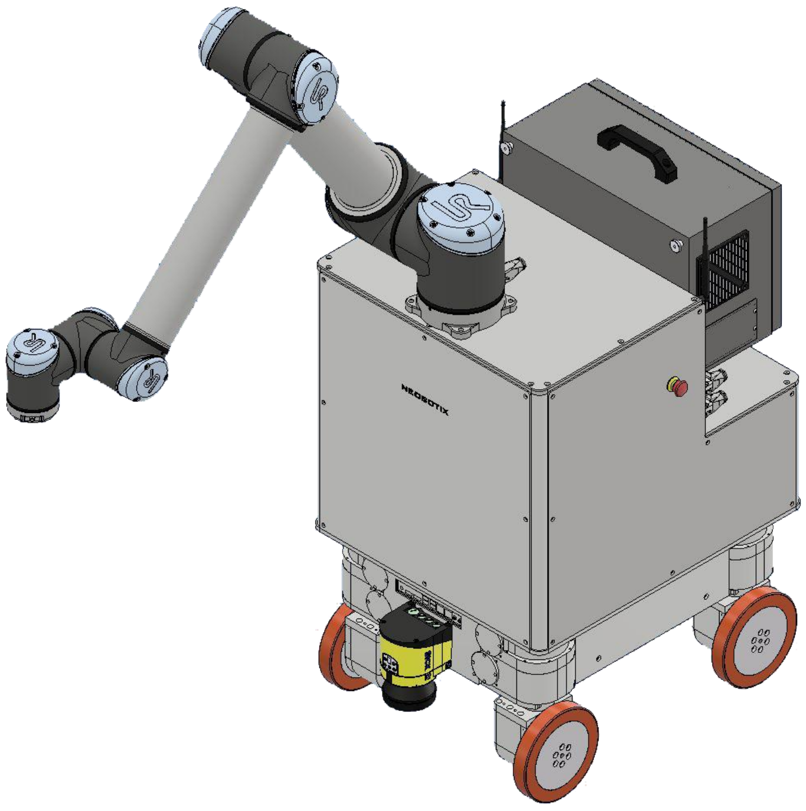


Fig. 2.4. Mobile robot “Leonart” (second view)

All peripheral connections of the on-board computer are accessible at the front of the platform. The battery connector is the connection between the buttery and the middle robot. The charging connector provides direct, not fuse protected access to the battery. The battery charger can be plugged in there. Charging contacts can be connected to the battery via a high-power relay if the MPO-700 has been prepared to use the automatic charging station.

2.2. Lasers

A laser is a device that emits light through a process of optical amplification based on the stimulated emission of electromagnetic radiation. The term “laser” originated as an acronym for “light amplification by stimulated emission of radiation”.

A laser differs from other sources of light in that it emits coherent light. Spatial coherence allows a laser to be focused to a tight spot, enabling applications such as laser cutting and lithography.

Fig. 2.5. Laser SICK S300

The mobile robot has S300 laser model from SICK. Measurement data from one or two laser scanners can be used for localization, navigation, and collision avoidance. The safety-approved laser scanner S300 can monitor user-defined safety fields around the robot, which can be dynamically activated by application-specific control software. As soon as an object is detected within the currently activated field, the robot is immediately set to an emergency stop. This control software is responsible for the correct activation of the safety fields according to the current condition.

The S300 Standard is a cost-effective solution for applications with freely definable protective fields. The triple field mode allows one protective field and two warning fields to be activated simultaneously.

As soon as a person or obstacle is detected within the currently active safety field, the robot is immediately set to an emergency stop. A scanning angle of 270° allows complete protection with only two scanners. The correct protective field of the laser at any speed avoids unnecessary stops. The stop will be reset automatically after the field has been cleared. No manual reset is required in this case.

2.3. Realsense Camera

Depth cameras are cameras that shoot video, in each pixel of which not a color is stored, but the distance to an object at that point. Such cameras have existed for more than 20 years, but in recent years the speed of their development has increased many times and we can already talk about a multi-vector revolution.

The depth of field of the imaged space is the range of distances in the image in which objects are perceived as sharp. Depth of field varies with camera type, aperture size, and focusing distance, although print size and viewing distance can alter our perception of depth of field.

Computer stereo vision is the extraction of three-dimensional information from digital images, such as those obtained with a CCD camera. By comparing scene information from two viewpoints, you can extract 3D information by examining the relative position of objects in the two panes.

The Time of Flight (ToF) is one method with which works the depth camera, radiating Infrared Eays (IR) and measuring the distance by the time it takes to go back to the sensor. The IR transmission unit and the setting unit are a pair, and the distance measured by each pixel is read. This method represents the most expensive one due to the sophisticated hardware needed.

A stereo camera, which is considered a Depth Camera, is the last and the most popular method for depth estimation. Their idea is based on the operation mode on which work the left and right eyes of the people. A stereo camera is a type of camera with two or more lenses with a separate image sensor or film frame for each lens. This allows the camera to simulate the binocular vision of a person and therefore enables it to capture three-dimensional images, a process known as stereo photography. Stereo cameras can be used to create stereo visual and 3D images for movies or long-range imaging.

The Intel RealSense Depth Camera D435 is used in the mobile robot “Leonart”. This camera is shown on Fig. 2.6.

Fig. 2.6. Intel RealSense Depth Camera D435

The Intel RealSense Depth Camera D435 Series uses stereo vision to calculate depth. They are fully calibrated and produce hardware-rectified pairs of images. This camera performs all depth calculations at up to 90 FPS and offers sub-pixel accuracy and a high fill-rate. It has the inertial measurement unit (IMU), which is used for the detection of movements and rotations.

An IMU combines a variety of sensors with gyroscopes to detect both rotation and movement in 3 axes, as well as pitch and roll. It allows refining depth awareness in any situation where the camera moves. Thus, it is widely used for SLAM and tracking allowing better point cloud alignment.

2.4. ROS

The Robot Operating System (ROS) – is a widely used robot software development framework with distributed teamwork capabilities. The way of thinking of ROS is to create software that works with different robots, with only minor changes to the code. This system allows the parallel launch of several programs, as well as support the exchange of information between them.

ROS was developed in 2007 at the Stanford Artificial Intelligence Laboratory (SAIL) to support the Stanford AI Robot project. Since 2008, development has continued mainly at the Willow Garage research institute, collaborating with over twenty different institutes in a co-development model.

The atomic unit of the environment is a package, which must solve a particular issue. According to the ideology, ROS packages can be improved by any developer, and new packages can be added to the general knowledge base with a description of instructions for use and ROS composite modules. In this way, unique functions can be created that can be used by various robots without much effort to transfer information.

ROS has standard operating system features such as hardware abstraction, low-level device management, commonly used functionality, interposes message passing, and library management. The ROS architecture is not based on a graph with a centralized topology. Processing takes place in nodes that can receive or send data from sensors, condition monitoring, and planning systems, drives, and so on. The library is focused on Unix-like systems (works great under Ubuntu Linux, and Fedora and Mac OS X are experimental).

A wide range of libraries and tools are provided by ROS to help software developers to create inventive applications in a formed arrangement. ROS also supports many different sensors and actuators used in robotics. New devices that are compatible with this framework appear every day.

The opportunity to get useful fundamental highlights is the welfare of using ROS. Different drivers, libraries, visualizers, packages, tools, and simulations are provided to diminish the time and intricacy of development. Many research institutes have started to develop their projects in ROS, adding support for their hardware and sharing examples of their code. Several robot companies have begun adapting their products for use with ROS.

ROS is a language-independent architecture that associates with various programming languages (C, C++, Java, Python, and so forth). Such scripting languages as Ruby or Python, handily contrasted with others and simplify the achievement of numerous product errands.

There are relatively few robot frameworks out there that uphold the development of broadly useful robot programs across numerous platforms. It makes it harder to achieve even basic and trifling undertakings that may appear to be not difficult for people. However, the Robot developer ought to explore and figure out numerous varieties between instances of assignments and environments.

The *-ros-pkg package is a general repository for high-level library development. Many of the features often associated with ROS, such as the navigation libraries and the RViz renderer, are stored in this repository. These libraries provide a powerful set of tools (various visualizers, simulators, debugging tools) to simplify your work.

ROS supports parallel computing, has good integration with popular C ++ libraries such as OpenCV, Qt, Point Cloud Library, etc., and it can run on single-board computers such as the Raspberry Pi or BeagleBone Black, as well as microcontroller platforms such as Arduino. You can create your own Arduino or Raspberry Pi based robot and use the Robot Operating System to control it.

ROS is free and open-source software, which in turn allows anyone to work with existing packages and create their own, share and work with packages from other developers. All these factors have played a significant role in the popularization of this framework.

Three main levels of concepts in ROS are Filesystem level, Computation Graph, and Community level.

Filesystem level concepts mainly cover ROS resources. The primary objective of the ROS Filesystem is to unify the build process of a project while simultaneously give enough adaptability and tooling to decentralize its conditions.

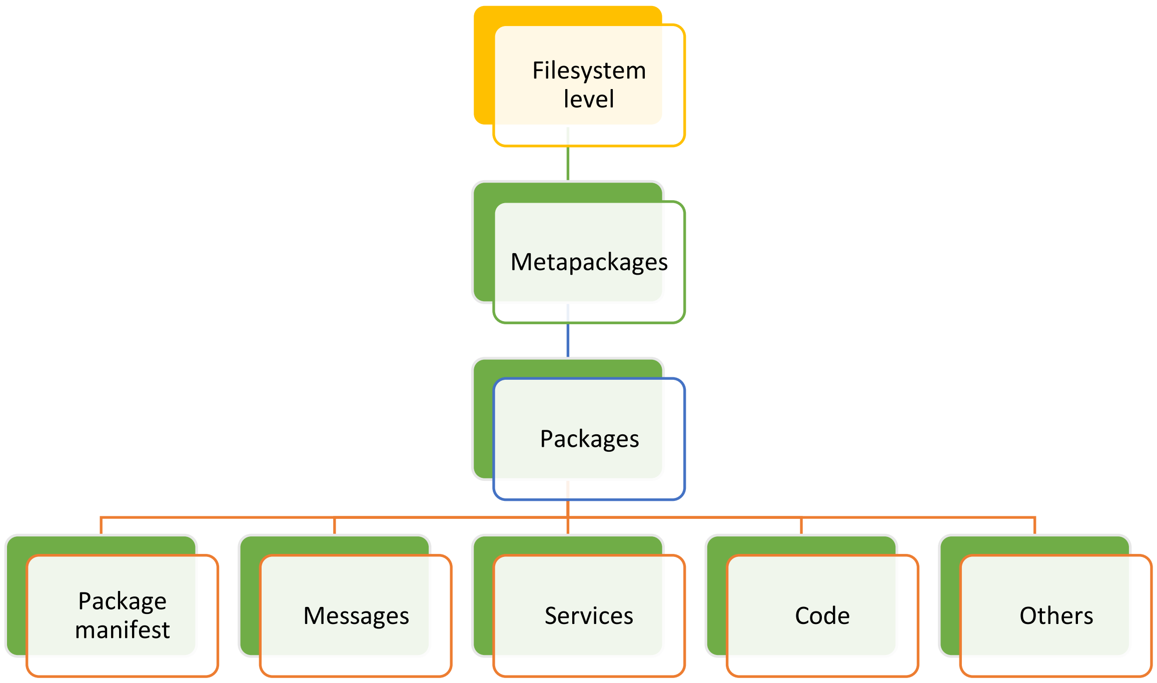


Fig. 2.7. ROS Filesystem level

Resources from ROS Filesystem level concepts are packages, metapackages, repositories, messages, and service types. Packages are the lowest level of software organization and the central unit for organizing software in ROS. Each package contains libraries, executables, scripts, etc. Each package has its own manifest, which contains a brief description of the package, dependencies between packages, and various meta information such as package version, license, etc. The package.xml file is the manifest of the package. Metapackages are virtual packages; they do not contain any source. The metapackage manifest might include packages inside it as runtime dependencies and declare an export tag.

ROS messages of different types are sent from one ROS process to the other. A custom message can be defined in the msg folder. ROS services implement these request-response type of communications. They consist of two message types: One for requesting data, and one for the response. These services are the gateway to event-based ROS executions. The reply and request data types are defined in the srv folder inside the package.

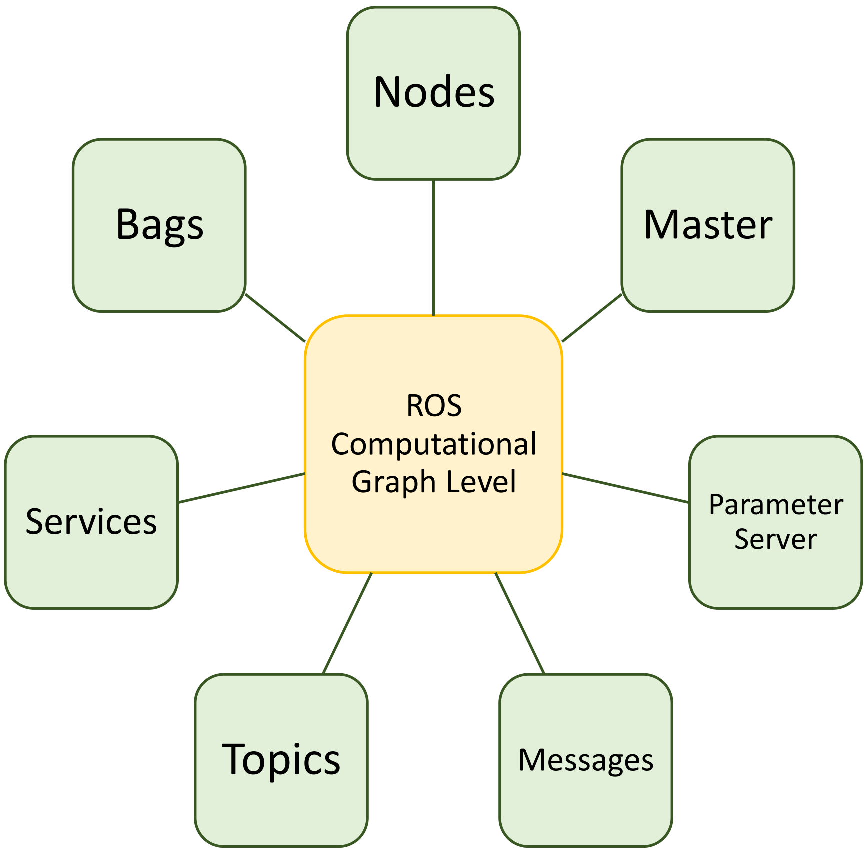


Fig. 2.8. ROS Computational Graph level

The Computation Graph level consists of the following main concepts: ROS Nodes, Master, Parameter server, Messages, Topics, Services, and Bags. Each concept in the graph is contributed to this graph in different ways. The main element of the ROS system is the node. This element corresponds to a set of executable files belonging to one program. The ROS architecture is based on a directed graph: processing occurs at its vertices - nodes that can receive or send data from sensors, actuators, condition monitoring, and planning systems through message transmission channels, which are the edges of the graph. This interaction between each other is done through topics through specific messages. Each process can publish its own topics by sending messages with data to them, as well as subscribe to other topics, subscribe to messages from other processes. In turn, each topic supports only one type of transmitted messages. Several nodes designed to perform one task can be combined into a package.

The ROS Master performs as a nameservice in the ROS Computation Graph and stores topics and services registration information for ROS nodes. Nodes communicate with the Master to report their registration data. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run.

Nodes connect to other nodes directly. Nodes that subscribe to a topic will request connections from nodes that publish that topic and will establish that connection over an agreed upon connection protocol.

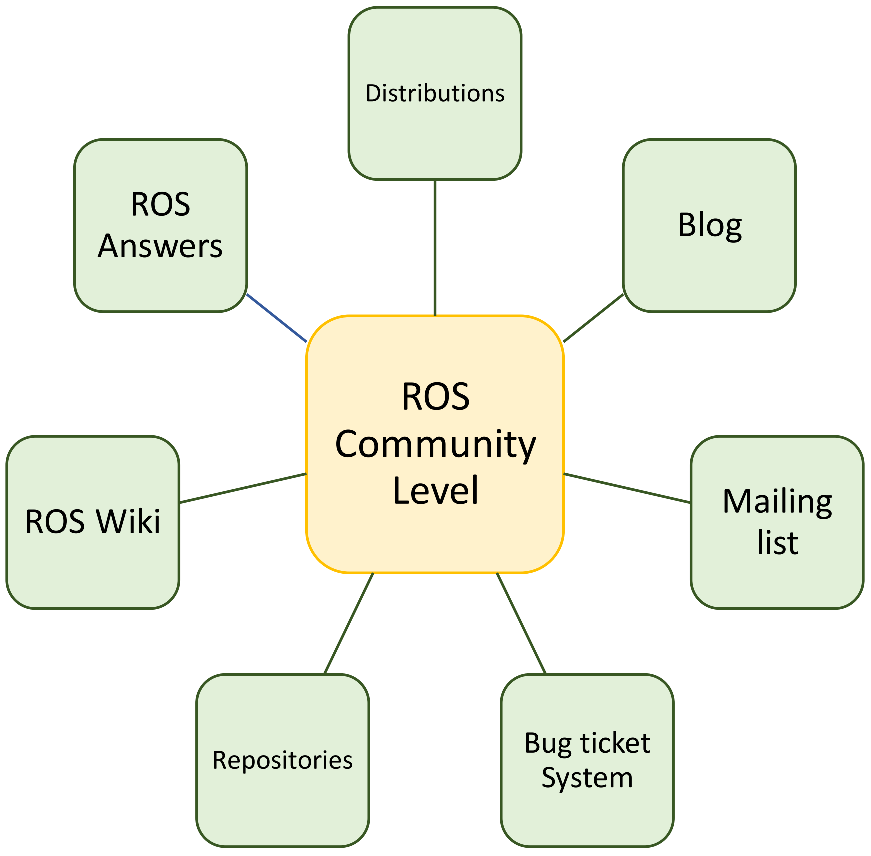


Fig. 2.9. ROS Community Level

The Community level consists of the next concepts: Distributions, Repositories, The ROS Wiki, Bug ticket system, Mailing lists, ROS Answers, and Blog. These ROS resources enable an opportunity for the community to exchange software and knowledge.

2.4.1. ROS Tools and Simulators

A variety of tools augments ROS's core functionality. Applying tools hugely increases the capacity of systems, which are using ROS. They simplify and provide solutions to common robotics development problems. Tools are in packages and provide tasks and robot-agnostic tools which come with the core of most modern ROS installations.

Catkin is a collection of CMake macros and associated code used to build packages used in ROS. Catkin is the official ROS build system and the successor to the original ROS build system, rosbuild.

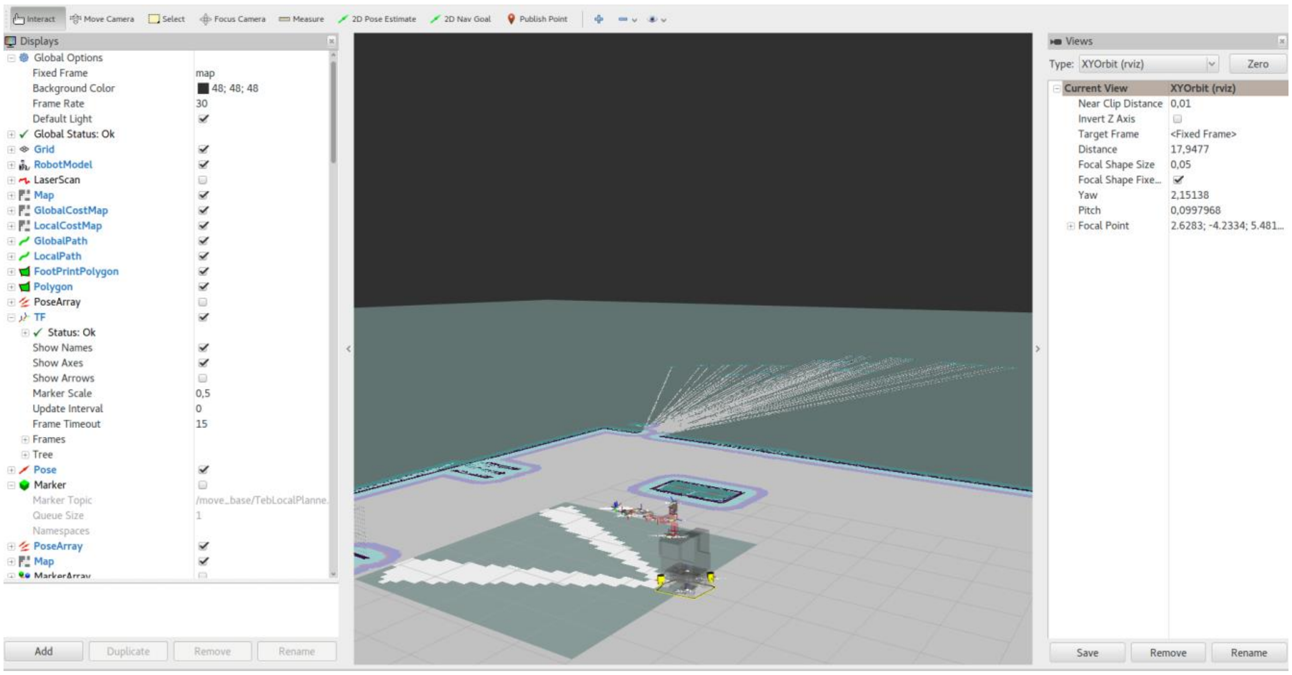
Roslaunch is a tool used to launch multiple ROS nodes both locally and remotely, as well as setting parameters on the ROS parameter server. Roslaunch configuration files, which are written using XML, can easily automate a complex startup and configuration process into a single command. Roslaunch scripts can include other roslaunch scripts, launch nodes on specific machines, and even restart processes that die during execution.

2.4.2. RViz

RViz is a ROS graphical interface that allows users to visualize in real-time on a 3D scene all the components of a robotic system - coordinate systems, moving parts, sensor readings, images from cameras, using plugins for many kinds of available topics.

RViz is the primary visualizer in ROS and an incredibly useful tool for debugging robotics. RViz is a tool for 3D visualization of data (messages) coming in on ROS topics. By visualizing data, developers can evaluate how the robot "sees" its environment and how it perceives itself in it - its position, orientation in space, and location on the map. RViz assists in debugging ROS software components and is an indispensable tool when working with a robot in both real environment and simulation.

The graphical interface of the software is shown in Fig. 2.10.



Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
Global Status: Ok	
Grid	✓
RobotModel	✓
LaserScan	☐
Map	✓
GlobalCostMap	✓
LocalCostMap	✓
GlobalPath	✓
LocalPath	✓
FootPrintPolygon	✓
Polygon	✓
PoseArray	☐
TF	✓
Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	☐
Marker Scale	0,5
Update Interval	0
Frame Timeout	15
Frames	
Tree	
Pose	✓
Marker	☐
Marker Topic	/move_base/TebLocalPlanne…
Queue Size	1
Namespaces	
PoseArray	✓
Map	✓
MarkerArray	☐

Type: XYOrbit (rviz)

Current View	XYOrbit (rviz)
Near Clip Distance	0,01
Invert Z Axis	☐
Target Frame	<Fixed Frame>
Distance	17,9477
Focal Shape Size	0,05
Focal Shape Fixe…	✓
Yaw	2,15138
Pitch	0,0997968
Focal Point	2.6283; -4.2334; 5.481…

Fig. 2.10. RViz

When RViz starts for the first time, you will see an empty window. A display is something that draws something in the 3D world and likely has some options available in the displays list. An example is a point cloud, the robot state, etc. Each display gets its own list of properties.

Each display gets its own status to help let you know if everything is OK or not. The status can be one of 4: OK, Warning, Error, and Disabled. The status is indicated in the display’s title by the background color. The Status category also expands to show specific status information. This information is different for different displays, and the messages should be self-explanatory.

A configuration contains:

– Displays + their properties.

– Tool properties.

– Camera type + settings for the initial viewpoint.

Different configurations of displays are often useful for different uses of the visualizer. To this end, the visualizer lets you load and save different configurations.

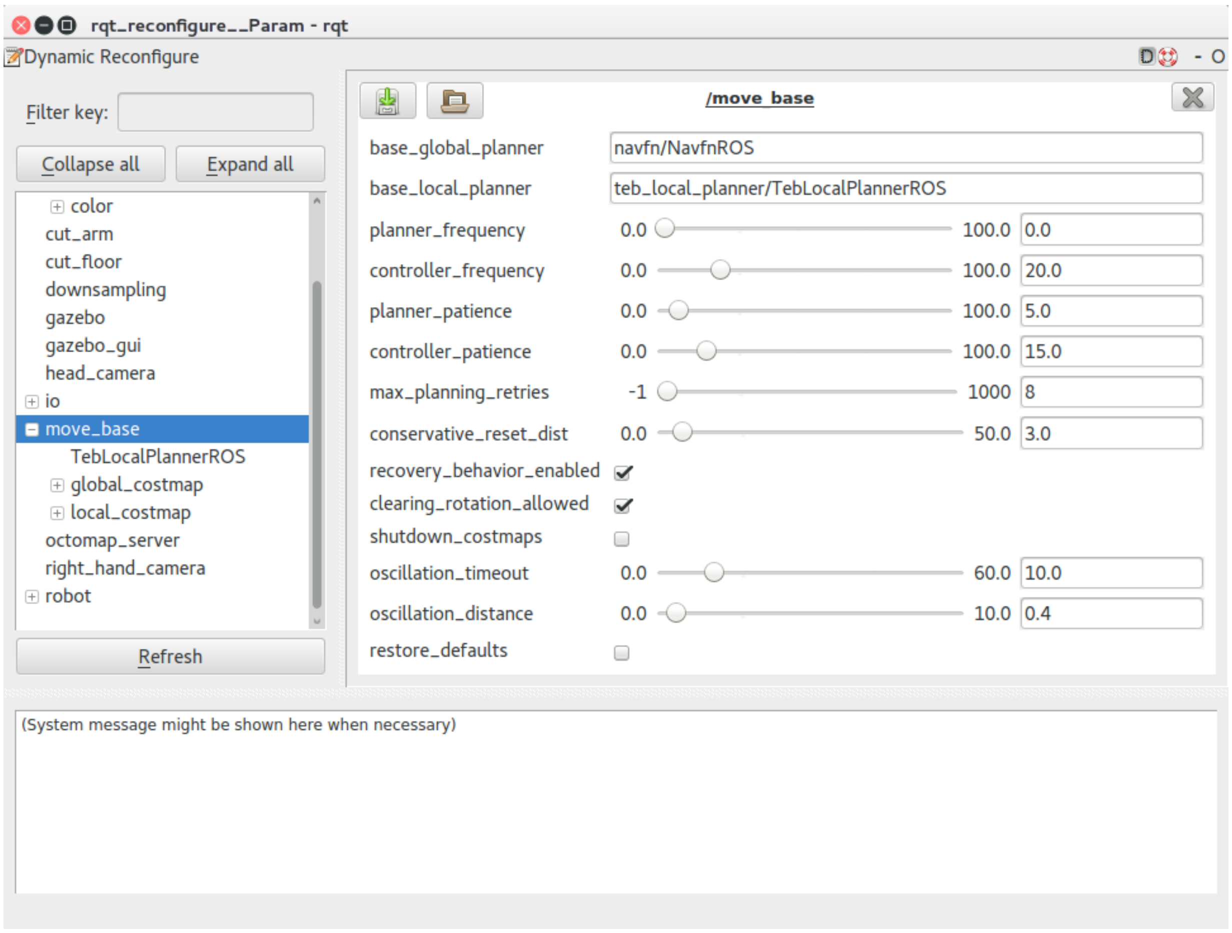
The views panel also lets you create different named views, which are saved and can be switched between. A view consists of a target frame, camera type and camera pose. A view consists of:

– View controller type.

– View configuration (position, orientation, etc., Possibly different for each view controller type).

– The Target Frame.

2.4.3. ROS GUI Development Tool (Rqt_reconfigure)



color
cut_arm
cut_floor
downsampling
gazebo
gazebo_gui
head_camera
io
move_base
TebLocalPlannerROS
global_costmap
local_costmap
octomap_server
right_hand_camera
robot

base_global_planner	navfn/NavfnROS		
base_local_planner	teb_local_planner/TebLocalPlannerROS		
planner_frequency	0.0	100.0	0.0
controller_frequency	0.0	100.0	20.0
planner_patience	0.0	100.0	5.0
controller_patience	0.0	100.0	15.0
max_planning_retries	-1	1000	8
conservative_reset_dist	0.0	50.0	3.0
recovery_behavior_enabled	✓		
clearing_rotation_allowed	✓		
shutdown_costmaps	☐		
oscillation_timeout	0.0	60.0	10.0
oscillation_distance	0.0	10.0	0.4
restore_defaults	☐		

(System message might be shown here when necessary)

Fig. 2.11. Rqt_reconfigure GUI

The qt_reconfigure plugin provides a means to update parameters at runtime without having to restart the node. It replaces former dynamic_reconfigure`s GUI and provides the way to view and edit parameters that are accessible via dynamic_reconfigure.

The next command is used to launch rqt_reconfigure GUI: rosrun rqt_reconfigure rqt_reconfigure. The user can select one or multiple nodes from the left part to reconfigure them.

From the Rqt_reconfigure GUI it is possible to select one or multiple of the left-hand nodes to reconfigure it.

2.4.4. Gazebo

Gazebo is an open-source dynamic 3D simulator that is being developed by the Open-Source Robotic Foundation and has been quite used with ROS. The gazebo allows you to accurately and efficiently model robots both in difficult indoor conditions and outside.

Gazebo is a 3D simulator that provides robots, sensors, environment models for 3D simulation required for robot development and offers realistic simulation with its physics engine. Gazebo is one of the most popular simulators for open-source robotics in recent years and has been widely used in the field of robotics because of its high performance and reliability.

Gazebo uses OGRE (Open-source Graphics Rendering Engines) for the 3D Graphics, not only for the robot model but also for the light, that can be realistically drawn on the screen.

Gazebo uses a distributed architecture with separate libraries for physics simulation, rendering, user interface, communication, and sensor generation. The simulator consists of a server gzserver, which deals with the calculation of physics, collisions, and sensor simulation. Clients such as gzclient (for desktop) and gzweb (for desktop) can connect to the server. They are the ones who render the models.

The communication library currently uses the open-source Google Protobuf for the message serialization and boost::ASIO for the transport mechanism. It supports the publish/subscribe communication paradigm. For example, a simulated world publishes body pose updates, and sensor generation and GUI will consume these messages to produce output.

A lot of sensors are already supported Laser range finder (LRF), 2D/3D camera, depth camera, a contact sensor, force-torque sensor; noise can be considered as added to the sensor data like in real environment.

Some robot models are already available in gazebo: PR2, Pioneer2 DX, iRobot Create, and TurtleBot are already supported in the form of SDF, a Gazebo model file, and users can add their own robots with an SDF file.

This mechanism allows for introspection of a running simulation and provides a convenient mechanism to control aspects of Gazebo.

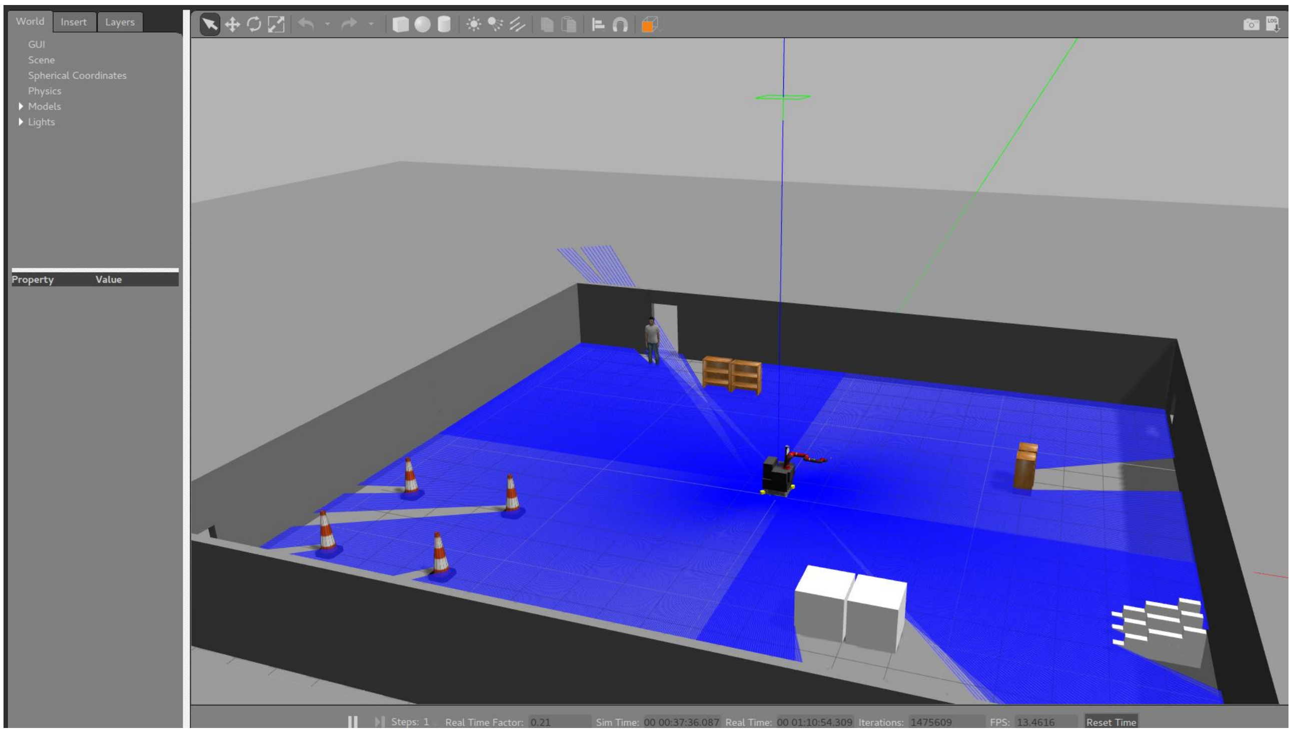


Fig. 2.12. Gazebo simulator

All this makes it possible to test complex robotic systems in virtual space much faster and without the risk of inflicting damage on real expensive robots. Gazebo is included in the complete ROS installation package, so nothing extra is needed.

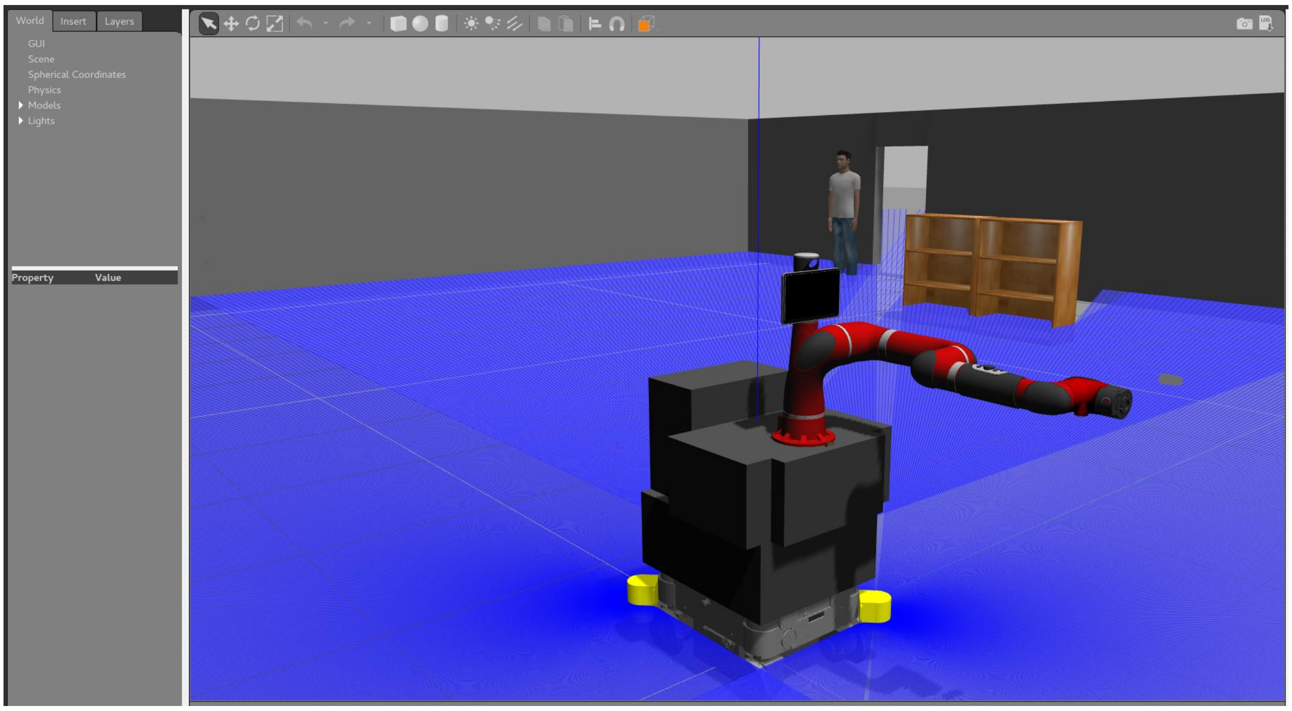


Fig. 2.13. Mobile robot “Leonart” in Gazebo

Gazebo uses a number of environment variables to locate files and set up communications between the server and clients. Default values that work for most cases are compiled in. The server is the workhorse of Gazebo. It parses a world description file given on the command line and then simulates the world using a physics and sensor engine. Plugins specified on the command line are loaded first, then plugins specified in the SDF files are loaded. Some plugins are loaded by the server, such as plugins that affect physics properties, while the graphical client loads other plugins to facilitate custom GUI generation.

2.4.5. TF package

The Transform Library TF consists of tf package, which is used within ROS nodes. The tf library was designed to provide a standard way to keep track of coordinate frames and transform data within an entire system such that individual component users can be confident that the data is in the coordinate frame that they want without requiring knowledge of all the coordinate frames in the system.

The tf library has two standard modules, a Broadcaster, and Listener. These two modules are designed to integrate inside ROS, but generally useful outside of ROS, too [13].

The tf package is used for the optimal interaction between the various sensors/components of a robot. TF is a package that lets the user keep track of multiple coordinate frames over time. TF maintains the relationship between coordinate frames in a tree structure buffered in time and lets the user transform points, vectors, etc., between any two coordinate frames at any desired point in time.

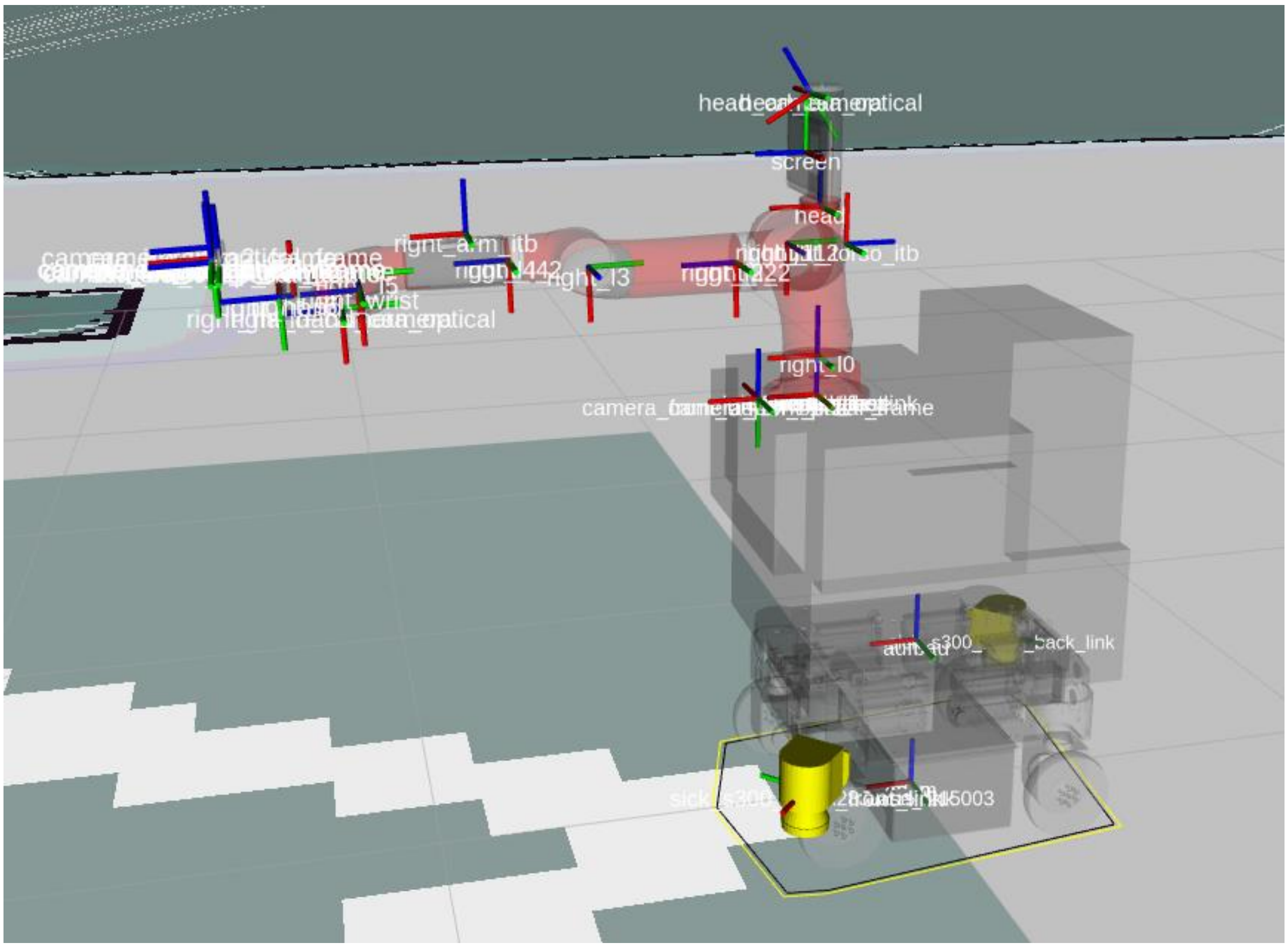


Fig. 2.14. TF origins of mobile robot parts

In order to avoid errors in the context, a system always consists of a base (Leonart: in the middle of the lower base). Thereupon, e.g., the camera attached to the robot. The library now ensures that the camera is permanently in the correct relationship to the robot as soon as the robot moves (in the simulation). This is particularly important for path planning so that the object recognized by the camera is transmitted with the correct relationship to the robot.

2.5. ROS Navigation Stack

Below the main components and links between navigation elements will be presented.

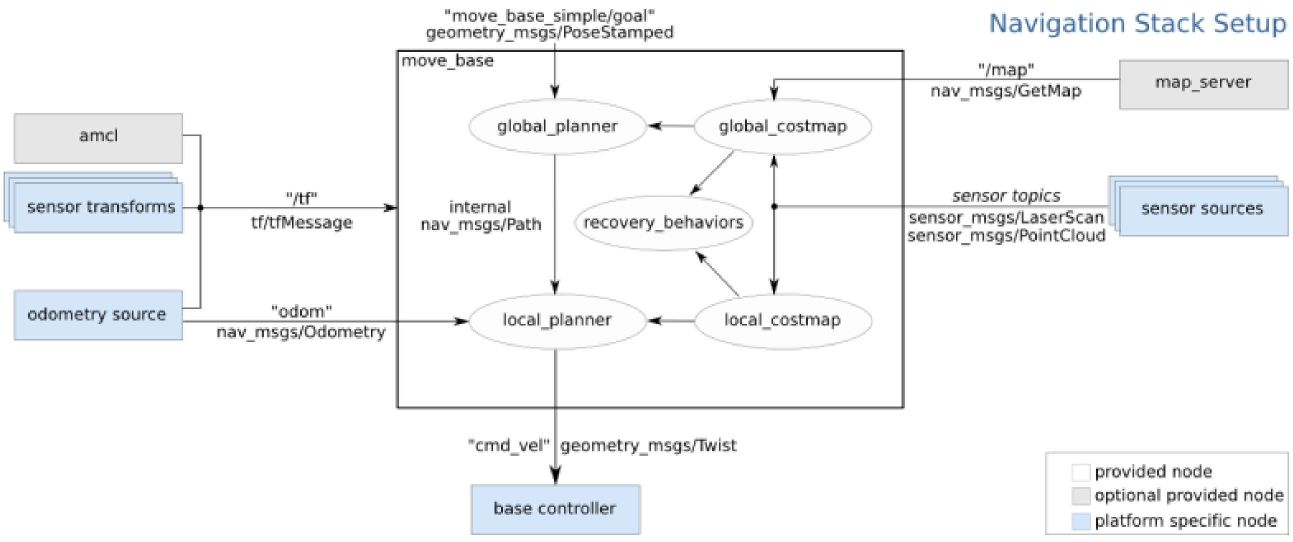


Fig. 2.15. ROS Navigation Stack

AMCL is a probabilistic localization system for a robot moving in 2D. This system implements the adaptive Monte Carlo localization approach (as described by Dieter Fox [18]), which uses a particle filter to track a robot's pose against a known map.

AMCL tries to match the laser scans to the map, thus detecting if there is any drift occurring in the pose estimate based on the odometry (dead reckoning). This drift is then compensated by publishing a transform between the map frame and the Odom frame such that at the end, the transform map->base_frame corresponds to the real pose of the robot in the world.

Gmapping package contains a ROS wrapper for OpenSlam’s Gmapping. The gmapping package provides laser-based SLAM, as a ROS node called slam_gmapping.

Using slam_gmapping, it is possible to create a 2-D occupancy grid map (like a building floorplan) from the laser and pose data collected by a mobile robot.

2.5.1. Move_base node

This project uses the capabilities of ROS, which offers a solution to the problem using the move_base package [15]. Further in the paper, the properties and organization of these management packs are considered.

The move_base node provides a ROS interface for configuring, running, and interacting with the navigation stack on a robot. A high-level view of the move_base node and its interaction with other components is shown above. The blue vary based on the robot platform, the gray are optional but are provided for all systems, and the white nodes are required but also provided for all systems [15].

The components of this configuration will be discussed below.

Map (map_server). The map_server node feeds some RO map to the input; this map will be used to build a global_costmap - a global obstacle map, according to which the algorithm will then decide at which points to plot the route.

Odometry source. It is necessary that the tf module can get the transformation from odom to base_link, where base_link is the coordinate system associated with the robot, and odom is the coordinate system in which the user agent moves.

Mobile robot coordinates (amcl, sensor transforms). It is necessary that using some algorithm (for example, the adaptive Monte Carlo method - amcl), which through tf provides a group of sequential coordinate transformations: map → odom → base_link.

Sensor data (sensor sources), which will be used to build local_costmap, which will be used by the program to ensure the movement along the route.

For any global or local planner to be used by the move base package, they have to ad- here to some interfaces defined in the nav core package. The global planner must adhere to the nav core::BaseGlobalPlanner interface and the local planner to the nav core::BaseLocalPlanner interface; also, they must be added as plugins to ROS in order to work with the navigation stack.

Global_planner. There is an interface nav_core::BaseGlobalPlanner, which is used by path planners. One such scheduler is navfn. It uses Dijkstra’s algorithm to calculate the minimum path from the start point to the endpoint.

Local_planner. nav_core::BaseLocalPlanner provides the interface used by local schedulers. base_local_planner is one of the plugins that use this interface. This package provides implementations of the TRA (Trajectory Rollout approach) and DWA (Dynamic Window approach) methods for local robot navigation on the plane. Taking a route and an obstacle map (costmap) as input, it outputs the output speed, which is then transmitted to the robot controller.

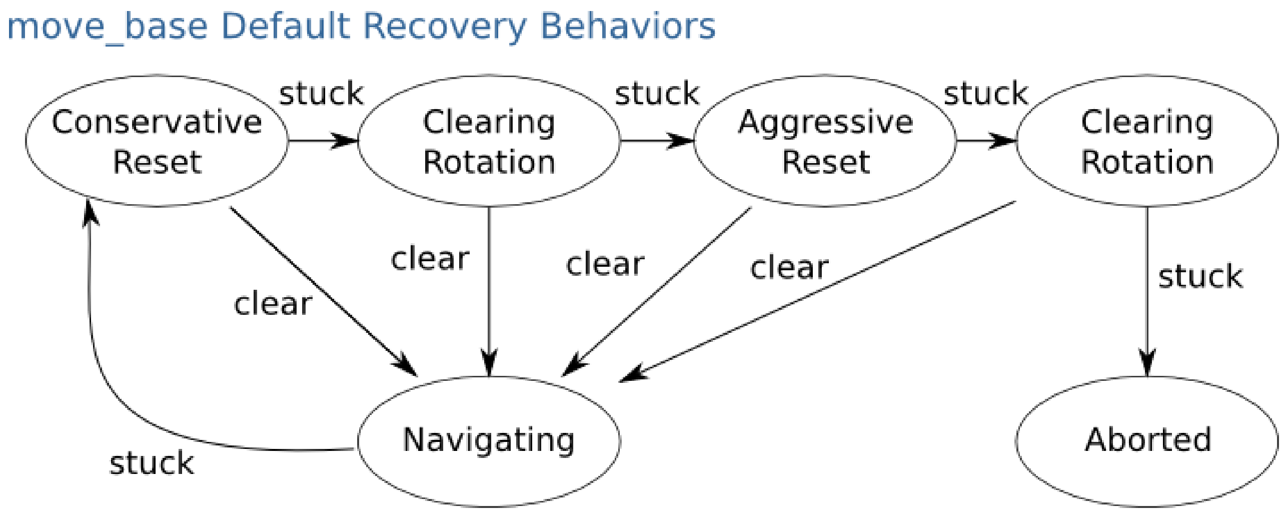


Fig. 2.16. Robot behavior during recovery

When called, move_base will try to ensure that the user-specified target is achieved within some user-specified margin of error. Ultimately, move_base will inform the user that the goal has been achieved or the goal is unattainable. If the robot gets stuck, then the module goes into recovery mode. By default, move_base will do the following to restore the robot’s movement:

1. Obstacles outside the user-defined area will be removed from the robot map.

2. If possible, the robot will rotate in place to clear the space in front of it.

3. If this also fails, the robot will completely clear the area of the map in which the robot can rotate.

4. After that, it will continue its rotation.

If none of the above helps, the goal will be considered unattainable, and its implementation will be interrupted. You can configure the recovery mode yourself.

2.5.2. Teb_local_planner

The Timed Elastic Band (TEB) is an online collision avoidance method for online trajectory optimization. Timed Elastic Band local planner optimizes locally the robot's trajectory minimizing the trajectory execution time (time-optimal objective), separation from obstacles, and compliance with kinodynamic constraints such as satisfying maximum velocities and accelerations [16].

TEB local planner can sometimes get stuck in a locally optimal trajectory as it is unable to transit across obstacles. A subset of admissible trajectories of distinctive topologies is optimized in parallel. The local planner is able to switch to the current globally optimal trajectory among the candidate set. Distinctive topologies are obtained by utilizing the concept of homotopy/homology classes.

Teb_local_planner/obstacles is a topic used to avoid obstacles by a mobile robot in this navigation system. This topic accepts polygons or points as input. The data format is an array ObstacleMsg, which consists of:

– std_msgs/Header header - standard ROS message header;

– geometry_msgs/Polygon polygon - obstacle footprint (polygon descriptions). A polygon consists of an array of points in space. It can also consist of one point, in which case the obstacle will be the point;

– float64 radius - radius for circular / point obstacles;

– int64 id - IDs in order to provide (temporal) relationships between obstacles among multiple messages;

– geometry_msgs/Quaternion orientation - individual orientation (centroid);

– geometry_msgs/TwistWithCovariance velocities - individual velocities (centroid).

2.6. Point cloud to obstacles projection using octomap

The next step after retrieving the raw point cloud is filtering it with PCL. After, the filtered point cloud should be processed the point cloud by octomap, which provides us octrees. Octrees are cast to the floor; what is a projection. This projection should be converted to obstacle format for teb_local_planner.

2.6.1. Point cloud filtering with PCL

The Point Cloud Library (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing. PCL is released under the terms of the BSD license and thus free for commercial and research use. The PCL is used as a package for downsampling and clipping redundant points.

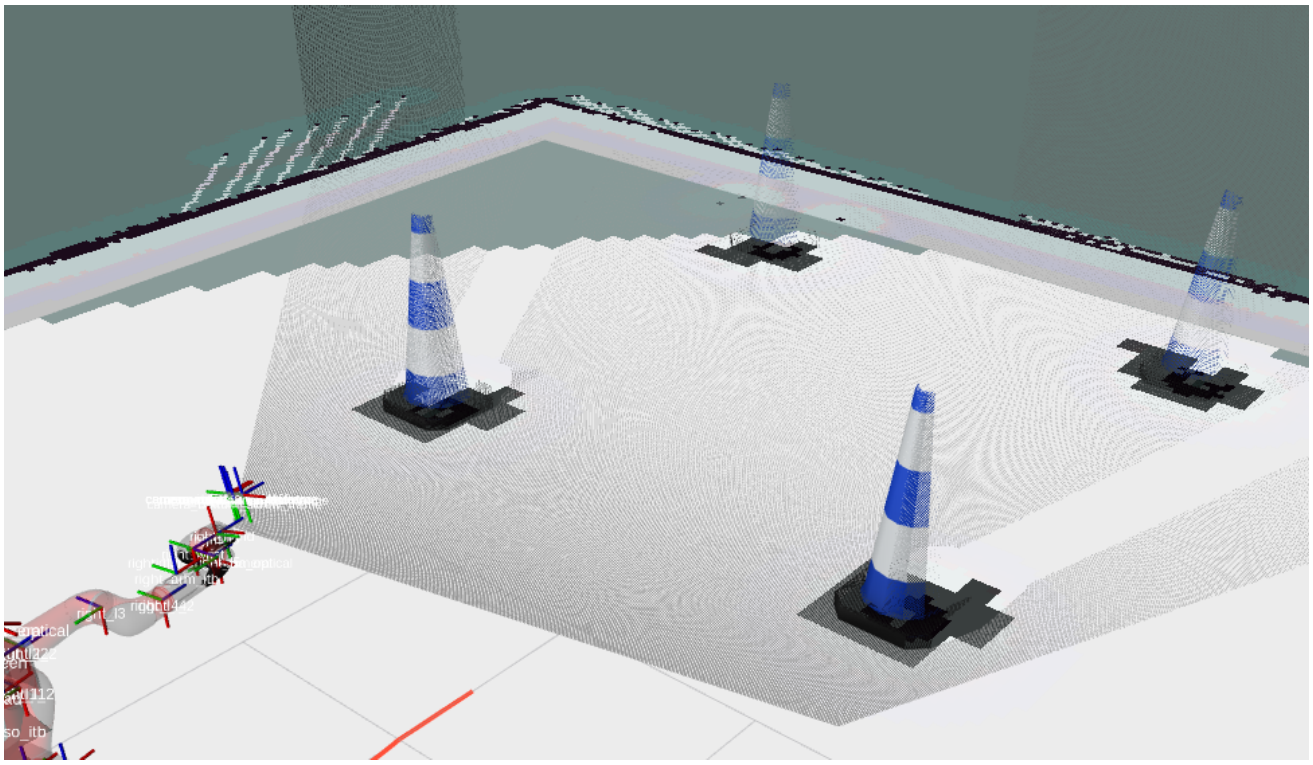
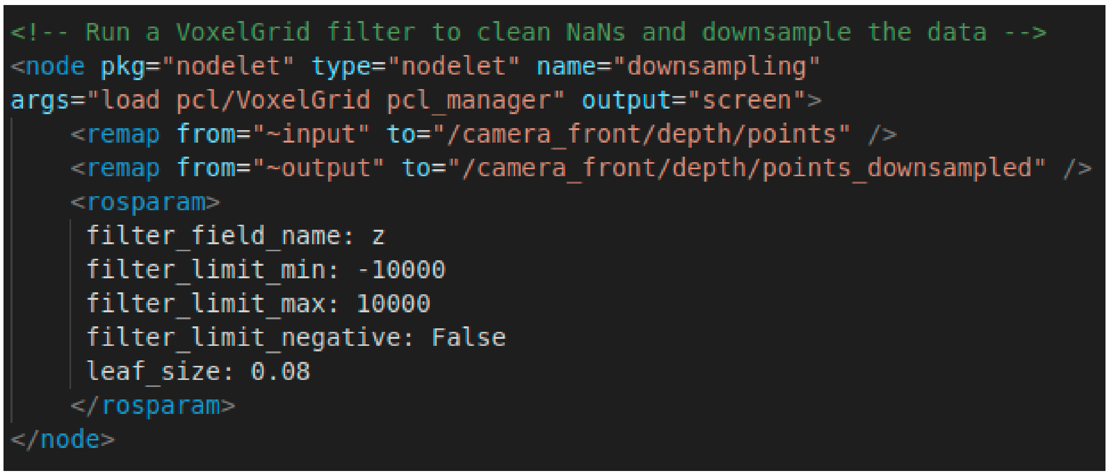


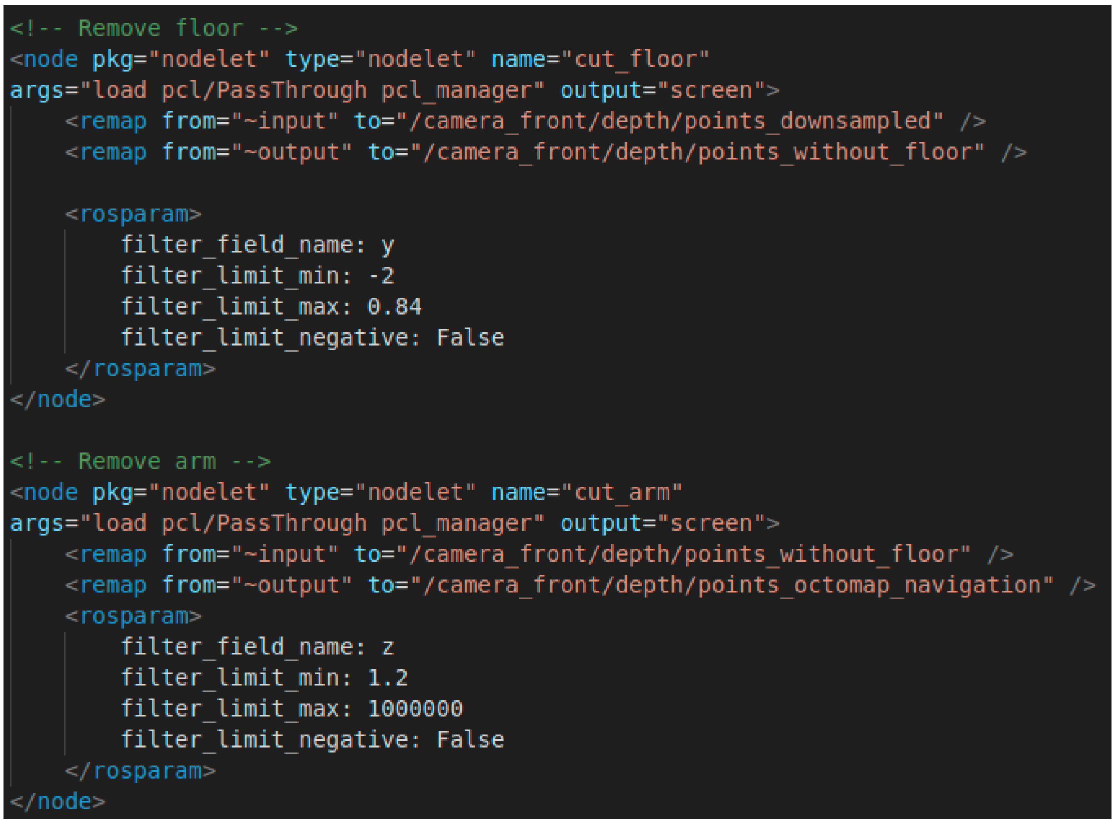
Fig. 2.17. Raw density of point cloud from the realsense camera

Because the density of the received data in the form of a point cloud is too high (Fig. 2.17) for further processing, a downsample VoxelGrid filter was used, which reduces the point cloud's density. VoxelGrid belongs to the PCL library. Configuration of VoxelGrid filter is presented in Fig. 2.18.



```
<!-- Run a VoxelGrid filter to clean NaNs and downsample the data -->
<node pkg="nodelet" type="nodelet" name="downsampling"
args="load pcl/VoxelGrid pcl_manager" output="screen">
    <remap from="~input" to="/camera_front/depth/points" />
    <remap from="~output" to="/camera_front/depth/points_downsampled" />
    <rosparam>
      filter_field_name: z
      filter_limit_min: -10000
      filter_limit_max: 10000
      filter_limit_negative: False
      leaf_size: 0.08
    </rosparam>
</node>
```

Fig. 2.18. Configuration of VoxelGrid filter



```
<!-- Remove floor -->
<node pkg="nodelet" type="nodelet" name="cut_floor"
args="load pcl/PassThrough pcl_manager" output="screen">
    <remap from="~input" to="/camera_front/depth/points_downsampled" />
    <remap from="~output" to="/camera_front/depth/points_without_floor" />

    <rosparam>
        filter_field_name: y
        filter_limit_min: -2
        filter_limit_max: 0.84
        filter_limit_negative: False
    </rosparam>
</node>

<!-- Remove arm -->
<node pkg="nodelet" type="nodelet" name="cut_arm"
args="load pcl/PassThrough pcl_manager" output="screen">
    <remap from="~input" to="/camera_front/depth/points_without_floor" />
    <remap from="~output" to="/camera_front/depth/points_octomap_navigation" />
    <rosparam>
        filter_field_name: z
        filter_limit_min: 1.2
        filter_limit_max: 1000000
        filter_limit_negative: False
    </rosparam>
</node>
```

Fig. 2.19. Configuration of PassThrough filter

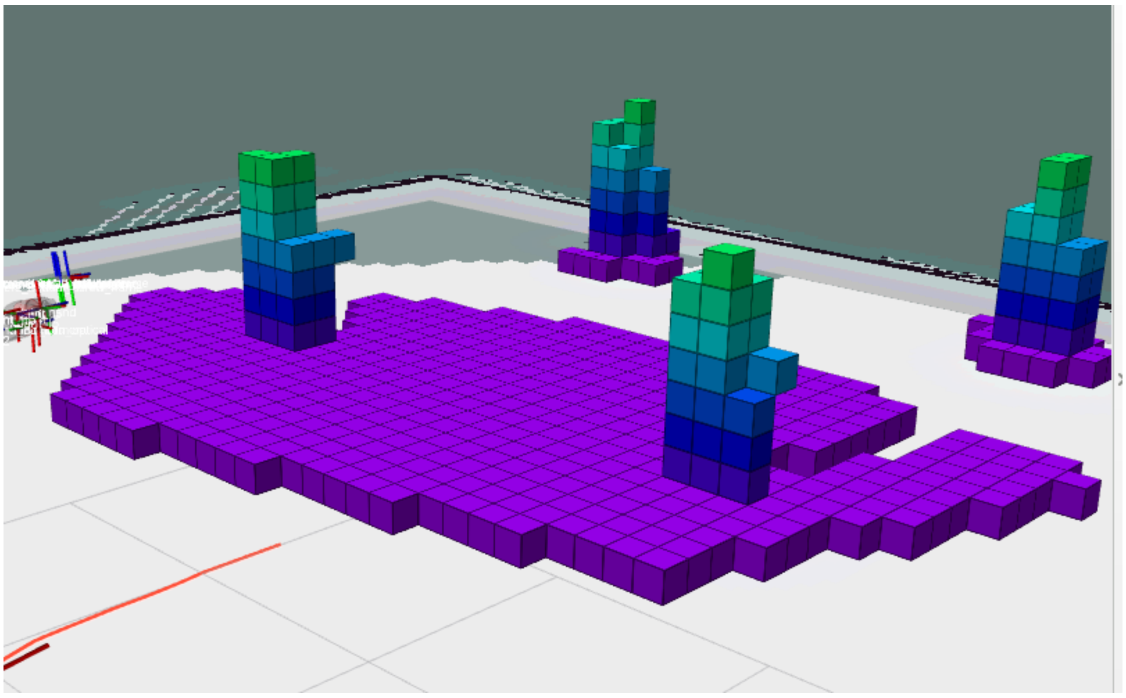


Fig. 2.20. Floor recognized as an obstacle

Another problem was the removal of information about unnecessary objects in the camera’s field of view. These include the floor, ceiling, parts of the robot (such as a manipulator). The PassThrough filter from the PCL library was used to solve this problem. Its configuration is shown in Fig. 2.19.

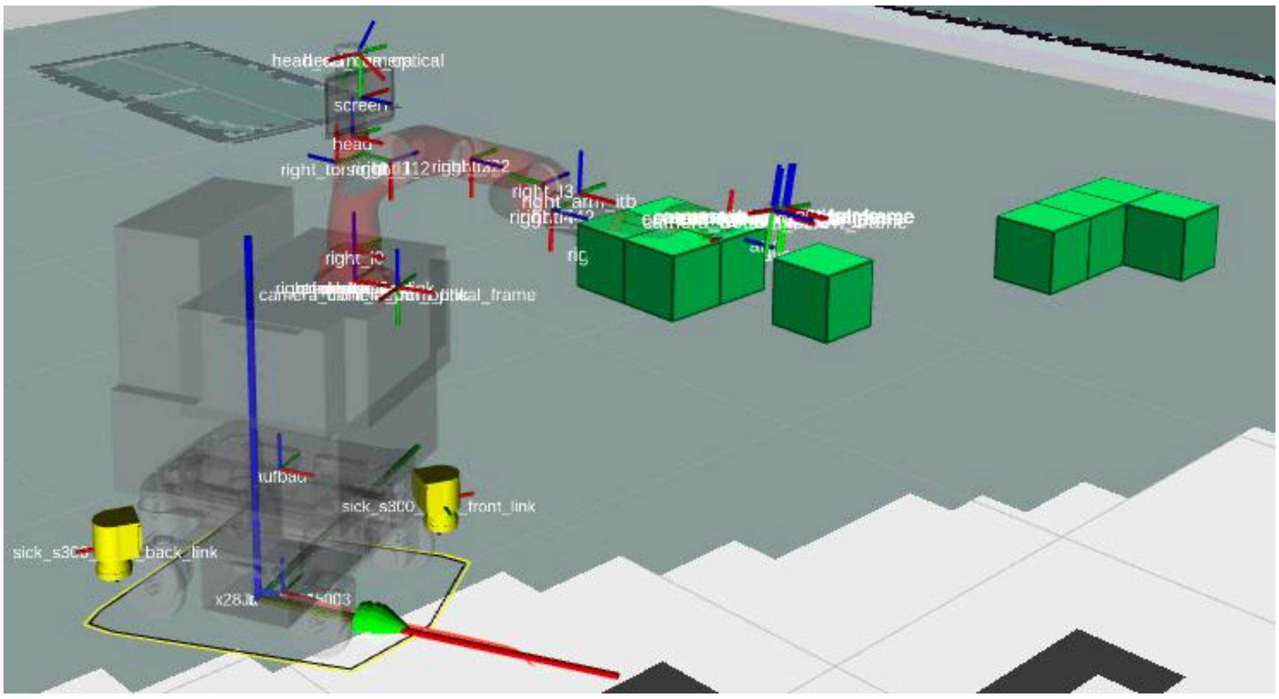


Fig. 2.21. Arm recognized as an obstacle

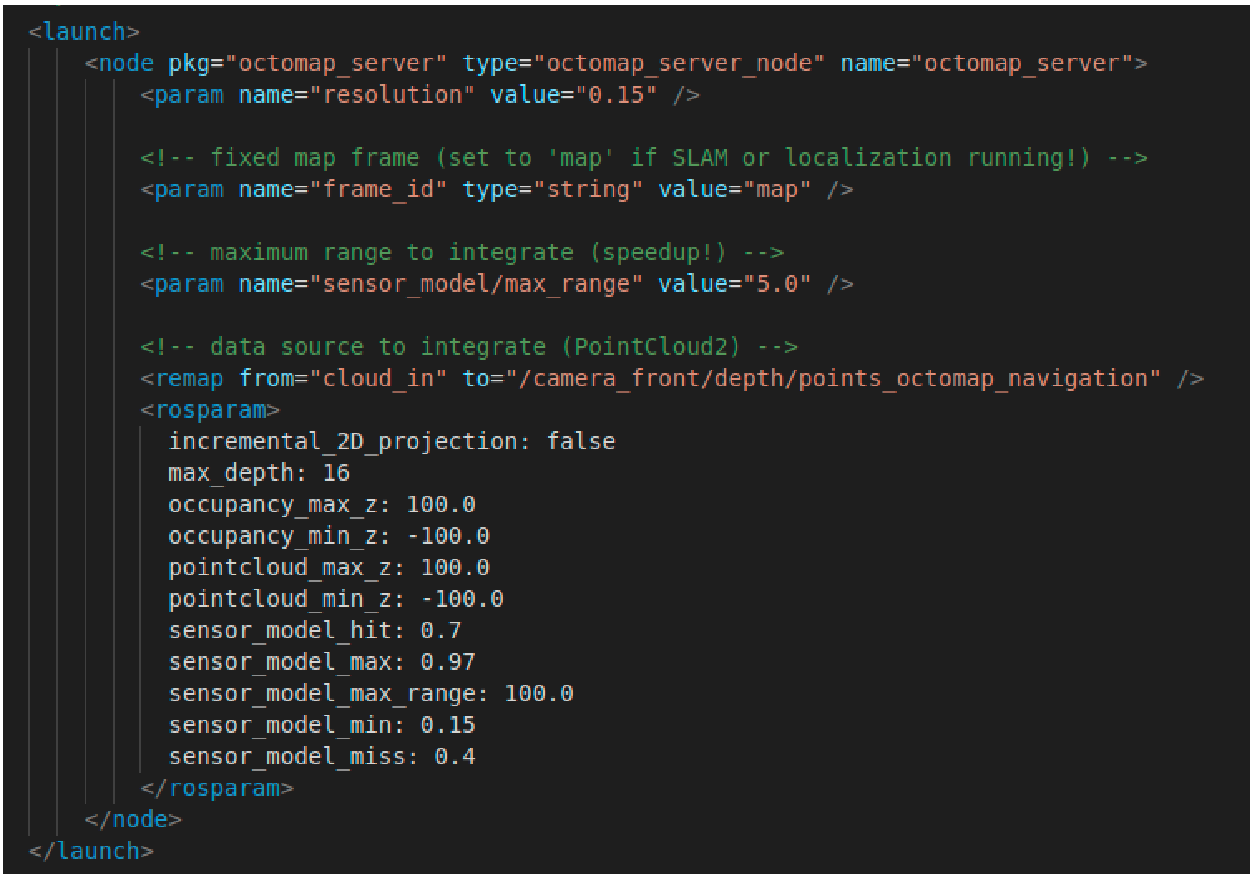
In figure 2.20. the whole floor in the field of view of the camera is converted to octree. If it had left this way, it would have recognized as an obstacle.

Fig. 2.21. represents the issue with the robot`s arm recognition as obstacles. Thus, the configuration of the PassThrough filter for arm and floor was applied.

2.6.2. Octomap

The next step is to process the filtered data using OctoMap. As a result, an octree is generated.

OctoMap provides converting point cloud data to occupancy grid octree. Then these occupied octants(voxels) are projected to the floor, providing an image of the occupied area. The configuration is presented in Fig. 2.22. Fig. 2.23 shows the resulting octree.



```xml
<launch>
    <node pkg="octomap_server" type="octomap_server_node" name="octomap_server">
        <param name="resolution" value="0.15" />

        <!-- fixed map frame (set to 'map' if SLAM or localization running!) -->
        <param name="frame_id" type="string" value="map" />

        <!-- maximum range to integrate (speedup!) -->
        <param name="sensor_model/max_range" value="5.0" />

        <!-- data source to integrate (PointCloud2) -->
        <remap from="cloud_in" to="/camera_front/depth/points_octomap_navigation" />
        <rosparam>
          incremental_2D_projection: false
          max_depth: 16
          occupancy_max_z: 100.0
          occupancy_min_z: -100.0
          pointcloud_max_z: 100.0
          pointcloud_min_z: -100.0
          sensor_model_hit: 0.7
          sensor_model_max: 0.97
          sensor_model_max_range: 100.0
          sensor_model_min: 0.15
          sensor_model_miss: 0.4
        </rosparam>
    </node>
</launch>
```

Fig. 2.22. Octomap server configuration

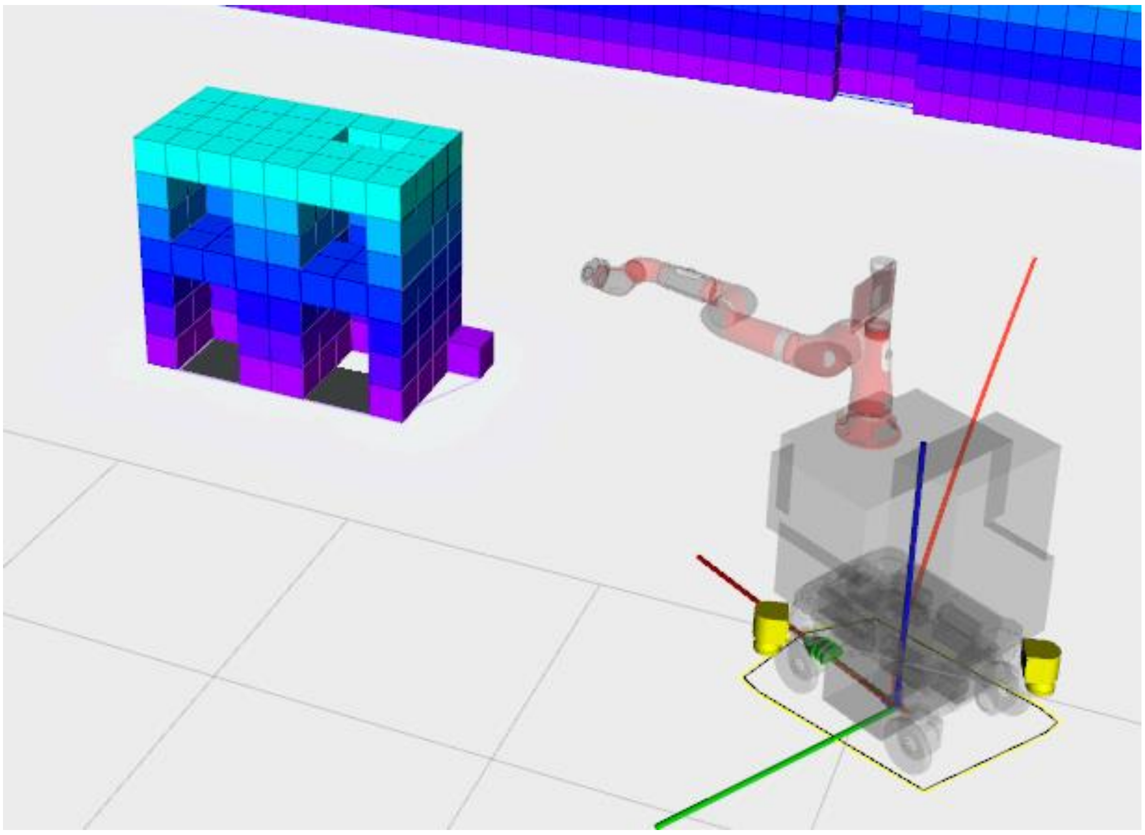


Fig. 2.23. Octree produced by octomap

A top-down projection is created by the Octomap itself, and it is published to /projected_map from the octree generated in the previous step. Fig. 2.24. shows projection as big black pixels.

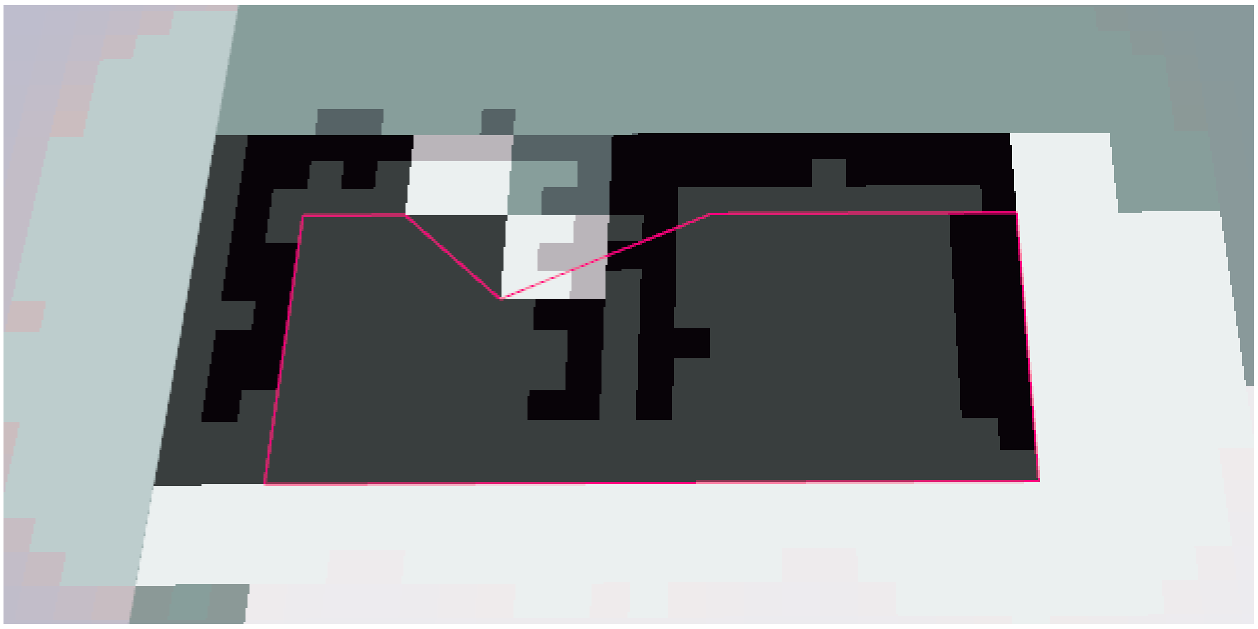


Fig. 2.24. Projection generated from octree

The OctoMap library implements a 3D occupancy grid mapping approach, providing data structures and mapping algorithms in C++, particularly suited for robotics. The map implementation is based on an octree and is designed to meet the following requirements:

– Full 3D model. The map is able to model arbitrary environments without prior assumptions about them. The representation models occupied areas as well as free space. Unknown areas of the environment are implicitly encoded in the map. While the distinction between free and occupied space is essential for safe robot navigation, information about unknown areas is important, e.g., for autonomous exploration of an environment.

– Updatable. It is possible to add new information or sensor readings at any time. Modeling and updating are done in a probabilistic fashion. This accounts for sensor noise or measurements which result from dynamic changes in the environment, e.g., because of dynamic objects. Furthermore, multiple robots are able to contribute to the same map, and a previously recorded map is extendable when new areas are explored.

– Flexible. The extent of the map does not have to be known in advance. Instead, the map is dynamically expanded as needed. The map is multi-resolution so that, for instance, a high-level planner is able to use a coarse map, while a local planner may operate using a fine resolution. This also allows for efficient visualizations that scale from coarse overviews to detailed close-up views.

– Compact. The map is stored efficiently, both in memory and on disk. It is possible to generate compressed files for later usage or convenient exchange between robots, even under bandwidth constraints [17].

2.7. Projection to obstacles

But projection itself is not an obstacle for the navigation stack. The most appropriate input with obstacles for the navigation stack is an obstacle topic teb_local_planner subscribed on. It can consume obstacles as point obstacles or polygons.

Firstly, direct conversion from occupied pixels to point obstacles were developed. This solution does not provide needed path planning because teb_local_planner tries to build the route through the obstacle between close points where it is not enough space to go.

The better way is to build polygons around the occupied areas on the projection.

In this case, there no small space inside the occupied area, and the path is built appropriately around the obstacle.

In this manner, the mobile robot will be able to recognize and to avoid the obstacle, which was found on the way from the starting point to the goal. The mobile robot will change the route, which was built, accordingly to the obstacle which was found.

2.8. Conclusions to the second section

The second section is devoted to the development of path planning algorithms. The section describes the robot on which the system was tested, the details of the sensors and depth cameras used. Stages of realization of the new system, steps of data processing are described. Attention is paid to the ROS system, simulators and important and relevant implementations of these algorithms and navigation approaches.

The research was carried out on the mobile robot “Leonart” of the RT Lions team at Reutlingen University. The mobile robot was created to educate students, with practical purposes, and for participation in RoboCup competitions.

The Intel RealSense Depth Camera D435 Series uses stereo vision to calculate depth. They are fully calibrated and produce hardware-rectified pairs of images. This camera performs all depth calculations at up to 90 FPS and offers sub-pixel accuracy and a high fill-rate. It has the inertial measurement unit (IMU), which is used for the detection of movements and rotations.

The mobile robot has S300 laser model from SICK. Measurement data from one or two laser scanners can be used for localization, navigation, and collision avoidance. The safety-approved laser scanner S300 can monitor user-defined safety fields around the robot, which can be dynamically activated by application-specific control software. As soon as an object is detected within the currently activated field, the robot is immediately set to an emergency stop. This control software is responsible for the correct activation of the safety fields according to the current condition.

The Robot Operating System (ROS) – is a widely used robot software development framework with distributed teamwork capabilities. The way of thinking of ROS is to create software that works with different robots, with only minor changes to the code. This system allows the parallel launch of several programs, as well as support the exchange of information between them.

RViz is the primary visualizer in ROS and an incredibly useful tool for debugging robotics. RViz is a tool for 3D visualization of data (messages) coming in on ROS topics. By visualizing data, developers can evaluate how the robot “sees” its environment and how it perceives itself in it - its position, orientation in space, and location on the map. RViz assists in debugging ROS software components and is an indispensable tool when working with a robot in both real environment and simulation.

Gazebo is a 3D simulator that provides robots, sensors, environment models for 3D simulation required for robot development and offers realistic simulation with its physics engine. Gazebo is one of the most popular simulators for open-source robotics in recent years and has been widely used in the field of robotics.

TF maintains the relationship between coordinate frames in a tree structure buffered in time and lets the user transform points, vectors, etc., between any two coordinate frames at any desired point in time.

The move_base node provides a ROS interface for configuring, running, and interacting with the navigation stack on a robot. A high-level view of the move_base node and its interaction with other components is shown above.

The first step of the developed algorithm is to obtain a cloud of raw points. There is also no need to use such many points. Therefore, using the PCL library, the resolution of row data has been reduced and redundant objects have been removed, such as robot parts, floor, and ceiling. After the above processing, the point cloud is sent to Octomap, where an octree is created from it, and an octree is projected onto the floor. Further, using the developed software, the projection is converted into obstacles suitable for sending to teb_local_planner. In this section, the above structure will be described in more detail along with a description of the hardware and software platform.

SECTION 3
RESULTS OF THE DEVELOPED SYSTEM

3.1. Configuration of the navigation stack

The navigation stack of the robot is consisting of amcl, gmapping, and move_base packages. To set up the stack properly, we need to set configuration parameters. The parameters are described below.

Table 3.1

AMCL parameters

Parameter	Value
min_particles	100
max_particles	2000
kld_err	0.01
kld_z	0.99
update_min_d	0.1
update_min_a	0.1
resample_interval	2
transform_tolerance	0.2
initial_pose_x	0.0
initial_pose_y	0.0
initial_pose_a	0.0
initial_cov_xx	0.01
initial_cov_yy	0.01
initial_cov_aa	0.01
gui_publish_rate	2.0
save_pose_rate	0.5
use_map_topic	true

Continuation of Table 3.1

Parameter	Value
first_map_only	false
laser_min_range	-1.0
laser_max_range	-1.0
laser_max_beams	100
laser_likelihood_max_dist	2.0
laser_model_type	likelihood_field_prob
do_beamskip	true
odom_model_type	omni-corrected
odom_alpha1	0.1
odom_alpha3	0.1
odom_frame_id	odom
base_frame_id	base_link
global_frame_id	map
tf_broadcast	true

Table 3.2

Gmapping parameters

Parameter	Value
base_frame	"base_link"
map_frame	“map”
odom_frame	“odom”
map_update_interval	2.0
maxUrange	10.0
sigma	0.05
kernelSize	1
lstep	0.05
astep	0.05

Continuation of Table 3.2

Parameter	Value
iterations	5
ogain	3.0
minimumScore	50.0
linearUpdate	0.5
angularUpdate	0.5
temporalUpdate	-1.0
particles	50
delta	0.05
transform_publish_period	0.05
occ_thresh	0.25
maxRange	30

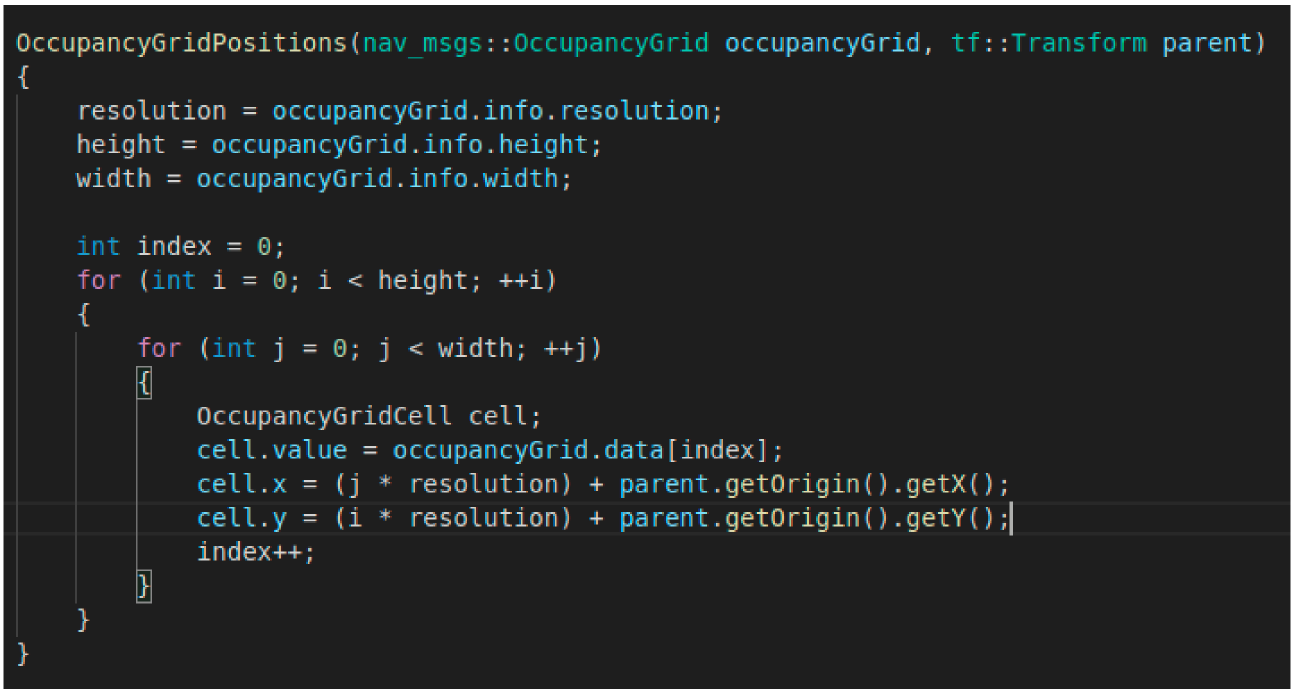
Table 3.3

Move_base parameters

Parameter	Value
controller_frequency	20.0
planner_patience	5.0
controller_patience	15.0
conservative_reset_dist	3.0
recovery_behavior_enabled	true
clearing_rotation_allowed	true
shutdown_costmaps	false
oscillation_distance	0.4
oscillation_timeout	10.0
planner_frequency	0.0
max_planning_retries	8

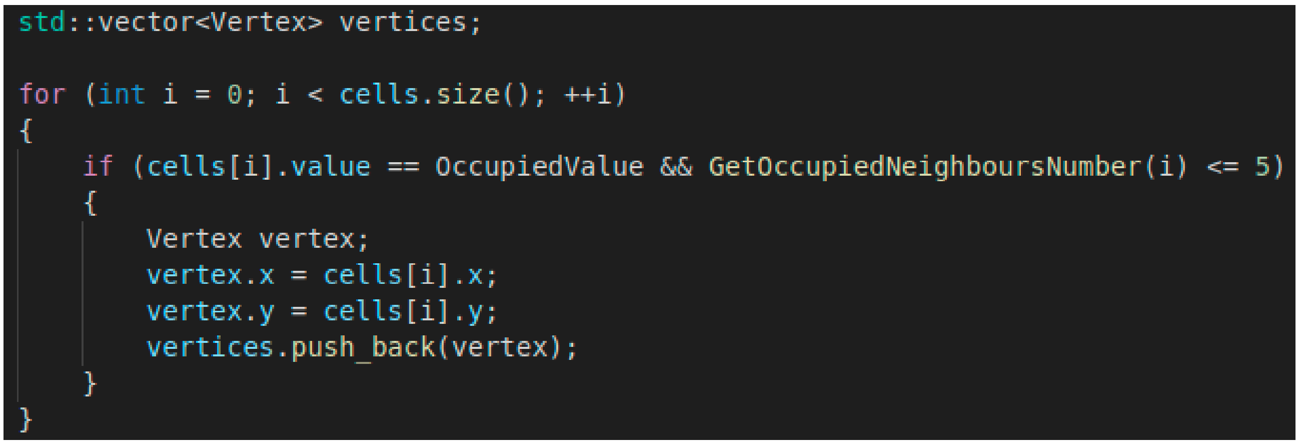
3.2. Projection to obstacles. Details.

In this part, projection_to_obstacles package code will be presented and described with details. In the very first step, OccupancyGrid is converted to the internal structure OccupancyGridPositions (Fig. 3.1). The new values are computed considering the origin of the projection, so that each pixel`s positions appeared in world space. Then non-border vertices are removed.



```
OccupancyGridPositions(nav_msgs::OccupancyGrid occupancyGrid, tf::Transform parent)
{
    resolution = occupancyGrid.info.resolution;
    height = occupancyGrid.info.height;
    width = occupancyGrid.info.width;

    int index = 0;
    for (int i = 0; i < height; ++i)
    {
        for (int j = 0; j < width; ++j)
        {
            OccupancyGridCell cell;
            cell.value = occupancyGrid.data[index];
            cell.x = (j * resolution) + parent.getOrigin().getX();
            cell.y = (i * resolution) + parent.getOrigin().getY();
            index++;
        }
    }
}
```

Fig. 3.1. Conversion from OccupancyGrid to OccupancyGridPositions

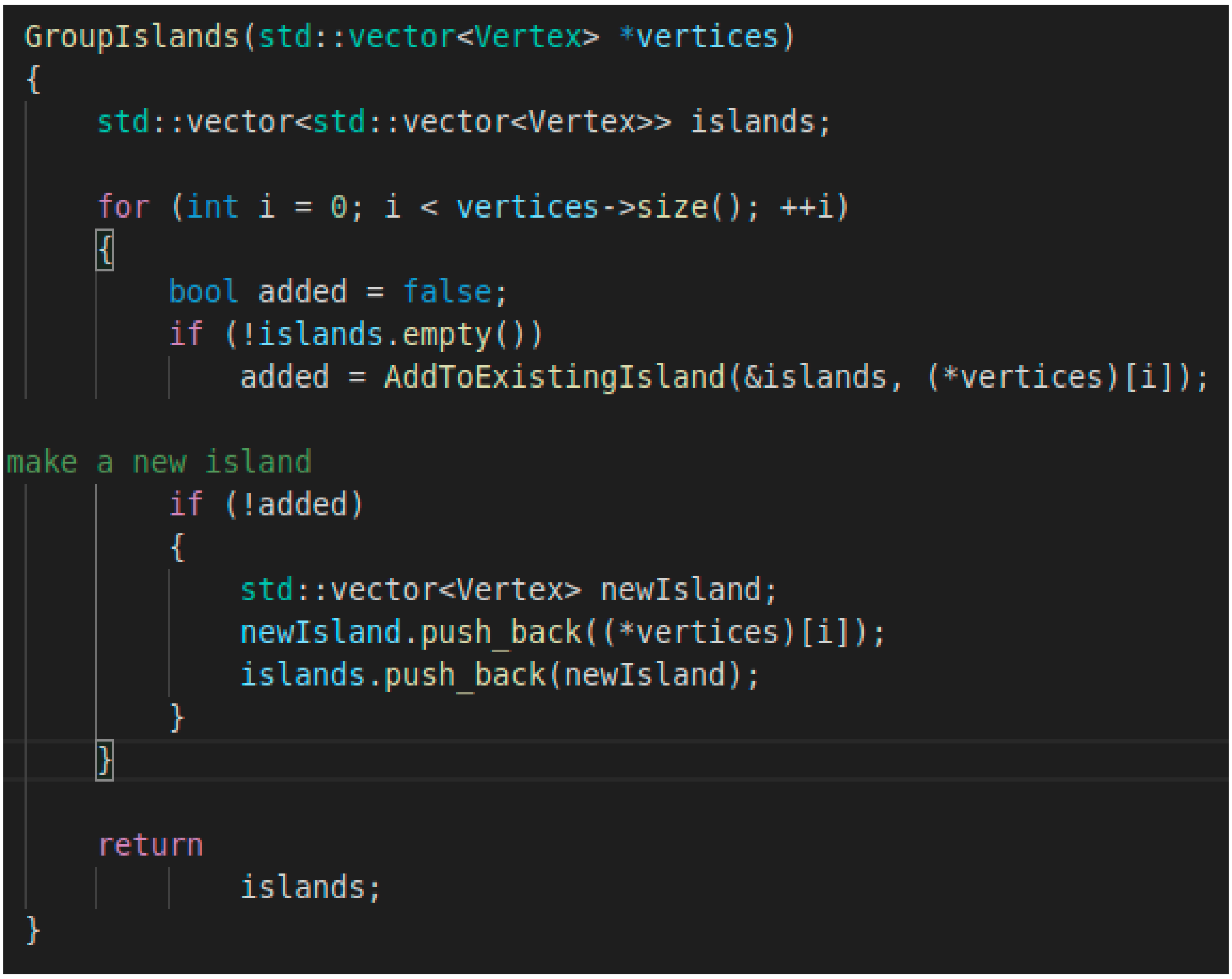


```
std::vector<Vertex> vertices;

for (int i = 0; i < cells.size(); ++i)
{
    if (cells[i].value == OccupiedValue && GetOccupiedNeighboursNumber(i) <= 5)
    {
        Vertex vertex;
        vertex.x = cells[i].x;
        vertex.y = cells[i].y;
        vertices.push_back(vertex);
    }
}
```

Fig. 3.2. Keep only key vertices

Then we keep only key vertices (Fig. 3.2). Key points are responsible for the shape of the group's border. Removing no key will not affect the shape. So they are redundant.

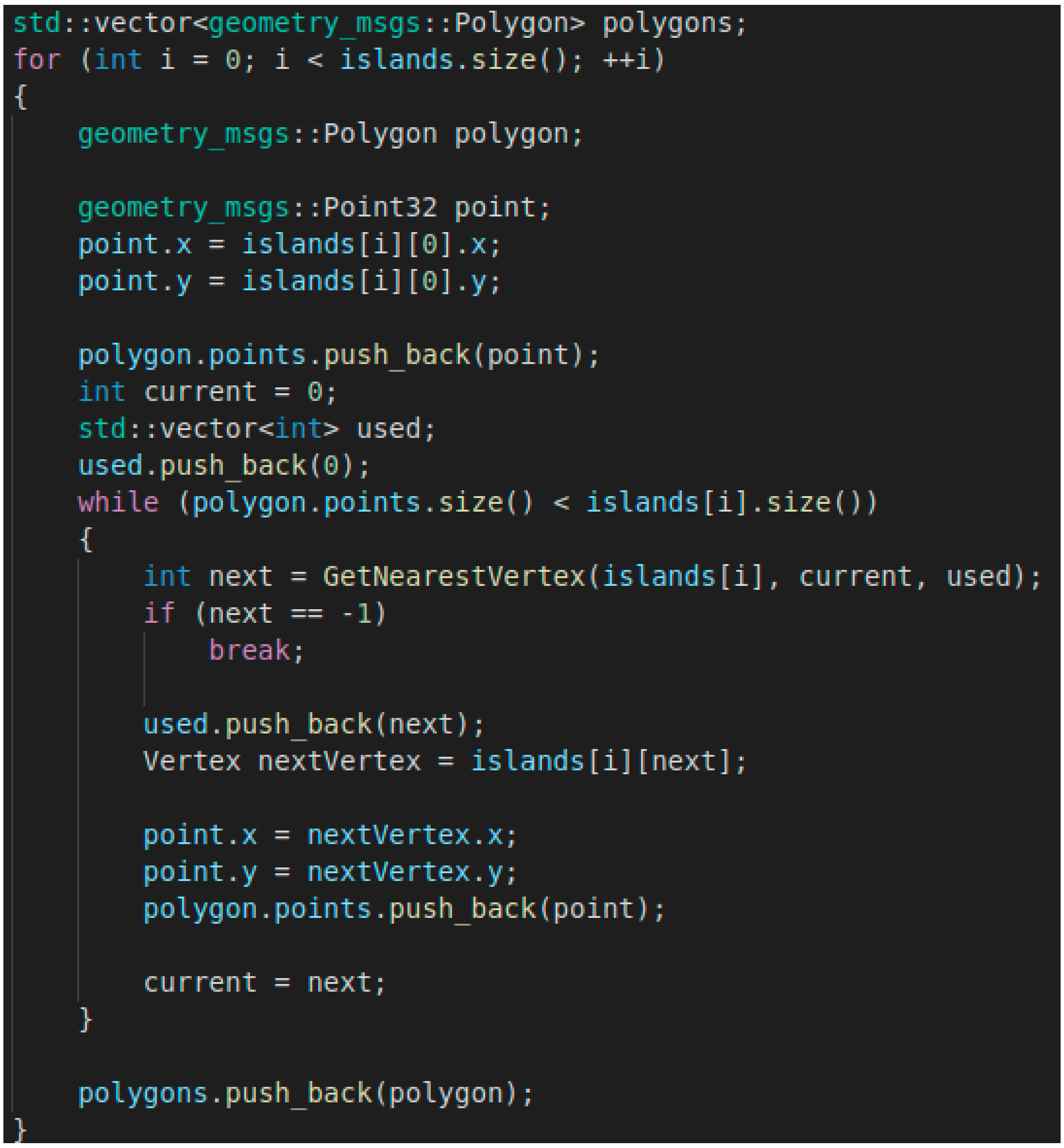
During the next step, we divide all vertices into independent islands when they have no adjacent vertices (Fig. 3.3).



```
GroupIslands(std::vector<Vertex> *vertices)
{
    std::vector<std::vector<Vertex>> islands;

    for (int i = 0; i < vertices->size(); ++i)
    {
        bool added = false;
        if (!islands.empty())
            added = AddToExistingIsland(&islands, (*vertices)[i]);

make a new island
        if (!added)
        {
            std::vector<Vertex> newIsland;
            newIsland.push_back((*vertices)[i]);
            islands.push_back(newIsland);
        }
    }

    return
            islands;
}
```

Fig. 3.3. Island producing

When there are only key points, they can be circularly connected. An appropriately connected array of vertices is basically a polygon (Fig. 3.4). This polygon can be sent directly to teb_local_planner.



```cpp
std::vector<geometry_msgs::Polygon> polygons;
for (int i = 0; i < islands.size(); ++i)
{
    geometry_msgs::Polygon polygon;

    geometry_msgs::Point32 point;
    point.x = islands[i][0].x;
    point.y = islands[i][0].y;

    polygon.points.push_back(point);
    int current = 0;
    std::vector<int> used;
    used.push_back(0);
    while (polygon.points.size() < islands[i].size())
    {
        int next = GetNearestVertex(islands[i], current, used);
        if (next == -1)
            break;

        used.push_back(next);
        Vertex nextVertex = islands[i][next];

        point.x = nextVertex.x;
        point.y = nextVertex.y;
        polygon.points.push_back(point);

        current = next;
    }

    polygons.push_back(polygon);
}
```

Fig. 3.4. Polygons producing

3.3. System setup for simulation

It is necessary to use Ubuntu 16.04 and install ROS Kinetic. You can follow instructions from this link: http://wiki.ros.org/kinetic/Installation/Ubuntu.

Next step is packages installation. You can use these commands:

```
$ sudo apt install ros-kinetic-ddynamic-reconfigure
$ sudo apt install ros-kinetic-map-server
$ sudo apt install ros-kinetic-move-base
$ sudo apt install ros-kinetic-hardware-interface
$ sudo apt install ros-kinetic-amcl
$ sudo apt install ros-kinetic-effort-controllers
$ sudo apt install ros-kinetic-gazebo-ros-control
$ sudo apt install ros-kinetic-sns-ik-lib
$ sudo apt install ros-kinetic-teb-local-planner
$ rosdep install teb_local_planner
```

Then you can download next packages. Firstly, open src folder in terminal and use next commands:

```
$ cd ~/catkin_ws/src
$ git clone https://gitlab.com/rtlions/robots/neo_mpo_700.git
$ git clone -
b navi_octomap https://gitlab.com/rtlions/navigation/navigation_teb.git
$ git clone -b navi_octomap https://gitlab.com/rtlions/robot/rtl_leonart.git
$ git clone -b navi_octomap https://gitlab.com/rtlions/robot/rtl_simulation.git
$ git clone https://github.com/neobotix/neo_simulation.git
$ git
clone https://RudViacheslav@bitbucket.org/RudViacheslav/projection_to_obstacles.g
it
$ git clone https://github.com/RethinkRobotics/intera_sdk.git
$ git clone https://github.com/RethinkRobotics/intera_common.git
$ git clone -
b obstacle_avoidance https://github.com/ViacheslavRud/sawyer_robot.git
$ git clone -
b obstacle_avoidance https://github.com/ViacheslavRud/sawyer_simulator.git
$ git clone -
b obstacle_avoidance https://github.com/ViacheslavRud/sawyer_moveit.git
$ git clone -b obstacle_avoidance https://github.com/ViacheslavRud/realsense-
ros.git
```

If you have an error with realsense lib version, you can change in file realsense-ros/ realsense2_camera next line find_package(realsense2 2.35.2) to find_package(realsense2 2.34).

Put https://github.com/ipa320/cob_driver/tree/kinetic_dev/cob_scan_unifier into src folder as well.

Apply a fix in sawyer_simulator/sawyer_gazebo/launch/sawyer_world.launch: remove “-z 0.93” in line 75:

Before fix:

```
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"  args="-param robot_description -urdf -z 0.93 -model sawyer $(arg initial_joint_states)" />
```

After fix:

```
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"  args="-param robot_description -urdf -model sawyer $(arg initial_joint_states)" />
```

Then build:

```
cd {workspace}
catkin_make
```

3.4. System setup for robot

The setup process for the robot is similar to the setup for simulation but has some differences.

It is necessary to use Ubuntu 16.04 and install ROS Kinetic. You can follow instructions from this link: http://wiki.ros.org/kinetic/Installation/Ubuntu.

Next step is packages installation. You can use these commands:

```
$ sudo apt install ros-kinetic-ddynamic-reconfigure
$ sudo apt install ros-kinetic-map-server
$ sudo apt install ros-kinetic-move-base
$ sudo apt install ros-kinetic-hardware-interface
$ sudo apt install ros-kinetic-amcl
$ sudo apt install ros-kinetic-effort-controllers
$ sudo apt install ros-kinetic-sns-ik-lib
$ sudo apt install ros-kinetic-teb-local-planner
$ rosdep install teb_local_planner
```

Then you can download next packages. Firstly, open src folder in terminal and use next commands:

```
$ cd ~/catkin_ws/src
$ git clone https://gitlab.com/rtlions/robots/neo_mpo_700.git
$ git clone -
b navi_octomap https://gitlab.com/rtlions/navigation/navigation_teb.git
$ git clone -b navi_octomap https://gitlab.com/rtlions/robot/rtl_leonart.git
$ git clone -b navi_octomap https://gitlab.com/rtlions/robot/rtl_simulation.git
$ git clone https://github.com/neobotix/neo_simulation.git
$ git
clone https://RudViacheslav@bitbucket.org/RudViacheslav/projection_to_obstacles.g
it
$ git clone https://github.com/RethinkRobotics/intera_sdk.git
$ git clone https://github.com/RethinkRobotics/intera_common.git
$ git clone -
b obstacle_avoidance https://github.com/ViacheslavRud/sawyer_robot.git
$ git clone -
b obstacle_avoidance https://github.com/ViacheslavRud/sawyer_simulator.git
$ git clone -
b obstacle_avoidance https://github.com/ViacheslavRud/sawyer_moveit.git
$ git clone -b obstacle_avoidance https://github.com/ViacheslavRud/realsense-
ros.git
```

If you have an error with realsense lib version, you can change in file realsense-ros/ realsense2_camera next line find_package(realsense2 2.35.2) to find_package(realsense2 2.34).

Then build:

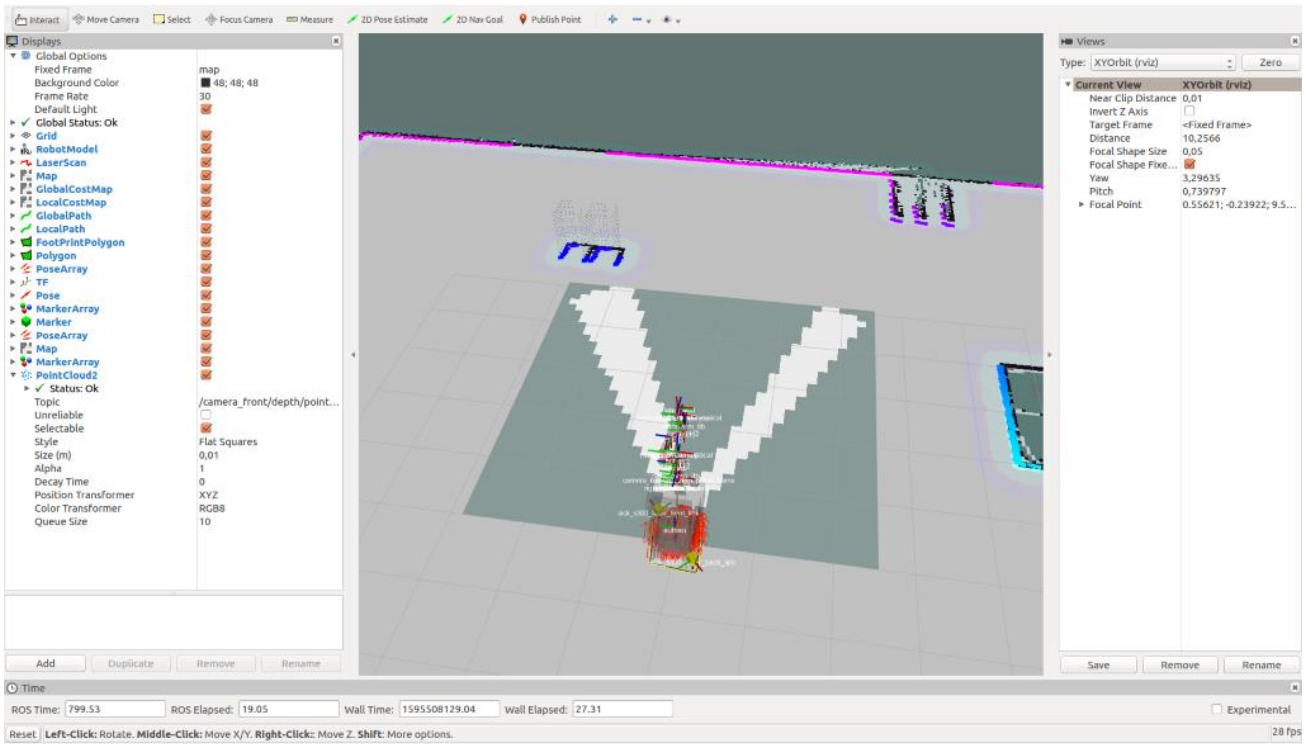
cd {workspace}

catkin_make

3.5. Run in simulation

It is important to have all the necessary modules and ROS installed on the system.

1. Open a new terminal window.
2. Call roslaunch rtl_simulation simulation.launch (Fig. 3.5. and Fig. 3.6).



Displays

Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
Global Status: Ok	
Grid	✓
RobotModel	✓
LaserScan	✓
Map	✓
GlobalCostMap	✓
LocalCostMap	✓
GlobalPath	✓
LocalPath	✓
FootPrintPolygon	✓
Polygon	✓
PoseArray	✓
TF	✓
Pose	✓
MarkerArray	✓
Marker	✓
PoseArray	✓
Map	✓
MarkerArray	✓
PointCloud2	✓
Status: Ok	
Topic	/camera_front/depth/point...
Unreliable	☐
Selectable	✓
Style	Flat Squares
Size (m)	0,01
Alpha	1
Decay Time	0
Position Transformer	XYZ
Color Transformer	RGB8
Queue Size	10

Views

Type: XYOrbit (rviz)

Current View	XYOrbit (rviz)
Near Clip Distance	0,01
Invert Z Axis	☐
Target Frame	<Fixed Frame>
Distance	10,2566
Focal Shape Size	0,05
Focal Shape Fixe...	✓
Yaw	3,29635
Pitch	0,739797
Focal Point	0.55621; -0.23922; 9.5...

Time

ROS Time: 799.53 ROS Elapsed: 19.05 Wall Time: 1595508129.04 Wall Elapsed: 27.31 ☐ Experimental

Reset **Left-Click:** Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Move Z. **Shift**: More options. 28 fps

Fig. 3.5. RViz screen after launch

3. Open a new terminal window.
4. Call rosrun projection_to_obstacles projection_to_obstacles (Fig. 3.7).

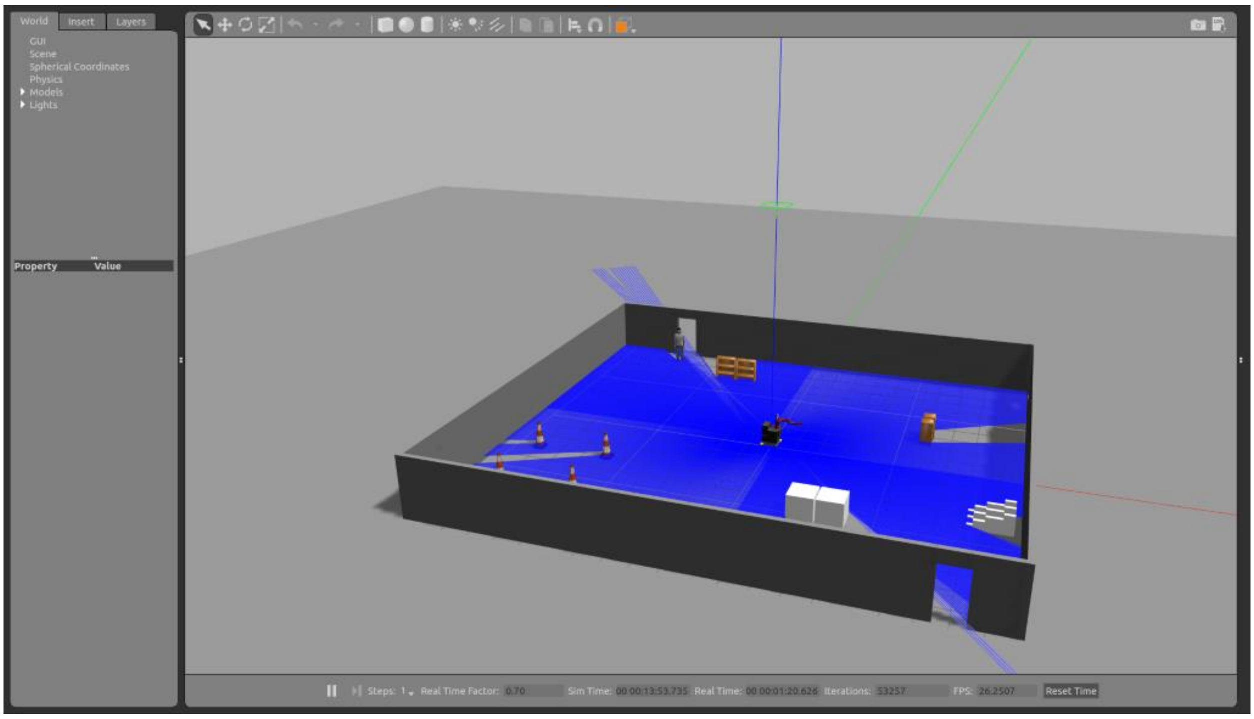


Fig. 3.6. Gazebo screen after launch

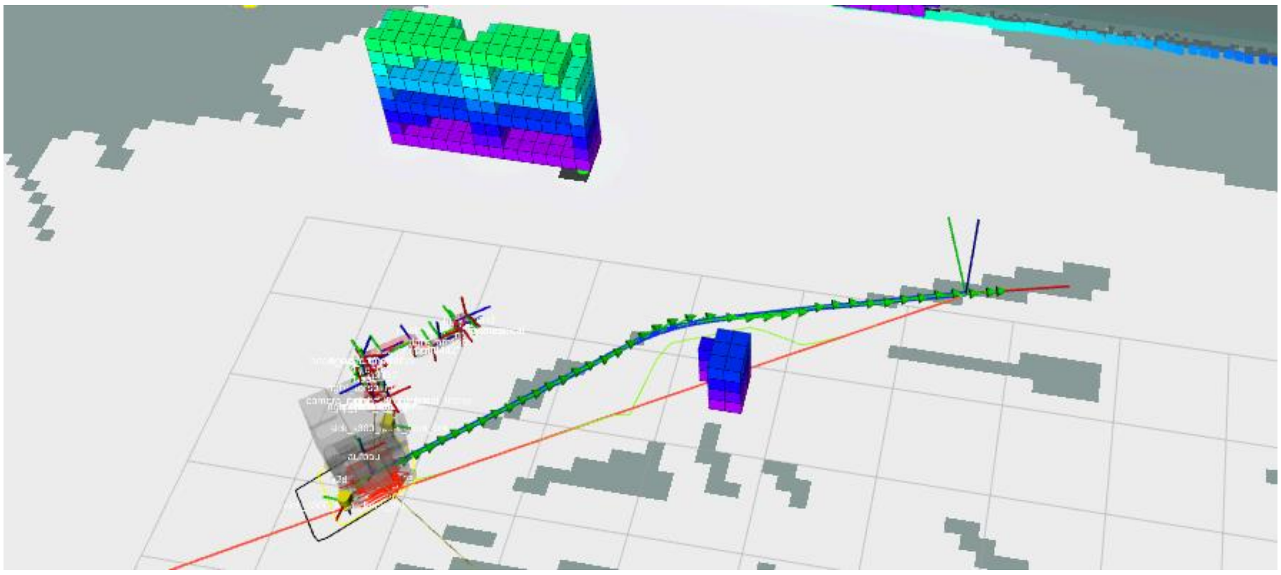
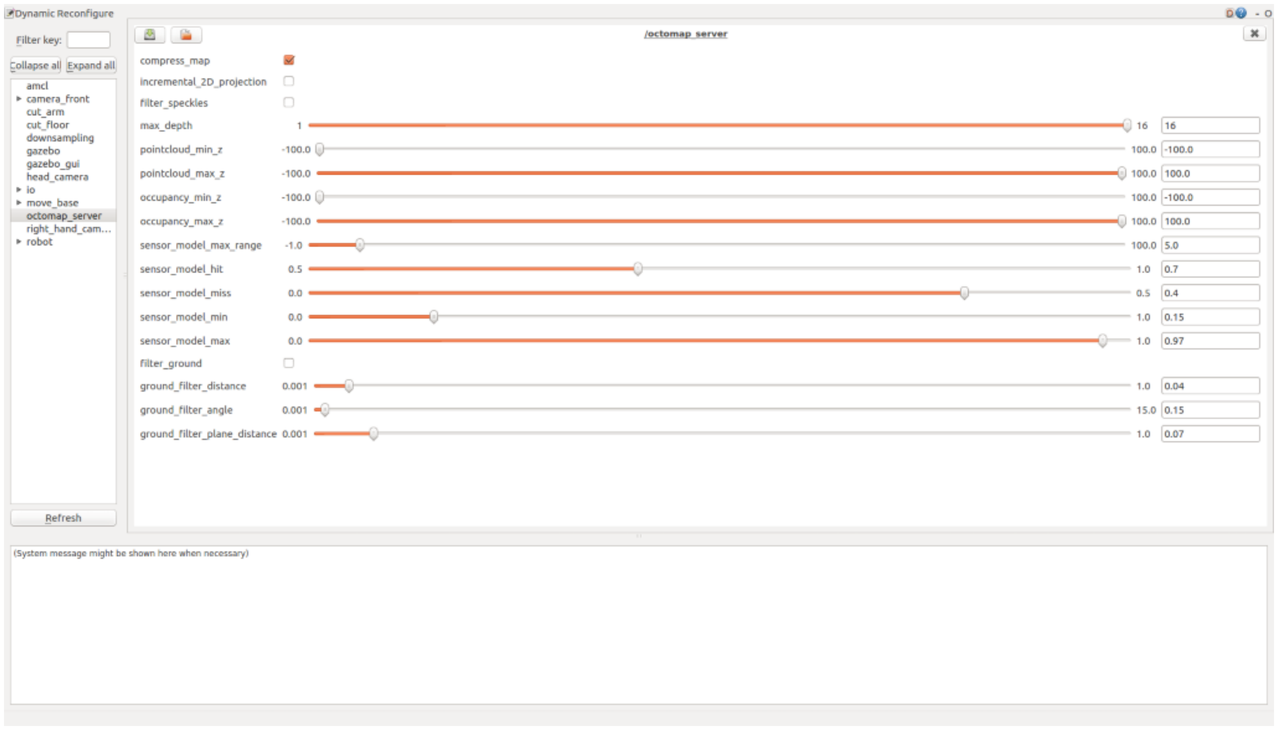


Fig. 3.7. Changes after projection_to_obstacles package execution

5. In case if you need to change parameters, you can use this command: rosrun rqt_reconfigure rqt_reconfigure (Fig. 3.8).



/octomap_server

compress_map	✓		
incremental_2D_projection	☐		
filter_speckles	☐		
max_depth	1	16	16
pointcloud_min_z	-100.0	100.0	-100.0
pointcloud_max_z	-100.0	100.0	100.0
occupancy_min_z	-100.0	100.0	-100.0
occupancy_max_z	-100.0	100.0	100.0
sensor_model_max_range	-1.0	100.0	5.0
sensor_model_hit	0.5	1.0	0.7
sensor_model_miss	0.0	0.5	0.4
sensor_model_min	0.0	1.0	0.15
sensor_model_max	0.0	1.0	0.97
filter_ground	☐		
ground_filter_distance	0.001	1.0	0.04
ground_filter_angle	0.001	15.0	0.15
ground_filter_plane_distance	0.001	1.0	0.07

(System message might be shown here when necessary)

Fig. 3.8. Rqt_reconfigure GUI screen

Required input data is a point cloud. It is a necessary input data, which should be written into the needed topic.

3.6. Run on real robot

Start the robot:

1. Start Leonart (the upper black button on the backside of the robot)
2. Wait for 30s
3. Start Neobotix (turn and hold the key for 5 seconds)
4. Connect to Neobotix WiFi
5. Open Remina Remote Desktop
6. Log onto Leonart VNC through Remina Remote Desktop
7. Log onto Neobotix VNC and check if ROS Master has been set
8. if not:
   - 8.1. On leonart:
     1. log onto leonart and verify that a roscore instance is running;

2. if not, start one (roscore) or execute ROS Start Bay 6 (lies on desktop);

3. Load map via command execution: roslaunch rtl_leonart navi.launch env_map:=/home/leonart/RTL/maps/hsrt/2019-06-09.2.yaml;

4. Turn on camera via command execution: roslaunch rtl_camera front_d435 launch filters:=point cloud;

5. Run projection_to_obstacles package: rosrun projection_to_obstacles projection_to_obstacles.

8.2. On neobotix:

1. Execute roslaunch neo_mpo_700 bringup.launch.

Check current SICK sensor emergency stop distance settings:

1. in leonarts rosenv (rosenv leonart), do rosparam list | grep neo;
2. /neo_sick_field/field may or may not show up. if it doesnt, it hasnt been set yet;
3. rosparam get /neo_sick_field/field/10 to get its current value (park, 10 (cm), nil means 0 cm).

Disable park mode:

1. to see all available options: rosservice list | grep neo;
2. to set the emergency stop distance, simply call the corresponding service e.g.: to set it to 10 cm, do rosservice call /neo_sick_field/10.

On the PC with system, on which setup for robot was done:

Run in terminal:

```
$ rosenv leonart
$ RViz
```

For appropriate visualization, it is better to open the dutchman_octomap.RViz file there.

The next command is used when it is necessary to clean Octomap: rosservice call /octomap_server/reset.

In case if you need to change parameters, you can use this command: rosrun rqt_reconfigure rqt_reconfigure.

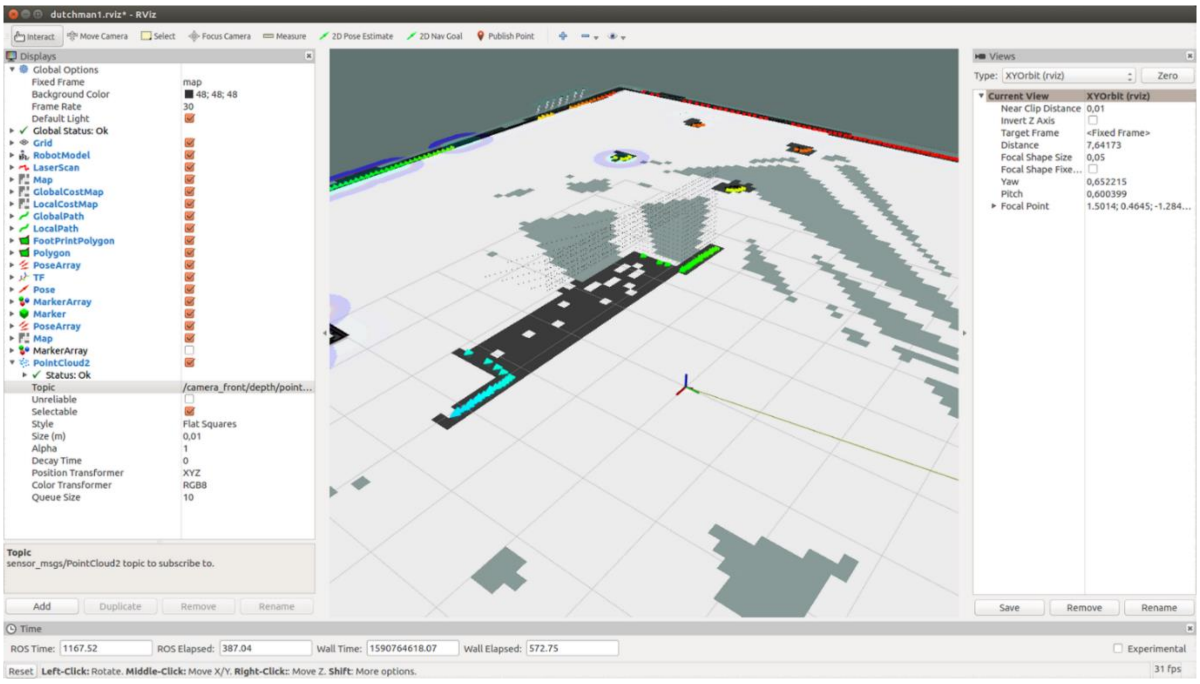
3.7. Results of solution testing

This section will demonstrate and analyze the result of the system before and after the changes. Situations of the robot’s behavior will be considered when the obstacle is a table (an object that is not completely at the level of the robot’s lasers) and an object that is located below the level of the robot’s lasers.

The original navigation system of the mobile robot was based on obstacle recognition using only lasers. The system, complemented by a depth camera, can detect obstacles that are above and below the laser level. Thus, the main problem of the “table” was solved. The new system retained all the properties of the previous one and only improved it.

The original system, which relied only on laser scanners, saw objects only at the laser level. In most cases, this was enough. However, some objects may have more complex shapes. A typical example of such an object is a table. This problem is also referred to as a table problem and is illustrated in Figure 3.9.

Also, the approach using only laser scanners does not allow the recognition of objects below the lasers' level. As shown in Fig. 3.10 and Fig. 3.11.



```
Global Options
  Fixed Frame            map
  Background Color       48; 48; 48
  Frame Rate             30
  Default Light          ✓
Global Status: Ok
Grid                     ✓
RobotModel               ✓
LaserScan                ✓
Map                      ✓
GlobalCostMap            ✓
LocalCostMap             ✓
GlobalPath               ✓
LocalPath                ✓
FootPrintPolygon         ✓
Polygon                  ✓
PoseArray                ✓
TF                       ✓
Pose                     ✓
MarkerArray              ✓
Marker                   ✓
PoseArray                ✓
Map                      ✓
MarkerArray              ☐
PointCloud2              ✓
  Status: Ok
  Topic                  /camera_front/depth/point...
  Unreliable             ☐
  Selectable             ✓
  Style                  Flat Squares
  Size (m)               0,01
  Alpha                  1
  Decay Time             0
  Position Transformer   XYZ
  Color Transformer      RGB8
  Queue Size             10

Topic
sensor_msgs/PointCloud2 topic to subscribe to.

Type: XYOrbit (rviz)     Zero
Current View             XYOrbit (rviz)
  Near Clip Distance     0,01
  Invert Z Axis          ☐
  Target Frame           <Fixed Frame>
  Distance               7,64173
  Focal Shape Size       0,05
  Focal Shape Fixe...    ☐
  Yaw                    0,652215
  Pitch                  0,600399
  Focal Point            1.5014; 0.4645; -1.284...

ROS Time: 1167.52   ROS Elapsed: 387.04   Wall Time: 1590764618.07   Wall Elapsed: 572.75   ☐ Experimental
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Move Z. Shift: More options.   31 fps
```

Fig. 3.9. Table legs detection

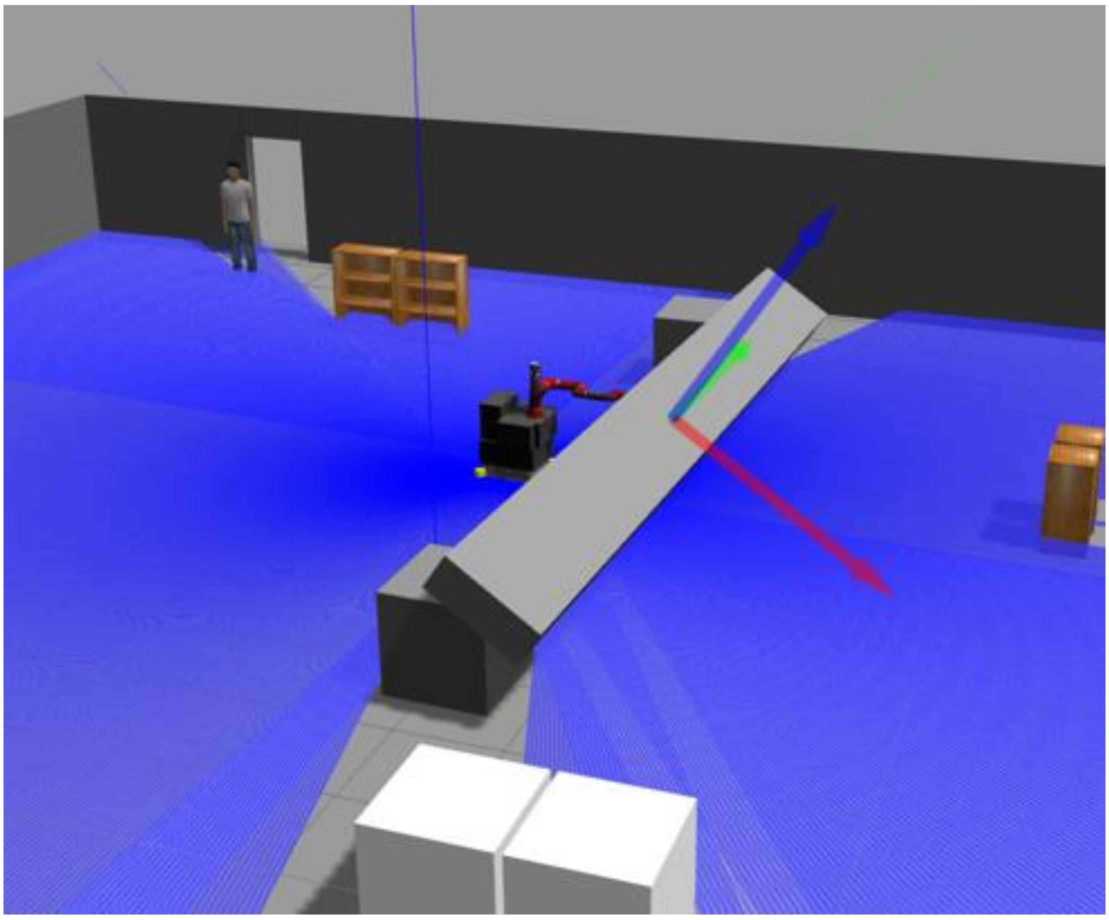


Fig. 3.10. The “table” problem

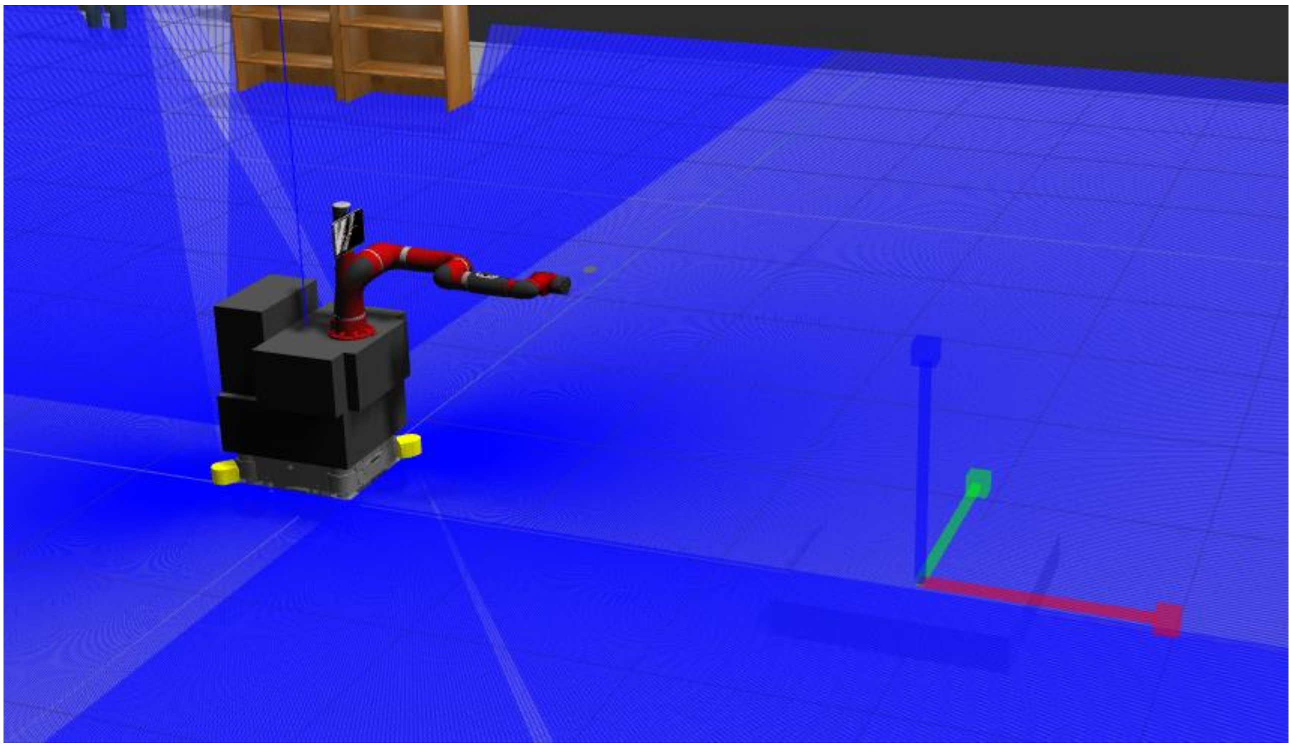


Fig. 3.11. The object below the level of lasers

As a result of applying the new system of path planning and obstacles detection, the solution to the above problems was achieved.

The experiments were carried out both in simulation and in real space. Several options for the movement of the robot were considered. If the robot encounters an obstacle on its way in the form of an object, some parts of which are not completely located at the level of the robot's lasers, this object must be recognized by the robot, and the robot must not crash into it. The second task was to find a way to avoid such an object, if possible. Both options have been successfully tested in real-time on the mobile robot Leonart. Moreover, experiments were carried out with the lasers turned off in order to test the operation of the algorithm included in the package projection_to_obstacles.

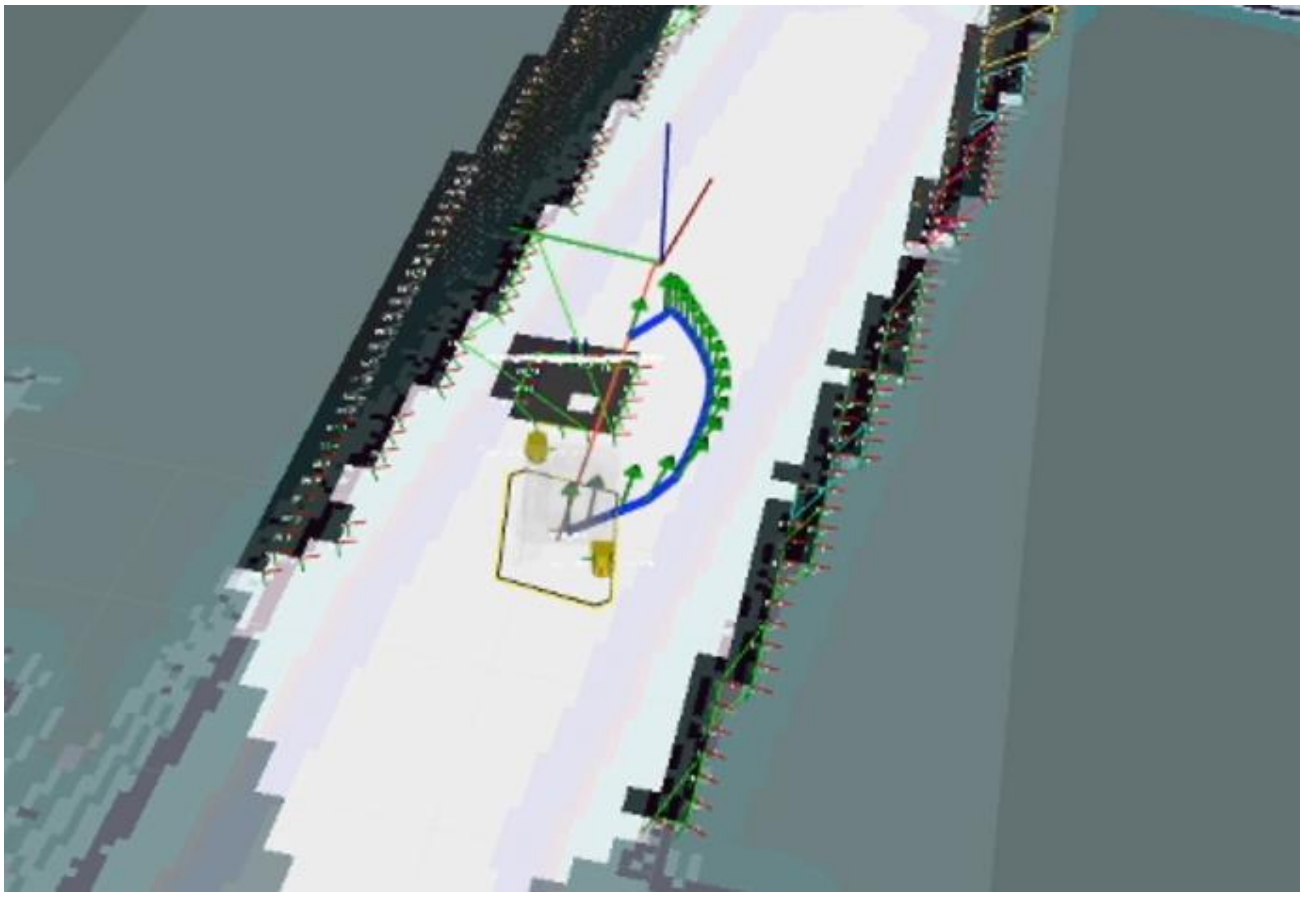


Fig. 3.12. Right path plan using projection_to_obstacles

Moreover, experiments were carried out with the lasers turned off in order to test the operation of the algorithm included in projection_to_obstacles. For an isolated system check, in one of the tests, the lasers were turned off. Thus, it was verified that the new local navigation system manages to avoid obstacles without relying on the data received from the lasers.

The image shows that the optimal path has been chosen to avoid the obstacle. The planned path of the robot’s movement is shown in blue and green arrows. Before applying the new system, the robot would simply drive straight from the start point to the endpoint. While now, the “table” object has been recognized, and moreover, the robot has successfully avoided this obstacle.

3.8. Conclusions to the third section

The third section presents the developed system and its configuration. The instruction of start of the developed product in simulation and on mobile work is given. The results of system testing are also presented, which show the successful avoidance of interference by a mobile robot.

Specific onfiguration parameters were chosen to set up the navigation stack properly. The navigation stack of the robot is consisting of amcl, gmapping, and move_base packages.

It has become possible to generate a pointcloud relative to the environment, using a depth-sensing camera to calculate the distance to objects.

Because the density of the received data in the form of a pointcloud is too high for further processing, a downsample VoxelGrid filter was used, which reduces the density of the point cloud. VoxelGrid belongs to the PCL library.

Another problem was the removal of information about unnecessary objects in the camera's field of view. These include the floor, ceiling, parts of the robot (such as a manipulator). The PassThrough filter from the PCL library was used to solve this problem.

The next step is to process the filtered data using OctoMap. As a result, an octree is generated.

A top-down projection is created from the octree generated in the previous step.

The resulting projection must be processed and converted into polygonal obstacles. Only then they will be marked by teb_local_planner as obstacles.

The developed system was successfully implemented and tested both in the Gazebo simulation and in the researche mobile robot. The path with obstacles will be completed without collisions.

РОЗДІЛ 4
ЕКОНОМІЧНИЙ РОЗДІЛ

При розробці програмного забезпечення важливими етапами є визначення трудомісткості розробки ПЗ, розрахунок витрат на створення програмного продукту і аналіз ринку збуту розробленого програмного забезпечення.

4.1. Визначення трудомісткості розробки програмного забезпечення

Початкові дані:

1) передбачуване число операторів програми – 1000;
2) коефіцієнт складності програми – 1,6;
3) коефіцієнт корекції програми в ході її розробки – 0,4;
4) годинна заробітна плата програміста, грн/год - 70;
5) коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,2;
6) коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1,5;
7) вартість машино-години ЕОМ, грн/год - 15.

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_u + t_a + t_n + t_{o\,тл} + t_д \text{, людино-годин,} \qquad (3.1)$$

де t_o - витрати праці на підготовку й опис поставленої задачі (приймається 50);

$t_и$ - витрати праці на дослідження алгоритму рішення задачі;

$t_а$ - витрати праці на розробку блок-схеми алгоритму;

$t_п$ - витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ - витрати праці на налагодження програми на ЕОМ;

$t_д$ - витрати праці на підготовку документації.

Умовне число операторів (підпрограм):

$$Q = q \times C \times (1+ p)\text{ , людино-годин} \qquad (3.2)$$

де q - передбачуване число операторів,

C - коефіцієнт складності програми,

p - коефіцієнт кореляції програми в ході її розробки.

Q передбачуване число операторів (q = 1000).

С коефіцієнт складності програми. Коефіцієнт складності завдання З характеризує відносну складність програми по відношенню до так званої типової задачі, що реалізує стандартні методи рішення, складність якої прийнята рівною одиниці (величина С лежить в межах від 1,25 до 2). Для даного програмного продукту, з урахуванням великої кількості і різноманітності оброблюваної інформації і складності складання звітів, коефіцієнт складності завдання візьмемо 1,6.

$$Q = q * C * (1 + p), \qquad (3.2)$$

де q - передбачуване число операторів;

С - коефіцієнт складності програми;

р - коефіцієнт корекції програми в ході її розробки.

С коефіцієнт складності програми. Коефіцієнт складності завдання З характеризує відносну складність програми по відношенню до так званої типової задачі, що реалізує стандартні методи рішення, складність якої прийнята рівною одиниці (величина С лежить в межах від 1,25 до 2). Для даного програмного продукту, з урахуванням великої кількості і різноманітності оброблюваної

інформації і складності складання звітів, коефіцієнт складності завдання візьмемо 1,6.

Р коефіцієнт корекції програми в ході її розробки. Коефіцієнт корекції програми р - збільшення обсягу робіт за рахунок внесення змін до алгоритму або програму за результатами уточнення постановок. В даному випадку програма вимагала численних доробок. З урахуванням цього візьмемо коефіцієнт рівний 0,4.

$$Q=1000*1,6*(1+0,4)= 2240, \text{ людино-годин.} \quad (3.3)$$

Витрати праці на вивчення опису задачі $t_и$ визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u=\frac{Q*B}{(75\ldots85)*k}, \text{ людино-годин,} \quad (3.4)$$

де В - коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k - коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності.

В коефіцієнт збільшення витрат праці внаслідок недостатнього опису завдання. Коефіцієнт збільшення витрат праці в залежності від складності завдання приймається від 1,25 до 1,5, внаслідок недостатнього опису рішення задачі приймемо В = 1,3.

К коефіцієнт кваліфікації програміста, який визначається від стажу роботи за даною спеціальністю. К = 1,5.

$$t_u=\frac{2240*1,3}{80*1,5}=\frac{2912}{112,5}=24,27, \text{ людино-годин.} \quad (3.5)$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a=\frac{Q}{(20\ldots25)*k}, \text{ людино-годин.} \qquad (3.6)$$

$$t_a=\frac{2240}{22*1{,}5}=67{,}879, \text{ людино-годин.} \qquad (3.7)$$

Витрати на складання програми по готовій блок-схемі:

$$t_n=\frac{Q}{(20\ldots25)*k}, \text{ людино-годин} \qquad (3.8)$$

$$t_n=\frac{2240}{23*1{,}5}=64{,}928, \text{ людино-годин.} \qquad (3.9)$$

Витрати праці на налагодження програми на ЕОМ:

- за умови автономного налагодження одного завдання:

$$t_{отл}=\frac{Q}{(4\ldots5)*k}, \text{ людино-годин,} \qquad (3.10)$$

$$t_{отл}=\frac{2240}{5*1{,}5}=298{,}667, \text{ людино-годин,} \qquad (3.11)$$

- за умови комплексного налагодження завдання:

$$t^k_{отл}=1{,}5*t_{отл}, \text{ людино-годин.} \qquad (3.12)$$

$$t^k_{отл}=1{,}5*298{,}667=448, \text{ людино-годин.} \qquad (3.13)$$

Витрати праці на підготовку документації:

$$t_д = t_{др} + t_{до}, \text{ людино-годин,} \tag{3.14}$$

де $t_{др}$ - трудомісткість підготовки матеріалів і рукопису.

$$t_{др} = \frac{Q}{15..20*k}, \text{ людино-годин.} \tag{3.15}$$

$$t_{др} = \frac{2240}{18*1,5} = 82,963, \text{ людино-годин,} \tag{3.16}$$

$t_{до}$ - трудомісткість редагування, печатки й оформлення документації

$$t_{до} = 0,75 * t_{др}, \text{ людино-годин} \tag{3.17}$$

$$t_{до} = 0,75*82,963 = 62,222, \text{ людино-годин.} \tag{3.18}$$

$$t_д = 82,963 + 62,222 = 145,185, \text{ людино-годин.} \tag{3.19}$$

Отримуємо трудомісткість розробки програмного забезпечення:

$$t = 50 + 24,27 + 67,879 + 448 + 145,185 = 735,334, \text{ людино-годин.} \tag{3.20}$$

4.2. Витрати на створення програмного забезпечення

Витрати на створення ПЗ Кпо включають витрати на заробітну плату виконавця програми Зз/п і витрат машинного часу, необхідного на налагодження програми на ЕОМ

$$К_{по}=З_{зп}+З_{мв}, грн. \tag{3.21}$$

Заробітна плата виконавців визначається за формулою:

$$З_{зп}=t*C_{пр}, грн, \tag{3.22}$$

де: t - загальна трудомісткість, людино-годин;

$С_{пр}$ - середня годинна заробітна плата програміста, грн/година

$$З_{зп}=735{,}334*70=51473{,}38, грн. \tag{3.23}$$

Вартість машинного часу, необхідного для налагодження програми на ЕОМ:

$$З_{мв}=t_{отл}*С_{мч}, грн, \tag{3.24}$$

де $t_{отл}$ - трудомісткість налагодження програми на ЕОМ, год.

$С_{мч}$ - вартість машино-години ЕОМ, грн/год.

Визначені в такий спосіб витрати на створення програмного забезпечення є частиною одноразових капітальних витрат на створення АСУП.

$$З_{мв}=298{,}667*15=4480, грн. \tag{3.25}$$

$$К_{по}=51473{,}38+4480=55953{,}38, грн. \tag{3.26}$$

Очікуваний період створення ПЗ:

$$T=\frac{t}{B_k*F_p}, міс, \tag{3.27}$$

де B_k - число виконавців;

F_p - місячний фонд робочого часу (при 40 годинному робочому тижні F_p=176 годин).

$B_k = 1$

$$T=\frac{735{,}334}{1*176}=4{,}2,\quad \text{міс.} \tag{3.28}$$

4.3. Маркетингові дослідження ринку збуту розробленого програмного продукту

У цьому підрозділі буде проаналізовано та досліджено ринок збуту та потенційних покупців нової системи навігації мобільного робота. Метою такого дослідження є реалізація створеного програмного продукту промисловим підприємствам різних галузей промисловості, торговим підприємствам, медичним та навчальним закладам, спеціальним організаціям та установам, які зацікавлені в залученні мобільних роботів для покращення умов праці.

Однією з багатьох тенденцій в робототехніці є перехід від дистанційно керованих систем, які вимагають постійної участі людини для виконання всіх дій робота, до автономних систем, в яких оператор вказує лише кінцеві та проміжні цілі.

Навігація залишається головною проблемою усіх існуючих мобільних пристроїв, що рухаються самостійно. Для успішної навігації у просторі, система робота повинна вміти планувати шлях, правильно інтерпретувати інформацію про навколишній світ, отриману від датчиків. Більше того, робот повинен контролювати параметри руху, власні координати та мати можливість адаптуватися до змін навколишнього середовища.

Одним із найскладніших аспектів автономної навігації мобільних роботів на відкритому повітрі є надійність. Тобто мобільний робот повинен мати можливість безпечно добиратись до місця призначення щоразу, не лише уникаючи зіткнень із перешкодами та людьми навколо нього, а й успішно проїжджаючи складні стежки. Мобільні роботи повинні бути оснащені різними датчиками, які передаватимуть дані про навколишнє середовище. При використанні лише лазерних сканерів для мобільних робіт предмети, що перевищують або нижче рівня лазерів, залишаться перешкодою для робота. Перешкоди вище або нижче рівня лазерного сканера не розпізнаються. Таким чином, маршрут побудований неправильно. Отже, робот стикається з перешкодами.

Результатом кваліфікаційної роботи є створення нової системи розпізнавання середовища, яка використовує дані отримані з камери глибини і оброблює їх. Використання такої системи дозволяє розпізнавати та уникати перешкоди, які не обов’язково повинні знаходитись на одному рівні з рівнем лазерів робота.

Сьогодні розробка мобільних роботів є актуальним напрямом у світі і активно фінансується та має дуже широкий сегмент збуту в індустрії. До таких напрямів входить машинобудування, до такої галузі входить близько 30% світових поставок. Роботи потрібні для модернізації існуючих підприємств і інвестицій у нові потужності у виробництві. Також великим напрямом є сервісна робототехніка, яка поділяється на персональну та професійну. До персональної зазвичай відносять роботів для домашніх справ, розважальних роботів, роботів-асистентів для людей похилого віку чи людей з інвалідністю, тощо. Професійна робототехніка поділяється на польову робототехніку, професійне прибирання, для моніторингу та експлуатації, будівництва. Існує медична робототехніка, воєнна, рятувальна, силова, мобільна та інша.

Одними з найбільш помітних виробників промислових роботів на міжнародному ринку є FANUC, Yaskawa, ABB, Kuka, Universal Robots (Mobile Industrial Robots), NEOBOTIX, Denso тощо. На мою думку, саме компанія Mobile Indastrial Robots може стати потенційним покупцем нової системи, так як основними продуктами компанії є роботи для логістики у приміщеннях.

Компанія активно займається покращенням навігації мобільних роботів та їх керуванням.

Напрямки, які до 2023 року складуть основу ринку сервісної робототехніки:

– логістичні системи (включають логістику всередині приміщень, безпілотні і повітряні засоби доставки поза приміщеннями);

– роботи для обслуговування клієнтів;

– промислові екзоскелети;

– роботи для домашніх завдань (персональні помічники).

Основними складнощами під час впровадження такого типу роботів - завищені очікування замовників і перебільшення здібностей штучного інтелекту. Вартість обслуговування часто обумовлена труднощами, що виникають при роботі з клієнтами, і прихованими витратами, пов'язаними з низьким рівнем обслуговування робототехніки.

Крім ринкових викликів галузь стикається також з технологічними труднощами. Більшість цих проблем обумовлено взаємодією роботів і людей, автономністю пересувань, машинним зором і розпізнаванням мови, ІІ і машинним навчанням, обмеженнями, пов'язаними з хмарною робототехнікою, а також безпекою і стандартами.

До завдань логістичних систем входить управління потоком товарів, їх перевезенням, обробленням та упаковкою. Всі логістичні системи вимагають мобільності в закритому чи у відкритому просторі. Основною перевагою роботехнічних рішень для логістики є скорочення потреби в ручній праці, зменшення людського фактору, безпека на робочому місці і велика точність при інвентаризації. Створена система є найбільш актуальною саме в подібних завданнях. Роботи для доставки в приміщення можуть вимагати значної інфраструктури, наприклад установки системи відкриття дверей або ескалаторів для переміщення з поверху на поверх. Ці інвестиції окупляться значно швидше, якщо робот використовується 24 години щодня. Більш того, вартість впровадження роботів в промисловості падає, що веде до підвищення рентабельності роботів і зниження порогу входу в галузь.

Розробку нової системи навігації мобільного робота можна віднести до складної програмної продукції, яка потребує спеціального налагодження. В цьому випадку найчастіше програмне забезпечення розробляється за замовленням споживача.

4.4. Оцінка економічної ефективності впровадження програмного забезпечення

У цьому розділі буде проведено аналіз ефективності впровадження створеної нової системи навігації мобільного робота на підприємстві.

За даними Barclays Research, середня собівартість робіт, виконуваних роботом, становить € 6 на годину. За деякими даними, аналогічний показник в два рази менше - близько € 3 за годину. Аналогічна робота, виконувана людиною, оцінюється по-різному в різних країнах і регіонах: € 40 - в Німеччині, € 12 - в США, € 11 - в східній Європі і € 9 - в Китаї.

Через наявність лише удосконаленої системи та відсутність програмного забезпечення, яке можна впровадити, неможливо розрахувати економічний ефект, в якому обсязі необхідні інвестиції, який термін окупності і очікуємий прибуток.

Тому розглядається тільки соціальний ефект.

За допомогою провадження удосконаленої системи до планування шляху мобільним роботом для навігації може:

- запобігти виникненню нещасного випадку на підприємстві;
- збільшити функціонал та можливості руху мобільного робота;
- підвищити продуктивність праці;
- скоротити кількість працівників.

Можна зробити висновок, що впровадження нової системи повинно мати позитивний економічний ефект тому, що дана розробка є актуальною та необхідною для широкого сектору робототехніки як напряму, з актуальним соціальним ефектом.

Висновок:

У результаті виконання кваліфікаційної роботи було створено нову систему для планування руху мобільного робота. У даному економічному розділі було визначено трудомісткість на розробку додатку, що складає 735.3 люд-год, проведено підрахунок вартості роботи по створенню описаної системи, які склали 55953,4 грн. та розраховано час на його створення – 4,7 міс. Було проаналізовано та досліджено ринок збуту та потенційних покупців нової системи навігації мобільного робота. Визначено, що саме компанія Mobile Indastrial Robots може стати потенційним покупцем нової системи, так як основними продуктами компанії є роботи для логістики у приміщеннях та компанія активно займається покращенням навігації мобільних роботів та їх керуванням. Створена система є найбільш актуальною для логістичних завдань, де дуже важливе вправне мобільне переміщення робота та уникнення різноманітних перешкод на шляху. Її впровадження повинно також мати позитивний соціальний ефект.

CONCLUSIONS

The aim of the research is to create a real-time obstacle recognition and avoidance system, using sensors to ensure the road of the path without collisions with objects that are not on the same level as the lasers. The research was carried out on the mobile robot “Leonart” of the RT Lions team at Reutlingen University.

During the work, all tasks of the research, given in the introduction were solved. The mobile robot requires a map of the environment and the ability to interpret that representation. The camera can be represented as the eyes of the robot, and the images taken from the camera are useful for recognizing the environment around the robot. For example, object recognition using a camera image, facial recognition, a distance value obtained from the difference between two different images using two cameras (stereo camera), mono camera visual SLAM, color recognition using information obtained from an image, and object tracking are very useful. Laser SICK S300 is used as the main environment scanner for navigation. The other type of sensor is a depth camera. The depth camera is represented by the Intel Realsense D435 camera. Using a dual camera rig provides a depth image that can be converted to point cloud. This conversion is done by ROS Realsense camera node.

There two types of mobile robot navigation: 2D and 3D. 3D is applicable for flying robots when the topic of this research is 2D navigation. The robot, the subject of the research, only able to move with wheels in 2D space. Action`s implementation is provided in the move_base package. The move_base node provides a ROS interface for configuring, running, and interacting with the navigation stack on a robot. The given goal in the world needs to be reached with a mobile base. A global and local planner are linked together by the move_base node to accomplish its global navigation task. In the nav_core package nav_core::BaseGlobalPlanner interface is specified and global planner is adhesive to it. The same situation with any local planner. Any local planner adhering to the nav_core::BaseLocalPlanner interface from the nav_core package. The node maintains two costmaps, from the global and local planners, which are used to accomplish navigation tasks. Timed Elastic Band optimizes the robot`s trajectory with an execution time locally, separation from obstacles, and compliance with kinodynamic constraints at runtime. The teb_local_planner package performs a plugin to the base_local_planner of the 2D navigation stack.

As a result of the work, the existing system of recognition and avoidance of obstacles was expanded. Prior to that, the system used only odometry and information obtained from laser scanners, without obtaining data from other sources of environmental information. It has become possible to generate a point cloud relative to the environment, using a depth-sensing camera to calculate the distance to objects. Because the density of the received data in the form of a point cloud is too high for further processing, a downsample VoxelGrid filter was used, which reduces the density of the point cloud.

Another problem was the removal of information about unnecessary objects in the camera's field of view. These include the floor, ceiling, parts of the robot (such as a manipulator). The PassThrough filter from the PCL library was used to solve this problem. After this step, octomap was used and octree is generated. OctoMap provides converting point cloud data to occupancy grid octree. Then these occupied octants(voxels) are projected to the floor, providing an image of the occupied area. The resulting projection must be processed and converted into polygonal obstacles. Only then they will be marked by teb_local_planner as obstacles.

The developed system was successfully implemented and tested both in the Gazebo simulation and in the research mobile robot. The path with obstacles will be completed without collisions. The paper presents the obtained test results.

REFERENCES

1. Бобровский С. Навигация мобильных роботов (в 3 ч.). Ч. 1 // PC Week/RE. – 2004. – №9. – С.52.

2. Минин А. А. Навигация и управление мобильным роботом, оснащенным лазерным дальномером: диссертация кандидата технических наук 05.02.05 / А. А. Минин; Москва, 2008. – 182 с.

3. Siciliano B. Springer handbook of robotics / B.Siciliano, O.Khatib. – Springer-Verlag Berlin, 2008. -P. 477-580.

4. Nister D. Visual odometry for ground vehicle applications / D. Nister, O.Naroditsky, J.Bergen // J. of Field Robotics. – 2006. – Vol.23(1). -P. 3-20.

5. Xu J, Robust stereo visual odometry for autonomous rover / J.Xu, M.Shen, W.Wang, L.Yang // 6th WSEAS Internat. Conf. on Signal, Speech and Image Processing. – 2006.

6. Hirschmuller H. Fast, unconstrained Camera Motion Estimation from stereo without Tracking and Robust Statistics / H.Hirschmuller // 7th Internat. Conf. on Control, Automation, Robotics and Vision. – 2002. – T.2. – P.1099-1104.

7. Bota S. Camera Motion Estimation Using Monocular and Stereo-Vision / S.Bota, S.Nedevschi // 4th Internat. Conf. on Intelligent Computer Communication and Processing. – 2008. P.275-278.

8. Бройнль Т. Встраиваемые робототехнические системы: проектирование и применение мобильных роботов со встроенными системами управления / Т. Бройнль, под редакцией В.Е. Павловского. – М. – Ижевск: Ижевский институт компьютерных исследований, 2012. – 520 с.

9. Puttkamer E. Autonome Mobile Roboter / E. Puttkamer, E. Von. – Lecture notes. Univ. Kaiserslautern, Fachbereich Informatik: 2000.

10. Lumelsky V. Dynamic Path Planning for a Mobile Automation with Limited Information on the Environment / V. Lumelsky, A. Stepanov: IEEE Transactions on Automatic Control. – Vol. 31. – 1986 – pp. 1058- 1063.

11. Kamon, I. Sensory-Based Motion Planning with Global Proofs / I. Kamon, E. Rivlin: IEEE Transactions on Robotics and Automation. – Vol. 13. – 6 Dec. – 1997. – pp. 814-822

12. Bradski G. Learning OpenCV / G.Bradski, A.Kaehler // Newgen Publishing and Data Services, 2008.

13. Quigley M., Conley K., B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in ICRA Workshop on Open Source Software, 2009.

14. Matt Pharr, Randima Fernando, Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation / Addison-Wesley Professional – 3. März 2005

15. Package move_base [Electronical resource]: http://wiki.ros.org/move_base

16. C. Roesmann, W. Feiten, T. Woesch, F. Hoffmann, and T. Bertram (2012) "Trajectory modification considering dynamic constraints of autonomous robots". Proc. 7th German Conference on Robotics, Germany, Munich, pp 74–79.

17. OctoMap [Electronical resource]: http://octomap.github.io/

18. S. Thrun, W. Burgard, D. Fox "Probabilistic Robotics" (2005)

19. D. Fox, W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance". The Dynamic Window Approach to local control. Robotics & Automation Magazine, IEEE, 1997, pp 23 – 33

20. Alonzo Kelly. "An Intelligent Predictive Controller for Autonomous Vehicles". A previous system that takes a similar approach to control, tech. report CMU-RI-TR-94-20, Robotics Institute, Carnegie Mellon University, May 1994

21. Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard: Improving Gridbased SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling, In Proc. of the IEEE International Conference on Robotics and Automation (ICRA), 2005

22. Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard: Improved 36 Techniques for Grid Mapping with Rao-Blackwellized Particle Filters, IEEE Transactions on Robotics, Volume 23, pages 34-46, 2007

23. Brian P. Gerkey and Kurt Konolige. "Planning and Control in Unstructured Terrain ". Discussion of the Trajectory Rollout algorithm in use on the LAGR robot. (2008)

24. http://wiki.ros.org/amcl

25. http://wiki.ros.org/robot_pose_ekf

26. Sebastian Thrun and others. Stanley: The Robot that Won the DARPA Grand Challenge. Journal of Field Robotics 23(9), 661–692 (2006)

27. Maryum F. Ahmed. Development of a stereo vision system for outdoor mobile Robots. Abstract of Thesis Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Master of Science. 2006. pp 78.

28. Kok Seng Chong , Lindsay Kleeman, Mobile Robot Map Building from an Advanced Sonar Array and Accurate Odometry. 1996.

29. M. A. Martinez and J. L. Martinez. The dual-frequency sonar system of the 35 mobile robot RAM-2. Journal Robotica Volume 22 Issue 3, June 2004. pp. 263 – 270

30. Lindsay Kleeman, Roman Kuc. Mobile Robot Sonar for Target Localization and Classification. The International Journal of Robotics Research (Impact Factor: 2.86). 01/1995; 14:295-318

31. Kiyoshi Okuda, Masamichi Miyake, Hiroyuki Takai, Keihachiro Tachibana. Obstacle arrangement detection using multichannel ultrasonic sonar for indoor mobile robots. Artificial Life and Robotics. September 2010, Volume 15, Issue 2, pp 229- 233.

32. Heale A., Kleeman L. Fast target classification using sonar. Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on (Volume:3). 2001. 1446 - 1451 vol.3.

33. Teruko YATA, Akihisa OHYA, Shinichi YUTA. A Fast and Accurate SonarRing Sensor for a Mobile Robot. Proceedings of the 1999 IEEE International Conference on Robotics and Automation, May, 1999, pp. 630-636.

34. Antoni Burguera, Yolanda Gonzalez and Gabriel Oliver. Sonar Sensor Models and Their Application to Mobile Robot Localization. Sensors (Basel). 2009; 9(12): 10217–10243.

35. Elfes A. A sonar-based mapping and navigation system. Robotics and Automation. Proceedings. 1986 IEEE International Conference on (Volume:3). Apr 1986. pp. 1151 — 1156.

36. Bruno Siciliano, Oussama Khatib. Springer Handbook of Robotics. SpringerVerlag Berlin Heidelberg. 2008. 1611 c.

37. S. Quinlan and O. Khatib, “Elastic bands: Connecting path planning and control,” in Proc. IEEE International Conference on Robotics and Automation (ICRA), 1993, pp. 802–807.

38. C. Rasmussen, Y. Lu, and M. Kocamaz, “Appearance contrast for fast, robust trail-following,” in Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), October 2009, pp. 3505 – 3512.

39. T. Kuhnl, F. Kummert, and J. Fritsch, “Monocular road segmentation using slow feature analysis,” in IEEE Intelligent Vehicles Symposium (IV), june 2011, pp. 800 – 806.

40. P. Santana, N. Alves, L. Correia, and J. Barata, “Swarm-based visual saliency for trail detection,” in Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), oct. 2010, pp. 759 –765.

41. H. Kong, J.-Y. Audibert, and J. Ponce, “General road detection from a single image,” IEEE Transactions on Image Processing, vol. 19, no. 8, pp. 2211 – 2220, August 2010.

42. O. Miksik, “Rapid vanishing point estimation for general road detection,” in Proc. IEEE International Conference on Robotics and Automation (ICRA), May 2012.

43. C.-K. Chang, C. Siagian, and L. Itti, “Mobile robot monocular vision navigation based on road region and boundary estimation,” in Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Oct 2012, bb, pp. 1043–1050.

44. C. Siagian, C.-K. Chang, R. Voorhies, and L. Itti, “Beobot 2.0: Cluster architecture for mobile robotics,” Journal of Field Robotics, vol. 28, no. 2, pp. 278–302, March/April 2011.

45. G. Bradski, “Open source computer vision library,” 2001. [Online]. Available: http://opencv.willowgarage.com

46. Ronald C. Arkin. Intelligent Robotics and Autonomous Agents. Nourbakhsh, 2004. – 321ł.

47. Lumelsky V. Dynamic Path Planning for a Mobile Automation with Limited Information on the Environment / V. Lumelsky, A. Stepanov: IEEE Transactions on Automatic Control. – Vol. 31. – 1986 – pp. 1058- 1063.

48. Puttkamer E. Autonome Mobile Roboter / E. Puttkamer, E. Von. – Lecture notes. Univ. Kaiserslautern, Fachbereich Informatik: 2000.

49. Borenstein J. Navigating Mobile Robots: Senfors and Techniques / J. Borenstein, H. Everett, L. Feng. – Wellesley MA – AK Peters: 1998. – 225 p.

50. Koren Y. Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation / Y. Koren, J. Borenstein // Proceedings of the IEEE Conference on Robotics and Automation. – Sacramento CA, – 1991. – pp. 1398-1404.

51. Arbib M. Depth and Detours: An Essay on Visually Guided Behavior M. Arbib, A. Hanson (Eds.), Vision, Brain and Cooperative Computation / M. Arbib, A. Hanson. – Cambridge MA. – MIT Press. 1987.

52. Choset H. Principles of Robot Motion: Theoty, Algorithms, and Implementations / H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, S. Thrun. – Cambridge MA – MIT Press: 2005. – 603 p.

53. Siegward R Introduction to Autonomous Mobile Robots / R. Siegward, R. Nourbakhsh - Cambridge MA: MIT Press 2004. – 321 p.

54. Borenstein J Where am I. Sensors and methods for mobile robot positioning / J/ Borenstein, H.R. Everett, L. Feng - Prepared by the University of Michigan. 1996 - 282 p.

55. Cherroun L., Boumehraz M. Designing of Goal Seeking and Obstacle Avoidance Behaviors for a Mobile Robot Using Fuzzy Techniques // Journal of Automation and Systems Engineering (JASE). 2012. Vol. 6, Issue 4. P. 164–171.

56. Ersson T., Hu X. Path Planning and Navigation of Mobile Robots in Unknown Environments // IEEE Journal of Robotics and Automation. 2010. Issue 6. P. 212–228.

57. Ventorim B. G., Dal Poz W. R. Performance Evaluation of GPS and GLONASS Systems, Combined and Individually, in Precise Point Positioning // Boletim de Ciencias Geodesicas. 2016. Vol. 22, Issue 2. P. 265–281. doi: https://doi.org/10.1590/s1982-21702016000200015

58. Kaemarungsi K., Krishnamurthy P. Modeling of indoor positioning systems based on location fingerprinting // IEEE INFOCOM 2004. 2004. doi: https://doi.org/10.1109/infcom.2004.1356988

59. Dhillon S. S., Chakrabarty K. Sensor placement for effective coverage and surveillance in distributed sensor networks // 2003 IEEE Wireless Communications and Networking, 2003. WCNC 2003. 2003. doi: https://doi.org/10.1109/wcnc.2003.1200627

60. Rosmann C., Hoffmann F., Bertram T. Planning of multiple robot trajectories in distinctive topologies // 2015 European Conference on Mobile Robots (ECMR). 2015. pp. 1-6.

APPENDIX A

SOURCE CODE

Projection_to_obstacles.cpp

```
#include <ros/ros.h>
#include <nav_msgs/OccupancyGrid.h>
#include <iostream>
#include "../include/projcection_to_obstacles/OccupancyGridPositions.h"
#include <tf/LinearMath/Transform.h>
#include <tf/transform_broadcaster.h>
#include <costmap_converter/ObstacleArrayMsg.h>
#include <geometry_msgs/Polygon.h>

void ProjectionHanler(const nav_msgs::OccupancyGridConstPtr &msg);

void CostmapHanler(const nav_msgs::OccupancyGridConstPtr &msg);

void GridToTF(OccupancyGridPositions *grid);

// Uncomment and uncomment usages for debug
tf::TransformBroadcaster *br;

ros::Publisher *obstaclePublisher;
std::vector<ros::Publisher *> polygonPublishers;

int main(int argc, char **argv)
{
	ros::init(argc, argv, "projection_to_obstacles");
	ros::NodeHandle nodeHandle;
	ros::Subscriber localCostmap_sub =
nodeHandle.subscribe("/move_base/local_costmap/costmap", 1, &CostmapHanler);
	ros::Subscriber projection_sub = nodeHandle.subscribe("projected_map", 1,
&ProjectionHanler);
	ros::Publisher pub = nodeHandle.advertise<costmap_converter::ObstacleArrayMsg>(
	"/move_base/TebLocalPlannerROS/obstacles", 1);
	obstaclePublisher = &pub;



	ros::Publisher ppub1 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/one", 1);
	polygonPublishers.push_back(&ppub1);
	ros::Publisher ppub2 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/two", 1);
	polygonPublishers.push_back(&ppub2);
```

```
        ros::Publisher ppub3 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/three", 1);
        polygonPublishers.push_back(&ppub3);
        ros::Publisher ppub4 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/four", 1);
        polygonPublishers.push_back(&ppub4);
        ros::Publisher ppub5 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/five", 1);
        polygonPublishers.push_back(&ppub5);
        ros::Publisher ppub6 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/six", 1);
        polygonPublishers.push_back(&ppub6);
        ros::Publisher ppub7 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/seven", 1);
        polygonPublishers.push_back(&ppub7);
        ros::Publisher ppub8 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/eight", 1);
        polygonPublishers.push_back(&ppub8);
        ros::Publisher ppub9 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/nine", 1);
        polygonPublishers.push_back(&ppub9);
        ros::Publisher ppub10 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/ten", 1);
        polygonPublishers.push_back(&ppub10);
        ros::Publisher ppub11 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/eleven", 1);
        polygonPublishers.push_back(&ppub11);
        ros::Publisher ppub12 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/twelve", 1);
        polygonPublishers.push_back(&ppub12);
        ros::Publisher ppub13 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/thirteen", 1);
        polygonPublishers.push_back(&ppub13);
        ros::Publisher ppub14 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/fourteen", 1);
        polygonPublishers.push_back(&ppub14);
        ros::Publisher ppub15 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/fifteen", 1);
        polygonPublishers.push_back(&ppub15);
        ros::Publisher ppub16 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/sixteen", 1);
        polygonPublishers.push_back(&ppub16);
        ros::Publisher ppub17 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/seventeen", 1);
        polygonPublishers.push_back(&ppub17);
        ros::Publisher ppub18 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/eighteen", 1);
        polygonPublishers.push_back(&ppub18);
        ros::Publisher ppub19 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/nineteen", 1);
```

```
        polygonPublishers.push_back(&ppub19);
        ros::Publisher ppub20 =
nodeHandle.advertise<geometry_msgs::PolygonStamped>("projectedPolygon/twenty", 1);
        polygonPublishers.push_back(&ppub20);


        tf::TransformBroadcaster brLocal;
        br = new tf::TransformBroadcaster();
        std::cout << "subscribed";
        ros::spin();
        return 0;
}

void ProjectionHanler(const nav_msgs::OccupancyGridConstPtr &msg)
{
        // tf::Quaternion q;
        tf::Transform projectionTransform;
        projectionTransform.setOrigin(tf::Vector3(msg->info.origin.position.x, msg-
>info.origin.position.y, 0.0));
        OccupancyGridPositions *grid = new OccupancyGridPositions(*msg, projectionTransform);
        // q.setRPY(0, 0, msg->theta);
        // transform.setRotation(q);
        // br->sendTransform(tf::StampedTransform(projectionTransform, ros::Time::now(),
"map", "projection"));
        //GridToTF(grid);
        costmap_converter::ObstacleArrayMsgPtr obstacleArrayMsg(new
costmap_converter::ObstacleArrayMsg()); //= grid->ToPointObstacles();
        obstacleArrayMsg->header.frame_id = "map";


        std::vector<geometry_msgs::Polygon> polygons = grid->ToPolygons(br);
        for (int i = 0; i < polygons.size() && i < polygonPublishers.size(); ++i)
        {
                costmap_converter::ObstacleMsg obstacle;
                obstacle.header.frame_id = "map";

                geometry_msgs::Polygon polygon;
                for (int j = 0; j < polygons[i].points.size(); ++j)
                {
                        obstacle.polygon.points.push_back(polygons[i].points[j]);
                }
                obstacleArrayMsg->obstacles.push_back(obstacle);

                geometry_msgs::PolygonStamped polygonMsg;
                polygonMsg.polygon = polygons[i];
                polygonMsg.header.stamp = ros::Time::now();
                polygonMsg.header.frame_id = "map"

                for (int j = 0; j < polygonMsg.polygon.points.size(); ++j)
                {
                        tfScalar x = polygonMsg.polygon.points[j].x;
                        tfScalar y = polygonMsg.polygon.points[j].y;
                        tf::Transform transform;
                        transform.setOrigin(tf::Vector3(x, y, 0));
                        std::stringstream ss;
                        ss << "p" << i << "_" << j;
                        br->sendTransform(tf::StampedTransform(transform, ros::Time::now(),
                "map", ss.str()));
                }
                polygonPublishers[i]->publish(polygonMsg);
        }
        obstaclePublisher->publish(obstacleArrayMsg);
}

void CostmapHanler(const nav_msgs::OccupancyGridConstPtr &msg)
{
        tf::Transform transform;
        //transform.setRotation(tf::createIdentityQuaternion());
        transform.setOrigin(tf::Vector3(msg->info.origin.position.x, msg-
>info.origin.position.y, 0.0));
        transform.setIdentity();
        // q.setRPY(0, 0, msg->theta);
        // tf::Quaternion q;
        // transform.setRotation(q);
        //br->sendTransform(tf::StampedTransform(transform, ros::Time::now(), "map",
"local"));

        int a = msg->info.height;
        //std::vector<int8_t> b = msg->data;
}

void GridToTF(OccupancyGridPositions *grid)
{
        for (int i = 0; i < grid->cells.size(); ++i)
{
        std::stringstream ss;
        ss << i << std::endl;
        tf::Transform transform;
        transform.setOrigin(tf::Vector3(grid->cells[i].x, grid->cells[i].y, 0.0));
        // br->sendTransform(tf::StampedTransform(transform, ros::Time::now(), "map",
ss.str()));
}
}

Occupancygridpositions.h

#ifndef SRC_OCCUPANCYGRIDPOSITIONS_H
#define SRC_OCCUPANCYGRIDPOSITIONS_H

#include <nav_msgs/OccupancyGrid.h>
```

```
#include <costmap_converter/ObstacleMsg.h>
#include <costmap_converter/ObstacleArrayMsg.h>
#include <tf/LinearMath/Transform.h>
#include <costmap_converter/costmap_to_polygons_concave.h>
#include <libavutil/log.h>
#include <math.h>
#include <tf/transform_broadcaster.h>

struct OccupancyGridCell
{
        int8_t value;
        double x;
        double y;
};

struct Vertex
{
        double x;
        double y;
};

bool CompareVertexX(const Vertex &a, const Vertex &b)
{
        return a.x < b.x;
}

bool CompareVertexY(const Vertex &a, const Vertex &b)
{
        return a.y < b.y;
}

class OccupancyGridPositions
{
        const int OccupiedValue = 100;

        float resolution;
        uint32_t height;
        uint32_t width;

        tf::TransformBroadcaster *br;

        public:
        //OccupancyGridPositions(nav_msgs::OccupancyGrid occupancyGrid, tf::Transform
parent);

        std::vector<OccupancyGridCell> cells;

        OccupancyGridPositions(nav_msgs::OccupancyGrid occupancyGrid, tf::Transform parent)
        {
                resolution = occupancyGrid.info.resolution;
                height = occupancyGrid.info.height;
                width = occupancyGrid.info.width;

                int index = 0;
                for (int i = 0; i < height; ++i)
                {
                for (int j = 0; j < width; ++j)
                        {
                                OccupancyGridCell cell;
                                cell.value = occupancyGrid.data[index];
                                cell.x = (j * resolution) + parent.getOrigin().getX();
                                cell.y = (i * resolution) + parent.getOrigin().getY();
                                index++;
                        }
                }
        }

        //boost::shared_ptr<costmap_converter::ObstacleArrayMsg> ToPolygons()
        std::vector<geometry_msgs::Polygon> ToPolygons(tf::TransformBroadcaster *tfDebug)
        {
                br = tfDebug;
                // take external vertices
                std::vector<Vertex> vertices;

                for (int i = 0; i < cells.size(); ++i)
                {
                        if (cells[i].value == OccupiedValue && GetOccupiedNeighboursNumber(i)
                <= 5)
                        {
                                Vertex vertex;
                                vertex.x = cells[i].x;
                                vertex.y = cells[i].y;
                                vertices.push_back(vertex);
                        }
                }
                //VisualizeVertices(vertices, "");

                // split to groups (islands)
                std::vector<std::vector<Vertex>> islands = GroupIslands(&vertices);
                for (int i = 0; i < islands.size(); ++i)
                        {
                                std::stringstream ss;
                                ss << "i" << i << "_";
                                std::string str = ss.str();
                                VisualizeVertices(islands[i], str);
                        }
                // keep only key points
                // for (int i = 0; i < islands.size(); ++i)
                // {
                // for (int j = 0; j < islands[i].size(); ++j) // vertices
                // {
                // int neighbours = GetNeighboursNumber(&vertices, j);
                // if (neighbours > 3)
                // {
                // islands[i].erase(islands[i].begin() + j);
                // j = 0;
                // } else
                // {
                // }
                // }
                // }

                // connect
                std::vector<geometry_msgs::Polygon> polygons;
                for (int i = 0; i < islands.size(); ++i)
                {
                        geometry_msgs::Polygon polygon;

                        geometry_msgs::Point32 point;
                        point.x = islands[i][0].x;
                        point.y = islands[i][0].y;

                        polygon.points.push_back(point);
                        int current = 0;
                        std::vector<int> used;
                        used.push_back(0);
                        while (polygon.points.size() < islands[i].size())
                        {
                                int next = GetNearestVertex(islands[i], current, used);
                                if (next == -1)
                                break;

                                used.push_back(next);
                                Vertex nextVertex = islands[i][next];

                                point.x = nextVertex.x;
                                point.y = nextVertex.y;
                                polygon.points.push_back(point);

                                current = next;
                        }

                        polygons.push_back(polygon);
                }

                return polygons;
        }

        boost::shared_ptr<costmap_converter::ObstacleArrayMsg> ToPointObstacles()
        {
        		costmap_converter::ObstacleArrayMsgPtr obstacleArrayMsg(new
        	costmap_converter::ObstacleArrayMsg());

        		for (int i = 0; i < cells.size(); ++i)
        		{
        			if (cells[i].value == OccupiedValue)
        			{
        				costmap_converter::ObstacleMsg obstacleMsg;
        				geometry_msgs::Point32 point;
        				point.x = cells[i].x;
        				point.y = cells[i].y;
        				obstacleMsg.polygon.points.push_back(point);
        				obstacleMsg.radius = resolution;
        				obstacleMsg.id = 1;
        				obstacleMsg.header.frame_id = "map";
        				obstacleArrayMsg->header.frame_id = "map";
        				obstacleArrayMsg->obstacles.push_back(obstacleMsg);
        			}
        		}
        		return obstacleArrayMsg;
        	}

        	private:

        	void VisualizeVertices(std::vector<Vertex> vertices, std::string prefix)
        	{
        		for (int i = 0; i < vertices.size(); ++i)
        		{
        			double x = vertices[i].x;
        			double y = vertices[i].y;
        			tf::Transform transform;
        			transform.setOrigin(tf::Vector3(x, y, 0));
        			transform.setRotation(tf::Quaternion());
        			std::stringstream ss;
        			ss << prefix << i;
        			std::string str = ss.str();
        			br->sendTransform(tf::StampedTransform(transform, ros::Time::now(),
        		"map", str));
        		}
        	}

        	int GetNearestVertex(std::vector<Vertex> vertices, int current, std::vector<int>
        used)
        	{
        		// find candidates
        		std::vector<Vertex> candidates;
        		Vertex c = vertices[current];

        		for (int i = 0; i < vertices.size(); ++i)
        		{
```

```
                Vertex v = vertices[i];

                double distance = sqrt(fabs((c.x - v.x) * (c.x - v.x) + (c.y - v.y) *
        (c.y - v.y)));
                if (distance <= resolution + resolution / 2)
                {
                }
        }
        int indexClosest = -1;

        double previousDistance = 10000000;
        for (int i = 0; i < vertices.size(); ++i)
        {
                if (i == current || std::count(used.begin(), used.end(), i) != 0)
                continue; // already used

                Vertex v = vertices[i];

                double distance = sqrt(fabs((c.x - v.x) * (c.x - v.x) + (c.y - v.y) *
        (c.y - v.y)));
                if (distance <= resolution + resolution / 100)
                {
                        return i;
                }

                if (distance < previousDistance)
                {
                        indexClosest = i;
                        previousDistance = distance;
                }
        }

        return indexClosest;
}

int GetNeighboursNumber(std::vector<Vertex> *vertices, int vertex)
{
int neighboursNumber = 0;

Vertex a = (*vertices)[vertex];
for (int j = 0; j < vertices->size(); ++j)
{
        if (vertex != j)
        {
                if (AreNeighbours(a, (*vertices)[j]))
                {
                        neighboursNumber++;
                }
        }
}
```

```
        return neighboursNumber;
}

int GetOccupiedNeighboursNumber(int cellIndex)
{
        double threshold = resolution / 2;
        int neighboursNumber = 0;
        OccupancyGridCell cell = cells[cellIndex];
        for (int i = 0; i < cells.size(); ++i)
        {
                if (cellIndex != i)
                {
                        if (fabs(cells[i].x - cell.x) < threshold + resolution
                        && fabs(cells[i].y - cell.y) < threshold + resolution
                        && cells[i].value == OccupiedValue)
                        {
                                neighboursNumber++;
                        }
                }
        }
        return neighboursNumber;
}

bool AreNeighbours(Vertex a, Vertex b) const
{
        return fabs(a.x - b.x) <= resolution * 2 + resolution / 4 &&
        fabs(a.y - b.y) <= resolution * 2 + resolution / 4;
}

        bool IsNeighbourFree(int i, OccupancyGridCell cell, double xOffset, double yOffset,
double threshold)
{
        if (fabs(cells[i].x - cell.x - xOffset) < threshold
        && fabs(cells[i].y - cell.y - yOffset) < threshold
        && cells[i].value != OccupiedValue)
        {
                return true;
        }
        return false;
}

bool HasFreeNeighbour(OccupancyGridCell cell)
{
        double threshold = resolution / 2;
        int freeNeighbours = 0;

        for (int i = 0; i < cells.size(); ++i)
        {
                // exclude given cell
```

```
		if (fabs(cells[i].x - cell.x) < threshold + resolution
		&& fabs(cells[i].y - cell.y) < threshold + resolution
		&& cells[i].value != OccupiedValue)
		{
			freeNeighbours++;
		}
		/* // return true if any
		if (IsNeighbourFree(i, cell, -resolution, resolution, threshold) || // left
	top
		//IsNeighbourFree(i, cell, 0, resolution, threshold) || // top
		IsNeighbourFree(i, cell, resolution, resolution, threshold) || // right top
		//IsNeighbourFree(i, cell, resolution, 0, threshold) || // right
		IsNeighbourFree(i, cell, resolution, -resolution, threshold) || // right
	bottom
		//IsNeighbourFree(i, cell, 0, -resolution, threshold) || // bottom
		IsNeighbourFree(i, cell, -resolution, resolution, threshold))// || // left
	bottom
		//IsNeighbourFree(i, cell, -resolution, 0, threshold)) // left
		{
			return true;
		}*/
	}

	return freeNeighbours >= 5;
}

std::vector<std::vector<Vertex>>

GroupIslands(std::vector<Vertex> *vertices)
{
	std::vector<std::vector<Vertex>> islands;

	for (int i = 0; i < vertices->size(); ++i)
	{
		bool added = false;
		if (!islands.empty())
		added = AddToExistingIsland(&islands, (*vertices)[i]);

		// make a new island
		if (!added)
		{
			std::vector<Vertex> newIsland;
			newIsland.push_back((*vertices)[i]);
			islands.push_back(newIsland);
		}

	return
	islands;
```

```
}

bool HaveCloseVertices(const std::vector<std::vector<Vertex>> *islands, int i, int j) const
{
        bool have = false;
        for (int k = 0; k < (*islands)[i].size(); ++k)
        {
                for (int l = 0; l < (*islands)[j].size(); ++l)
                {
                        have = AreNeighbours((*islands)[j][k], (*islands)[j][l]);
                }
        }
        return have;
}

bool AddToExistingIsland(std::vector<std::vector<Vertex>> *islands, Vertex vertex)
{
        for (int j = 0; j < islands->size(); ++j)
        {
                for (int k = 0; k < (*islands)[j].size(); ++k) // vertices in island
        {
        // if any vertex is neighbour
        if (AreNeighbours((*islands)[j][k], vertex))
        {
                (*islands)[j].push_back(vertex);
                return true;
        }
        }
}

return false;
}
};

#endif //SRC_OCCUPANCYGRIDPOSITIONS_H
```

ДОДАТОК Б

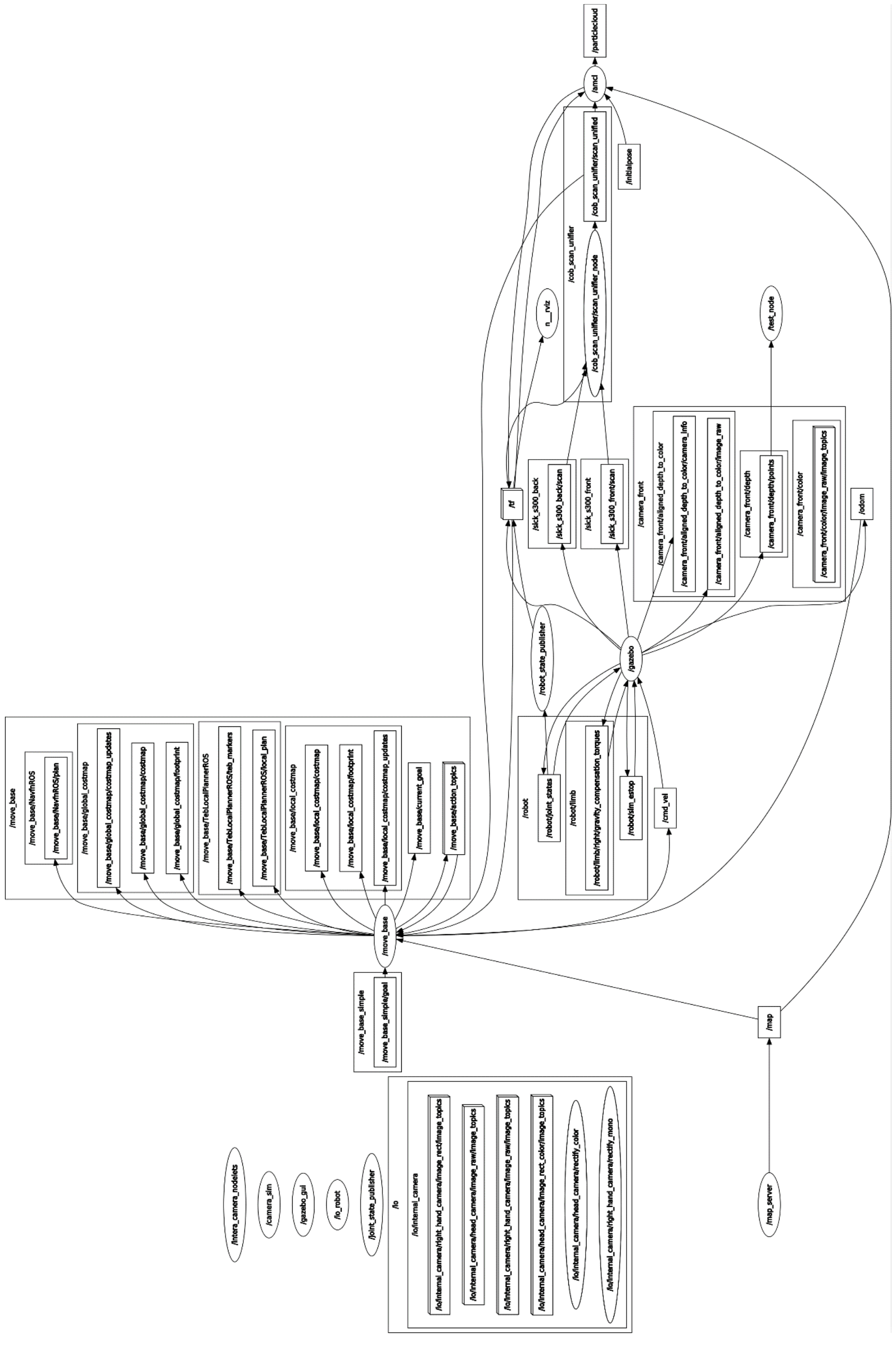
ВІДГУК

керівника економічного розділу
на кваліфікаційну роботу магістра
на тему: «Моделі, алгоритми та програмне забезпечення для планування шляху
для навігації мобільних роботів з уникненням перешкод
на основі дерева октантів»
студента групи 121м-19-1 Панасейко Ганни Миколаївни

Керівник економічного розділу
доцент каф. ПЕП та ПУ, к.е.н. **Л. В. Касьяненко**

APPENDIX C

NODES COMMUNICATION


/particlecloud
/amcl
/initialpose
/cob_scan_unifier
/cob_scan_unifier/scan_unified
/cob_scan_unifier/scan_unifier_node
n__rviz
/test_node
/tf
/sick_s300_back
/sick_s300_back/scan
/sick_s300_front
/sick_s300_front/scan
/camera_front
/camera_front/aligned_depth_to_color
/camera_front/aligned_depth_to_color/camera_info
/camera_front/aligned_depth_to_color/image_raw
/camera_front/depth
/camera_front/depth/points
/camera_front/color
/camera_front/color/image_raw/image_topics
/odom
/robot_state_publisher
/gazebo
/robot
/robot/joint_states
/robot/limb
/robot/limb/right/gravity_compensation_torques
/robot/sim_estop
/cmd_vel
/move_base
/move_base/NavfnROS
/move_base/NavfnROS/plan
/move_base/global_costmap
/move_base/global_costmap/costmap_updates
/move_base/global_costmap/costmap
/move_base/global_costmap/footprint
/move_base/TebLocalPlannerROS
/move_base/TebLocalPlannerROS/teb_markers
/move_base/TebLocalPlannerROS/local_plan
/move_base/local_costmap
/move_base/local_costmap/costmap
/move_base/local_costmap/footprint
/move_base/local_costmap/costmap_updates
/move_base/current_goal
/move_base/action_topics
/move_base_simple
/move_base_simple/goal
/map
/map_server
/io
/io/internal_camera
/io/internal_camera/right_hand_camera/image_rect/image_topics
/io/internal_camera/head_camera/image_raw/image_topics
/io/internal_camera/right_hand_camera/image_raw/image_topics
/io/internal_camera/head_camera/image_rect_color/image_topics
/io/internal_camera/head_camera/rectify_color
/io/internal_camera/right_hand_camera/rectify_mono
/joint_state_publisher
/io_robot
/gazebo_gui
/camera_sim
/intera_camera_nodelets

APPENDIX D

LIST OF FILES ON THE DISC

File name	**Description**
Explanatory documents	
ThesisPanaseikoHanna.doc	Explanatory note to the diploma project. Word document.
ThesisPanaseikoHanna.pdf	Explanatory note to the diploma project in PDF format
Program	
Navigation.zip	Archive. Contains program codes and a program
Presentation	
PresentationPanaseikoHanna.ppt	Presentation of the diploma project