Unaprjeđenje performansi robotskih usisavača putem poboljšane izgradnje karata

Krapinec, Leon

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 715

UNAPRJEĐENJE PERFORMANSI ROBOTSKIH USISAVAČA PUTEM POBOLJŠANE IZGRADNJE KARATA

Leon Krapinec

Zagreb, veljača 2025.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 715

UNAPRJEĐENJE PERFORMANSI ROBOTSKIH USISAVAČA PUTEM POBOLJŠANE IZGRADNJE KARATA

Leon Krapinec

Zagreb, veljača 2025.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 30. rujna 2024.

DIPLOMSKI ZADATAK br. 715

Pristupnik:	**Leon Krapinec (0036527325)**
Studij:	Računarstvo
Profil:	Programsko inženjerstvo i informacijski sustavi
Mentorica:	prof. dr. sc. Marija Seder

Zadatak: **Unaprjeđenje performansi robotskih usisavača putem poboljšane izgradnje karata**

Opis zadatka:

Cilj ovog diplomskog rada je unaprijediti performanse starijih modela robotskih usisavača poboljšanjem procedure izrade karata prostora uz ograničene senzorske sposobnosti i strategije istraživanja prostora. Rad se primarno fokusira na razvoj metode stvaranja karata njihove okoline unatoč ograničenim senzorskim podacima. Cilj je osmisliti strategiju istraživanja prostora prikladnu za računalne i senzorske mogućnosti starijih robotskih usisavača. Korištenjem fuzije senzora, predložena metodologija nastoji prevladati senzorska ograničenja, što rezultira poboljšanom točnošću karata uz maksimizaciju pokrivenosti. Implementacija ovog pristupa provest će se unutar okruženja Robot Operating System (ROS), a potom će biti validirana putem simulacija.

Rok za predaju rada: 14. veljače 2025.

Content table

Introduction 1

1. Sensors 2

1.1. Sensor Fusion 2

1.2. Sensor Noise and Errors 2

2. Hardware Used in Research 3

2.1. Anatomy of a Roomba 4

2.2. Roomba Open Interface 5

3. Fundamentals of Robotic Mapping 6

3.1. Mapping in Robotics 6

3.2. Types of Maps 7

3.3. Mapping Algorithms 8

3.3.1. Occupancy Grid 8

3.3.2. Mapping Using Waypoints 8

3.3.3. Mapping Using Landmarks 9

3.3.4. Simultaneous Localization and Mapping 9

3.3.5. Topological Mapping 9

3.3.6. Line Segment Based Mapping 10

3.4. Optimal Mapping Algorithm for Cleaning Robot 10

4. Fundamentals of Robot Localization 11

4.1. Pose 11

4.2. Odometry Based Localization 12

4.3. Localization Problem 13

5. ROS 2 Dundamentals 15

5.1. ROS 2 Components 15

5.2. Unified Robot Description Format 17

5.3. Tools for Visualization and Simulation 17
6. Roomba Driver 18
6.1. Odometry Implementation 19
7. Exploration Strategy 21
7.1. Code Implementation 25
Conclusion 40
Literature 41
Summary 42
Sažetak 43

Introduction

The rapid advancement of robotics technology has revolutionized household appliances, with robot vacuum cleaners emerging as one of the most prominent examples of consumer robotics. Once viewed as novelties, these devices have evolved into sophisticated systems capable of autonomously navigating and cleaning a variety of environments. However, as user expectations increase and environments grow more complex, there is a rising demand for improved performance, efficiency, and adaptability in robot vacuum cleaners. This research aims to meet these demands by exploring the integration of advanced robotics principles, sensor technologies, and software solutions to enhance the functionality of robot vacuum cleaners.

This study examines fundamental robotics terminology and the essential role of sensors in facilitating autonomous navigation and environmental interaction. Sensors function as the eyes and ears of robotic systems, providing crucial data for decision-making and task execution. By analyzing the types and functions of sensors, this research emphasizes their importance in augmenting the capabilities of robot vacuum cleaners, enabling them to perceive and adapt to their surroundings effectively.

The study further investigates the specific model of the robot vacuum cleaner used, offering an in-depth analysis of its construction, components, and operational mechanics. Understanding these devices' hardware architecture and design principles is vital for identifying areas for improvement and implementing enhancements. This analysis lays the groundwork for a deeper exploration of foundational robotics principles, particularly mapping and localization, which are essential for autonomous navigation.

Mapping and localization are critical to the functionality of robot vacuum cleaners, allowing them to create representations of their environment and ascertain their position within it. This research clarifies the utility of these concepts, the various mapping and localization techniques available, and their application in enhancing robotic systems. By utilizing these principles, robot vacuum cleaners can achieve greater accuracy, efficiency, and adaptability when navigating complex environments.

To translate theoretical concepts into practical solutions, this study assesses the software tools and frameworks employed to implement these advancements. The effectiveness and suitability of the chosen software are evaluated in terms of addressing the identified challenges. Additionally, the research examines external libraries that facilitate implementation, highlighting their advantages and contributions to the development process.

Ultimately, the study engages with the principal algorithm that operationalizes the theoretical framework, conducting a thorough review of the code to ensure its efficacy and reliability. By merging theoretical insights with practical implementation, this research aims to contribute to the ongoing evolution of robot vacuum cleaners, paving the way for more intelligent, efficient, and user-friendly devices.

1. Sensors

Sensors play a pivotal role in robotics. They serve as the interface between the robot and its environment. Sensors are akin to human senses in the context that they are a critical feedback system. They provide essential data about the surroundings that enable the robot to perceive its environment. Without them, robots would be just static devices lacking dynamic adaptability. The sensor can be put into two primary categories: proprioceptive and exteroceptive sensors.

Proprioceptive sensors provide information about the internal state of the robot. Some examples are encoders that measure rotational or linear position and velocity, and inertial measurement units (IMUs) that measure acceleration, angular velocity, and orientation.

The second type of sensor is the exteroceptive sensor. It provides a measure of external stimuli or environmental conditions. Some examples of exteroceptive sensors are a camera, lidar, bumper, and infrared sensor.

Robot sensors have a multitude of functions, and with that, they manage to transform a simple machine into a dynamic system. Sensors can be used in different applications.

Position and navigation are key functions if a robot can move. A common practice for the machine is to move from one point in the environment to another while avoiding obstacles. In this scenario, the camera and lidar are one of the useful sensors to make that process smoother.

Safety is another application where sensors can be used. With an infrared sensor, the robot can detect living presence. With this knowledge, the machine can stop actions that can harm humans and prevent accidents.

The third use case is environment monitoring. If the environment needs special conditions to function, the machine can constantly measure and take action if those requirements are not satisfied. Temperature and humidity sensors are one of the examples.

1.1. Sensor Fusion

Sensor fusion is the ability to bring together data from different sensors to form a single model of the environment around the vehicle. This results in a more accurate model because it balances the strengths of the different sensors. Moreover, because of that machine actions can be more complex. Each sensor type has its benefits and challenges. For instance, cameras are excellent for recognizing objects, but they can be easily disturbed by rain or dirt. Light sensor is a low-cost sensor that recognizes distance, but it has a hard time with reflective surfaces. Sensor fusion enables the use of the benefits of one sensor and replaces its disadvantages with a different type of sensor.

1.2. Sensor Noise and Errors

Sensor noise refers to disturbance in the signal produced by sensors. This causes deviation of the correct measurements, which results in bad perception and decision-making. Sensor noise is a common challenge in robotics because it can degrade performance.

2. Hardware Used in Research

The Roomba series 600 is an older line of robotic vacuum cleaners designed for small to medium-sized indoor spaces. It employs a random cleaning pattern because it lacks advanced sensors found in newer, higher-end models. Consequently, it does not offer the ability to map the room. While there is no detailed documentation available for this series, alternative resources can be utilized. The Roomba Create 2 is a robot specifically designed for educational, hobbyist, and developmental purposes. It is based on the Roomba series 600 but does not include a cleaning mechanism. However, it features comprehensive documentation regarding its physical structure and components. This documentation explains how the sensors operate, what data they collect, and the interface between the robot and the computer. A special cable is required to connect the Roomba to the computer. The cable used in this research is a mini-DIN 8-pin TTL serial port cable with an FT232RL serial module chip.

Image 2.1 Roomba series

Image 2.2 600Mini-DIN 8-pin TTL serial port cable

2.1. Anatomy of a Roomba

Like most other models, the Roomba Series 600 features a circular design. It has a width of 347mm, a height of 92mm, and weighs 3.6 kg. The track width between the two motorized wheels measures 235mm.

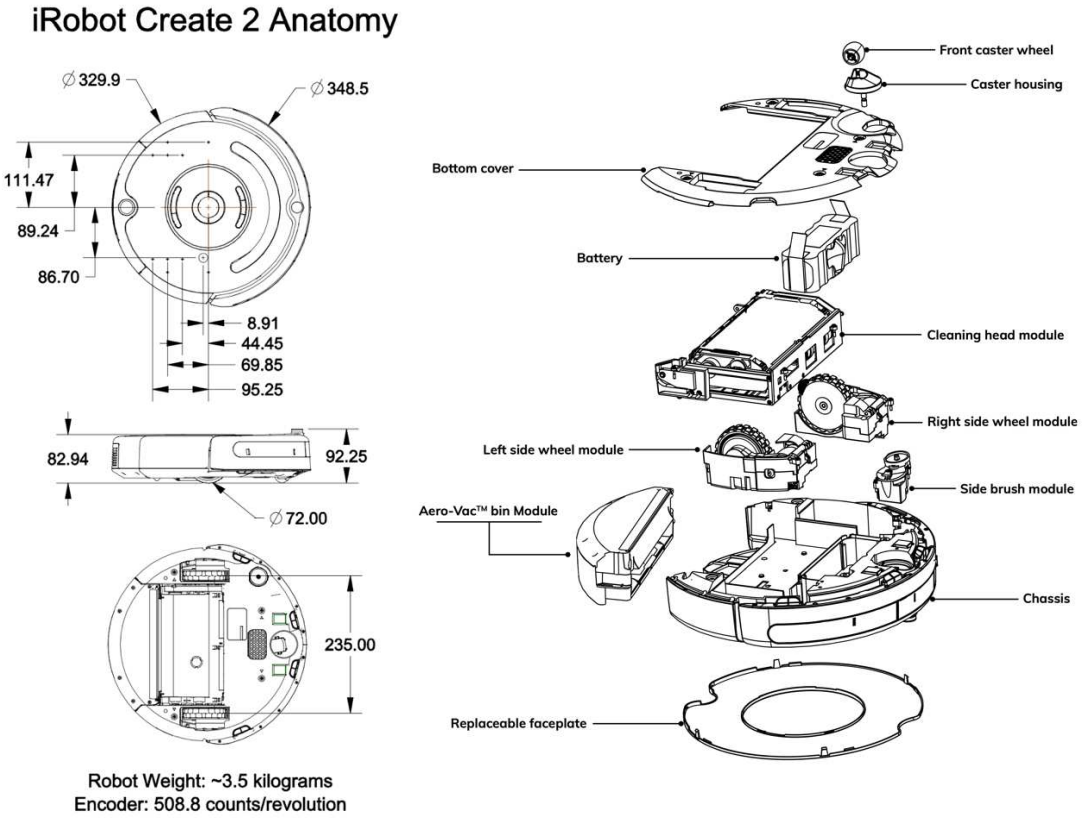

Image 2.3 Roomba Anatomy

Wheel encoders are located on the robot's left and right sides, with a caster wheel positioned at the center front. These encoders track wheel rotation to estimate the distance traveled and assist in navigation. The second type of sensor detects physical obstacles, consisting of bumper sensors that sense contact with obstacles and light sensors that measure an obstacle's proximity to the robot. The position of the bumper sensor is at the front of the robot, and it consists of two sections: the left and right sides. The light sensor is positioned at the front of the robot and is centered between the bumper. These sensors provide awareness within a two-dimensional space. Another type of sensor is the cliff sensor, which uses infrared technology to detect drop-offs. This helps prevent the robot from falling by detecting height changes. The robot features six cliff sensors located on its underside, positioned around the edges. Two are placed at the front, and two are positioned above the right and left wheels, with the last two situated beneath the two wheels. The fourth type of sensor is specifically designed for cleaning functions. Dirt detectors identify areas with higher concentrations of dirt, prompting the robot to focus on those spots.

2.2. Roomba Open Interface

The Roomba Open Interface (OI) is a software interface designed for controlling and manipulating Roomba's operational behavior. This interface allows users to modify Roomba's functionalities and access its sensor data through a comprehensive set of commands. These commands include mode commands, actuator commands, song commands, cleaning commands, and sensor commands, which are transmitted to Roomba's serial port via a personal computer or microcontroller connected to the mini-DIN connector. Each command begins with a one-byte opcode, and some must be followed by data bytes. In response to a Sensors command, Query List command, or Stream command requesting a packet of sensor data bytes, Roomba sends back one of 58 different sensor data packets, depending on the value of the packet data byte. Some packets contain groups of other packets, and some sensor data values are 16-bit values. The most important packet will be explained.

The bumper state is included in a packet with ID 7. This packet consists of one unsigned byte. The value of the left bumper is found on bit number one, while the value of the right bumper is located on byte number zero. If the bumper state is zero, it indicates that the bumper is not pressed. Conversely, if the bumper is pressed, its state will be one.

Bit	7	6	5	4	3	2	1	0
Value	Reserved				Wheel Drop Left?	Wheel Drop Right?	Bump Left?	Bump Right?

Image 2.4 Structure of the bumper packet

A packet with ID 45 refers to the light sensor. It also consists of one unsigned byte. This sensor has six positions where it can detect an obstacle. Like the bumper sensor, if an obstacle is detected, that position returns a value of one; otherwise, it returns zero. To obtain the strength of each signal, packets with IDs between 46 and 51 should be requested. They are constructed of two unsigned data bytes. The strength is represented as an unsigned 16-bit value with the high byte first. The value can range from 0 to 4095.

Bit	7	6	5	4	3	2	1	0
Value	Reserved		Lt Bumper Right?	Lt Bumper Front Right?	Lt Bumper Center Right?	Lt Bumper Center Left?	Lt Bumper Front Left?	Lt Bumper Left?

Image 2.5 Structure of the light sensor packet

The packet with ID 19 contains the distance the robot has traveled since the last request was made. The value is sent as a signed 16-bit integer, with the high byte first. This measurement is in millimeters. Distance represents the sum of the distances traveled by both wheels divided by two. Positive values indicate forward travel; negative values indicate reverse travel. If the value is not polled frequently enough, it is capped at its minimum or maximum. The value range is between -32768 and 32767.

To obtain the angle in degrees that Roomba has turned, the packet with ID 20 must be requested. The value represents the difference between the current angle and the last requested angle, sent as a signed 16-bit integer, high byte first. The value range is between -32768 and 32767.

3. Fundamentals of Robotic Mapping

This chapter provides a fundamental knowledge of mapping in robotics. It explains what mapping is and why it is important. It also explores the different types of maps and the algorithms used to generate them. Finally, it concludes with an explanation of the selected mapping approach implemented in practice and the rationale behind this choice.

3.1. Mapping in Robotics

In robotics, mapping refers to the process of creating a representation of the environment that the robot can perceive through its sensors. This representation can be two-dimensional or three-dimensional. The goal of mapping is to capture the spatial layout, obstacles, and other significant features of the environment. It is a crucial aspect of robot navigation and autonomous operation in space. Autonomous navigation necessitates precise localization and mapping. The more accurate the generated map, the easier it is for the robot to navigate. With a highly accurate map, the robot becomes fully aware of its surroundings and can maneuver with ease. Conversely, uncertainty can complicate matters. Uncertainties in low-accuracy maps stem from sensor noise, localization errors, environmental factors, and limitations in algorithms. Additionally, decreased tracking and mapping accuracy adversely affect control, planning, and overall performance.

3.2. Types of Maps

A map depicts the environment in which the robot operates. There are two standard models of indoor environments: the metric map and the topological map. A metric or grid-based map consists of cells, each indicating occupied or empty space. In contrast, the topological approach represents the environment as a graph, composed of nodes that denote distinct obstacles, locations, or situations.

A metric map can be easily generated and maintained in large environments. Geometric data enables the robot to distinguish similar areas within the environment. Sensor and odometry data progressively update the robot's position, making it ideal for dynamic settings. Furthermore, it simplifies computing the shortest path.

One limitation of the grid-based approach is its planning inefficiency. A higher grid resolution increases the number of cells that path-planning algorithms must search, resulting in greater computational costs. Additionally, this method heavily relies on accurate odometry to ascertain the robot's position.

One advantage of a topological map is its compact size. The complexity of the space determines the map's resolution, allowing symbolic planners and problem solvers to devise plans quickly and represent their work conveniently. Moreover, topological maps can easily recover from drift and slippage since they do not require an exact geometric location.

The main disadvantages of the topological approach compared to the metric map include difficulties in large-scale environments, challenges in recognizing places, and issues with computing the shortest path. A topological map determines the robot's position based on landmarks, making it challenging to differentiate between two similar locations. If sensor information is unclear, it becomes difficult to construct and maintain a comprehensive environment.

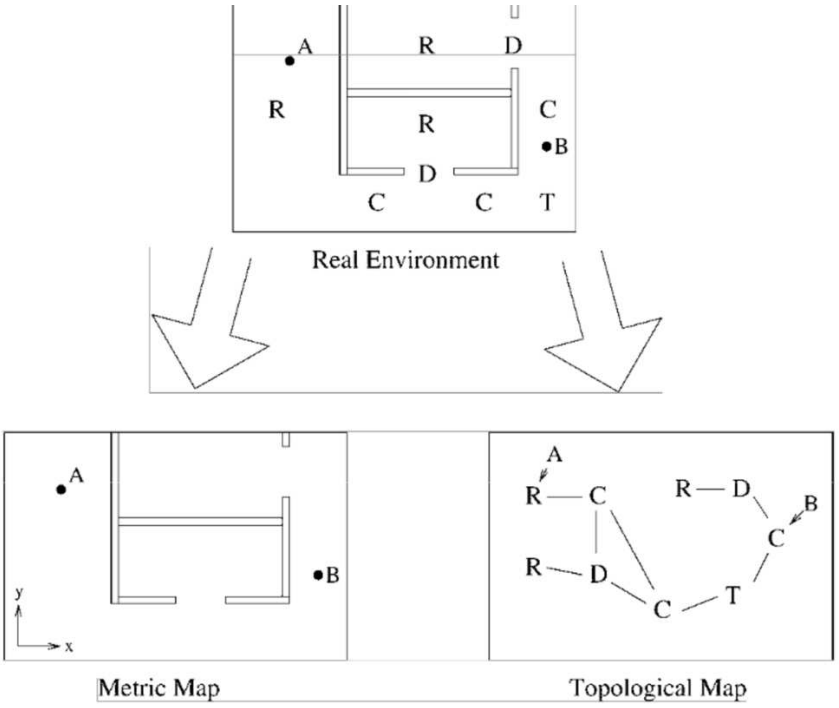

Image 3.1 Example of metric and topological map

3.3. Mapping Algorithms

A mapping algorithm is a method used by robots to create a representation of their environment using sensor data. There are various algorithms, each best tailored to a specific application. The functionality, advantages, and limitations of the most widely used algorithms will be analyzed and explained.

3.3.1. Occupancy Grid

Occupancy grid mapping is the most popular mapping method. It represents a map of the environment as an evenly spaced grid of cells, with each cell value indicating the presence of an obstacle at that location in space. The appropriate set of possible values is chosen based on the task, the capabilities of the robot's sensor system, and the specifics of the environment. Each cell's initial state is set to unknown. When a sensor detects an obstacle or free space, this information is recorded in the corresponding cell. The map is completed when all cells have a definitive value.

Occupancy grids can be used for effective map planning because the distance between the goal and each cell can be easily labeled. To perform path planning, a robot must select the neighboring cell with the lowest label until it reaches the goal. Another advantage is that a grid provides a discrete representation, allowing computation to be focused on only a portion of the grid. Moreover, this algorithm is easy to understand and implement, and its efficiency does not depend on the size of the environment. By adjusting the map resolution, computation costs can be controlled.

One disadvantage of occupancy grids is that, due to fixed resolution, objects can only be mapped if they are within the grid boundary. The more cells the robot successfully marks, the more memory is consumed, leading to a challenging a priori estimation of the required memory size. Additionally, since each cell must be marked, occupancy grids can become memory-intensive in large, high-resolution maps. While occupancy grids can manage dynamic environments, they require frequent updates, which can incur high computation costs. Each cell provides only the likelihood of obstacle presence and lacks semantic data. Another issue is the difficulty in representing complex shapes, as an occupancy grid cannot have only part of a cell occupied.

3.3.2. Mapping Using Waypoints

Waypoint refers to a set of predefined coordinates. Waypoints are interconnected, allowing the robot to navigate toward its goal. While navigating through waypoints, the robot follows initial instructions regarding the actions to take when reaching a specific waypoint.

This mapping approach is similar to mapping with landmarks. The distinction is that the reference point of a waypoint is represented as a coordinate. This method relies more on GPS and landmarks, whereas waypoints are detected through sensors.

3.3.3. Mapping Using Landmarks

Landmarks are objects found in the environment, such as furniture, signposts, speed signs, and trees. A special type of landmark called a beacon, can transmit a signal that a machine can receive. Landmarks serve as reference points for localization. Mapping with landmarks involves recording their positions relative to each other. A robot needs to follow a few landmarks to reach its goal. At any point in the environment, the robot must be able to detect multiple landmarks. Moreover, landmarks should not be too close to the robot, as localization is calculated by estimating the distance to all landmarks within the sensor's range.

Landmarks can be used to determine the robot's position and orientation. This method generates a map that contains only landmarks, not the entire environment, which consumes much less memory and computational power than grid-based maps. Mapping using landmarks is particularly robust in dynamic environments because landmarks are mostly static and stable points. The finished map is easy for humans to interpret.

One of the biggest flaws of this approach is that the generated map lacks details about the environment. Obstacles not identified as landmarks or free space are not represented. If the environment has only a few distinct features, the algorithm struggles to detect landmarks, which can lead to significant localization drift.

3.3.4. Simultaneous Localization and Mapping

Simultaneous localization and mapping, or SLAM for short, attempts to map the unknown space with the robot while defining its position in that environment. The resolution of the generated map heavily depends on the precision of robot sensors and the algorithm that integrates mapping and localization. SLAM can be thought of as a chicken and egg problem. Mapping an environment requires an estimate of the robot's localization, while localization needs an estimate of the map.

One of the main advantages of using SLAM for navigation is that it enables the robot to operate in unknown or changing environments without relying on external localization systems or predefined maps. The map generated by SLAM is in high definition.

A disadvantage of SLAM is that it requires high computation power and memory. Moreover, it requires sensors with the ability to accurately measure range. Also, the implementation process is complicated and requires a deep understanding of probability theory, statistics, recursion, and system dynamics.

3.3.5. Topological Mapping

A topological map is a graph made up of nodes and edges. A node signifies a distinct location or landmark and may include semantic labels or features. An edge represents a connection between nodes and can convey information about distance, direction, or the difficulty of traversing the path. If two nodes are connected by an edge, it indicates that the path between them is traversable and free of obstacles. The nodes do not represent areas on the map, and the edges do not indicate the distances between nodes. The primary motivation behind the topological approach is that the environment may include vital features that lack geometric relevance but are essential for localization.

Topological maps are memory-efficient as they do not encompass every detail of the environment, making them suitable for larger areas. Additionally, multiple maps can be

combined into a larger one. Path planning is simplified due to the ease of employing graph traversal algorithms.

This algorithm's limitation is that it lacks detailed information about the environment and does not provide accurate metric data. Topological mapping can struggle in dynamic environments because edges are designed to always be obstacle-free.

3.3.6. Line Segment Based Mapping

Line segment based mapping is a technique used in robotics and computer vision to represent an environment by extracting and mapping line segments from sensor data. A line segment typically represents an object's edge or boundary. Two sequential scans are selected from the obtained set, and scan matching is performed. Scan matching is the process of finding the rotation angle and translating it so that at least one angle in one scan superimposes an equal angle in another scan.

The advantage of this approach is that odometry is not needed to build the complete map. Therefore, odometry errors, control mechanisms, or computation do not affect line segment-based mapping. Line segments are more memory-efficient than grid maps. The algorithm is not dependent on the time segment, allowing different robots to generate the map in multiple sessions.

One of the flaws of this algorithm is that a maximum of two scans can be matched at any given time. Another limitation is that a curved or irregular surface cannot be accurately represented due to the nature of a line segment. The scans must be ordered clockwise or counterclockwise to be suitable for scan-matching, which restricts the robot's movement while exploring the environment.

3.4. Optimal Mapping Algorithm for Cleaning Robot

Cleaning robots are primarily utilized in relatively confined indoor settings. Consequently, employing waypoint mapping algorithms is not advisable. The primary function of these robots is to navigate the entire environment while performing cleaning tasks. In the event that the robot becomes immobilized, knowledge of its exact location is critical. Mapping methodologies based on landmarks and topological mapping are inadequate in this context, as they lack detailed environmental information. A cleaning robot must maintain functionality in the presence of both humans and pets. This necessitates the implementation of algorithms designed for dynamic environments, further reinforcing the unsuitability of several previously mentioned approaches. Our robot is equipped with basic sensors capable only of detecting obstacles within proximity. This limitation excludes the use of the SLAM algorithm and line segment-based mapping techniques. Only the occupancy grid approach meets all specified criteria. This presents an optimal use case for a cleaning robot. Upon detecting a soiled area, the robot is capable of marking it on the map; should the robot halt for any reason, it can be easily located. Furthermore, it can display the sections of the map it has cleaned and the remaining areas requiring attention. All of this can be easily labeled on the metric map.

4. Fundamentals of Robot Localization

When the map is successfully generated, a new obstacle arises: how to determine where the robot is located relative to the map environment. The solution to this problem is called localization. Without localization, the robot cannot make autonomous decisions.

4.1. Pose

Pose represents the position and orientation of the object. The position is represented with Euclidean coordinates. Orientation represents the direction the object is facing, often represented as an angle relative to the reference axis in 2D space, the robot moves within a single plane. Robot pose is notated as $\vec{x}$.

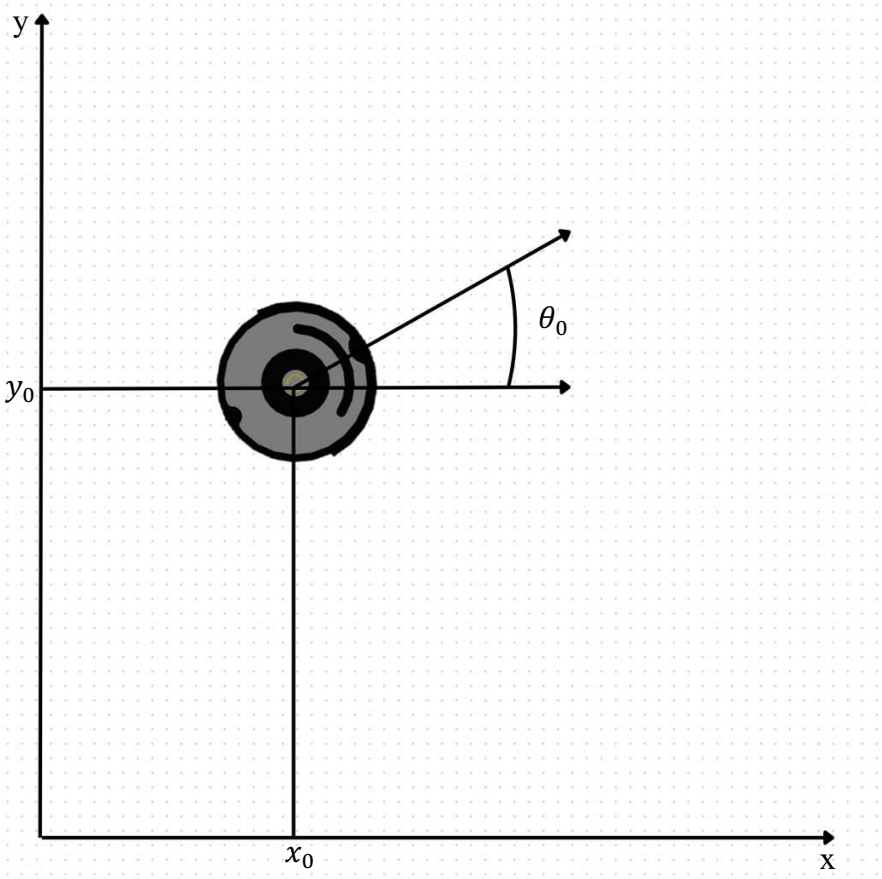

Image 4.1 An example of a pose

Current pose ($\vec{x}_0$) of the robot can be described as $\begin{pmatrix} x_0 \\ y_0 \\ \theta_0 \end{pmatrix}$. To update the pose, the transformed change is added to the previous pose where $\varphi = \Delta\theta$.

$$\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ \theta_0 \end{pmatrix} + \begin{pmatrix} \Delta x \\ \Delta y \\ \varphi \end{pmatrix}$$

4.2. Odometry Based Localization

Odometry uses sensor data to estimate the robot's pose over time. It enables computing small relative displacements that are integrated over time to yield the final pose in space.

To determine the current pose of the robot, the change from the previous pose needs to be calculated. The prerequisite for that is data from the encoder sensors. Oftentimes, there are two sensors placed parallel to the robot's body in the x-direction. The displacement of the left sensor is Δx_l, and Δx_r is the displacement of the right sensor. The lateral distance between these two sensors is called track width, notated as L. This is very important for determining the angle for turning approximations. This value will need to be tuned, which means tested repeatedly and then brought to some converging value that is close to the actual measurement.

$$\varphi = \frac{\Delta x_l - \Delta x_r}{L}$$

To understand the robot's true displacement, the center of the robot's motion needs to be calculated. This is calculated by taking an average of two displacements:

$$\Delta x_c = \frac{\Delta x_l + \Delta x_r}{2}$$

To calculate the rotation of the robot's body, the rotation matrix is used:

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Simplified calculation of relative deltas can be achieved by transforming robot's relative delta via a rotation matrix where the relative pose difference is rotated by the original heading:

$$\begin{pmatrix} \Delta x \\ \Delta y \\ \varphi \end{pmatrix} = R_z(\theta_0) * \begin{pmatrix} \Delta x_c \\ \Delta x_c \\ \varphi \end{pmatrix}$$

This method assumes that the robot follows the straight path between updates. In reality, the robot can travel around curves, and with this approach, the inaccurate approximation

appears. To overcome that, the robot's motion should be modeled as following a circular arc. The turn radius R is calculated as:

$$R = \frac{\Delta x_c}{\varphi}$$

The updated formula now looks like this:

$$\begin{pmatrix} \Delta x \\ \Delta y \\ \varphi \end{pmatrix} = R_z(\theta_0) * \begin{pmatrix} R * \sin\varphi \\ -R * (1 - \cos\varphi) \\ \varphi \end{pmatrix}$$

If there was no rotation ($\varphi = 0$), then the formula can be simplified to:

$$\begin{pmatrix} \Delta x \\ \Delta y \\ \varphi \end{pmatrix} = (\Delta x_c \quad \Delta x_c \quad 1ß) * \begin{pmatrix} \cos\theta_0 \\ \sin\theta_0 \\ \varphi \end{pmatrix}$$

4.3. Localization Problem

The robot can continuously be localized precisely if perfect sensory data about the environment is acquired. However, other sensors are often inaccurate and deficient in certain aspects. Sensors cannot be fully trusted and frequently yield incorrect measurements. Additionally, real-world conditions that are not typically modeled contribute to these challenges. The robot may deviate from its intended path for various reasons, such as collisions, slippery floors, or people picking it up and moving it. These situations must be addressed effectively to restore the robot's location, as they can be difficult or impossible to predict. Furthermore, maps often represent the environment as a collection of static objects, which poses a problem since the environment may contain additional barriers. These could be static or dynamic obstacles. Localization techniques must appropriately tackle these issues. Localization problems are generally very difficult or impossible to resolve without adequate sensors.

The robot localization problem can be categorized into position tracking, global positioning, and the kidnapped robot problem. Position tracking is a scenario in which the robot's starting position is known, and the aim is to ascertain its location at every moment in time. To resolve this, the previous position must be recorded. Initially, the starting position is established. Before the first movement, the initial position is documented, and the current position is calculated using odometry and sensor data. An internal map is maintained, and every movement is tracked within it. Global localization refers to the process of determining the robot's location when there is no information about its starting position or orientation. This poses a significant challenge if the robot relies solely on a bumper and light sensor. If the robot has a docking station and its position is indicated on a

map, it must employ simple heuristics to navigate to the dock and reset its position. If the docking station is unavailable, this issue remains unsolvable. The kidnapped robot problem, particularly for a robot vacuum cleaner, arises when the robot is picked up and transported to a different location. This dilemma can be resolved similarly to the global localization problem by utilizing only odometry, light, and bumper sensor data.

5. ROS 2 Dundamentals

The iRobot Roomba Open Interface operates by transmitting and receiving data packets. However, this functionality alone is insufficient for implementing complex logic. Therefore, it is imperative to develop a suitable framework to address this limitation. Fortunately, there exists software specifically engineered for such situations, namely the Robot Operating System (ROS). ROS comprises a suite of software libraries and tools designed to facilitate the development of robotic applications. It is characterized as an open-source, flexible framework that optimizes the process of crafting intricate and robust robotic behaviors across diverse platforms. Its offerings encompass hardware abstraction, low-level device control, interprocess message passing, and package management, thereby simplifying the development and integration of software for robotic applications. There are two iterations of ROS: ROS 1, which is the earlier and more restricted version, and ROS 2, which has garnered wider adoption. The emphasis of this discussion will be on ROS 2, with a detailed examination of its key components to follow subchapters.

5.1. ROS 2 Components

A node is a fundamental executable unit that performs a specific task or computation. Each node in ROS should be responsible for a single, modular purpose, e.g., controlling the wheel motors or publishing the sensor data from a laser range-finder. Each node can send and receive data from other nodes via topics, services, actions, or parameters. A full robotic system is comprised of many nodes working in concert. In ROS 2, a single executable (C++ program, Python program, etc.) can contain one or more nodes. This modular architecture allows developers to build complex robotic applications by combining multiple nodes, each contributing to the overall system's functionality.

Topics are a vital element of the ROS graph that acts as a bus for nodes to exchange messages. A topic is a named communication channel used for exchanging data between nodes in a publish-subscribe model. A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics. This decoupled communication mechanism allows for flexible and scalable system design, as nodes do not need to be aware of each other's existence. Topics are one of the main ways in which data is moved between nodes and, therefore, between different parts of the system. Services are based on a call-and-response model. They allow nodes to subscribe to data streams and get continual updates. Each topic is associated with a specific message type, which defines the structure and content of the data being transmitted.

Launch files allow you to start up and configure several executables containing ROS 2 nodes simultaneously. A launch file is a configuration file that automates the process of starting multiple nodes, setting parameters, and configuring the runtime environment for a robotic application. The launch system in ROS 2 is responsible for helping the user describe the configuration of their system and then execute it as described. The configuration of the system includes what programs to run, where to run them, what arguments to pass them, and ROS-specific conventions, which make it easy to reuse components throughout the system by giving them each a different configuration. It is also responsible for monitoring the state of the processes launched and reporting and/or reacting to changes in the state of those processes. Launch files written in Python, XML, or YAML can start and stop different nodes as well as trigger and act on various events. The package

providing this framework is launch_ros, which uses the non-ROS-specific launch framework underneath. Running a single launch file with the ros2 launch command will start up your entire system, all nodes and their configurations at once. Launch files are essential for streamlining development, testing, and deployment workflows in robotics applications.

A workspace is a directory containing ROS 2 packages. It serves as the central environment for creating, modifying, and compiling code, as well as managing dependencies and resources. Commonly, there is a src subdirectory, which is where the source code of ROS packages is located. Colcon is used to build source code. It is a command-line utility designed to simplify the process of compiling, testing, and managing ROS 2 packages within a workspace. By default, it will create the following directories as peers of the src directory. The build directory will be where intermediate files are stored. For each package, a subfolder will be created where build tools, such as CMake, are invoked. The install directory is where each package will be installed. By default, each package will be installed into a separate subdirectory. The log directory contains various logging information about each colon invocation. A package is the fundamental unit of software organization, containing all the necessary files, code, and resources required to implement a specific functionality or module. With packages, you can release your ROS 2 work and allow others to build and use it easily. Package creation in ROS 2 uses Ament as its build system and Colcon as its build tool. A package typically includes source code, configuration files, dependencies, build system files, documentation, and tests. You can create a package using either CMake or Python, which are officially supported, though other build types do exist.

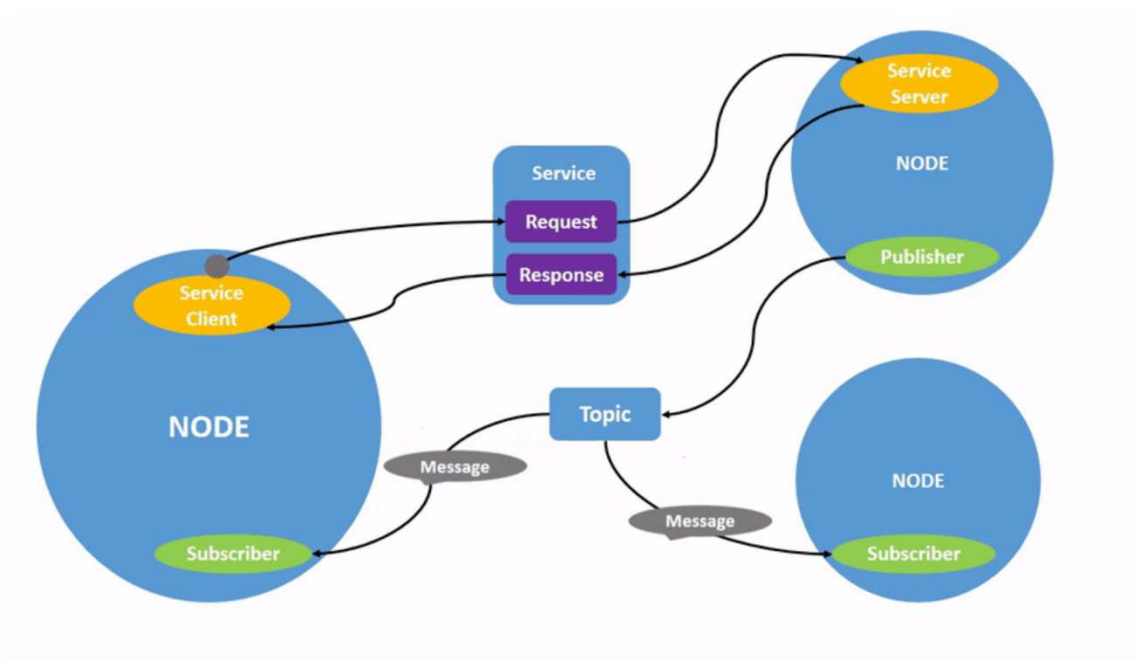

Image 5.1 ROS 2 components

5.2. Unified Robot Description Format

Unified Robot Description Format (URDF) is a file format for specifying the geometry and organization of robots in ROS. It describes a robot's structure, including its links, joints, and their geometric, inertial, and visual properties. Key components include links, joints, transforms, sensors, and actuators. Links are fixed parts of the robot. Some examples are limbs, wheels, or sensors. Links define the mass, inertia, and visual appearance of each part. On the other hand, joints specify the connections between links, including the type of joint and its range of motion, axis of rotation, or translation. Transform is the spatial relationship between links and joints. They enable the calculation of forward and inverse kinematics. Sensors and Actuators describe the placement and properties of sensors and actuators within the robot's structure. URDF files are used by RViz for 3D visualization Gazebo for simulation.

5.3. Tools for Visualization and Simulation

There are two important tools used for robot visualization and simulation. RViz2 is a powerful 3D visualization tool used in ROS 2 to visualize and debug robotic systems. It allows users to display and visualize robot data. It can display sensor data, robot geometry, and movement. Gazebo is a high-fidelity, open-source robotics simulator used to model, test, and validate robotic systems in realistic virtual environments. It has a physics and rendering engine. It can display and simulate sensor data to create complex scenarios. With the help of this tool, there is no need for physical hardware.

6. Roomba Driver

Before initiating the process of constructing the map, a connection between the robot and the computer should be established. Along with that, software for the navigation of the robot and the acquisition of sensor data should be developed. This is no trivial task. The initial phase involves establishing a connection with the robot. Subsequently, it is necessary to manage the transmission and reception of data packets. Furthermore, it is essential to determine the specific conditions under which various packets should be requested. The implementation of odometry is also required to determine the robot's pose. Additionally, transformations between the robot and the world frame must be executed. Furthermore, the creation of a URDF document is necessary to effectively display the robot within a virtual environment. These constitute only a portion of the prerequisites required before commencing the map-building procedure. Fortunately, all of these components are already available to us, allowing for their utilization without the concern of custom implementation.

The create_robot package from AutonomyLab operates as the ROS driver for the iRobot Create 1 and Create 2 models. Given that the Roomba series 600 is constructed on the same platform as the Create 2, the drivers are compatible with this series. This package encapsulates the libcreate C++ library, which conforms to the Open Interface Specification. The primary features include odometry, drive control, bumper, and light sensor display. Although this library encompasses a wide array of functionalities, but most of them are not relevant to this research. Only relevant nodes will be mentioned.

There are two publishers that are required for map building. The first one is published on the bumper topic. The purpose of this node is to fetch bumper and light sensor data from the robot and publish. It publishes a custom message, Bumper.msg. This message contains two boolean values for the left and right bumper sensors. The light sensor is displayed with six boolean values that are used to tell if an obstacle is detected and six unsigned integer values that tell how far away the obstacle is. This is aligned with the Open Interface specification packets. The second publisher is required to determine the robot's pose. It publishes to the topic odom. With the help of wheel encoders, it calculates the odometry of the robot.

Concerning subscribers, only one is relevant for this research. There is a node to move and navigate the robot. The Twist message can be published on the cmd_vel topic, which will order the robot to move. The subscriber listens to this topic and creates the packet to move the robot.

6.1. Odometry Implementation

The create_robot library provides odometry calculations out of the box. The robot used in this project uses protocol version 2. This means that the distance and angle fields are used to calculate odometry. Roomba has the implementation of distance Δx_c. This value is found in the packet with ID 19. The same case is for angle φ. Packet with ID 20 represents this value. The result of distance is in millimeters and angle in degrees. Library libcreate contains file create.cpp which has odometry implementation. Relevant code can be found in the provided code snippet:

```
...
// This is a standards compliant way of doing unsigned to signed conversion
uint16_t distanceRaw = GET_DATA(ID_DISTANCE);
int16_t distance;
std::memcpy(&distance, &distanceRaw, sizeof(distance));
deltaDist = distance / 1000.0; // mm -> m

// Angle is processed differently in versions 1 and 2
uint16_t angleRaw = GET_DATA(ID_ANGLE);
std::memcpy(&angleField, &angleRaw, sizeof(angleField));
...
wheelDistDiff = model.getAxleLength() * deltaYaw;
deltaYaw = angleField * (util::PI / 180.0); // D2R
leftWheelDist = deltaDist - (wheelDistDiff / 2.0);
rightWheelDist = deltaDist + (wheelDistDiff / 2.0);
...
if (fabs(wheelDistDiff) < util::EPS) {
    deltaX = deltaDist * cos(pose.yaw);
    deltaY = deltaDist * sin(pose.yaw);
} else {
    float turnRadius = (model.getAxleLength() / 2.0) * (leftWheelDist +
rightWheelDist) /                    wheelDistDiff;
    deltaX = turnRadius * (sin(pose.yaw + deltaYaw) - sin(pose.yaw));
    deltaY = -turnRadius * (cos(pose.yaw + deltaYaw) - cos(pose.yaw));
}
...
// Update pose
pose.x += deltaX;
pose.y += deltaY;
pose.yaw = util::normalizeAngle(pose.yaw + deltaYaw);
...
```

Code 6.1 – Program for odometry

Some steps in this code snippet differ from formulas in the Odometry chapter. Formula below is proof that odometry result is identical:

$$\Delta x_c = distance\ [mm] = \frac{distance}{1000}\ [m]$$

$$\varphi = \frac{\Delta x_l - \Delta x_r}{L} \rightarrow wheel\ dist.diff. = \Delta x_l - \Delta x_r\ = L * \varphi$$

$$\begin{cases} \Delta x_c = \dfrac{\Delta x_l + \Delta x_r}{2} \\ \varphi = \dfrac{\Delta x_l - \Delta x_r}{L} \end{cases} \rightarrow \Delta x_l = \Delta x_c - \frac{\Delta x_l - \Delta x_r}{2}, \Delta x_r = \Delta x_c - \frac{\Delta x_l - \Delta x_r}{2}$$

$$R = \frac{\Delta x_c}{\varphi} = \frac{\Delta x_l + \Delta x_r}{2} \div \frac{\Delta x_l - \Delta x_r}{L} = \frac{\Delta x_l + \Delta x_r}{2} * \frac{L}{\Delta x_l - \Delta x_r} = \frac{L}{2} * \frac{\Delta x_l + \Delta x_r}{\Delta x_l - \Delta x_r}$$

Left and right wheel distance are calculated because they are needed in other parts of code, but the way of calculating turn radius is not clearly explained.

7. Exploration Strategy

The primary focus of this research is the implementation of an exploration strategy. The concept was developed and refined through multiple phases, each becoming increasingly complex and efficient.

The initial phase represents the most basic form of the strategy. The robotic movement involves two options: proceeding forward or rotating. The distance of the forward movement is fixed, while the rotation angle is predetermined. The selection between these two options occurs randomly, and the newly chosen option cannot be executed until the previously selected option has been completed. Upon completing the selected option, the robot processes sensor data and records this information on a map. Subsequently, the process of selecting one of the two options recommences. This iterative process continues until a comprehensive map of the environment is generated. The efficiency of generating an algorithm in this manner can be likened to the process of sorting an array using a bogsort algorithm. While the mapping can be completed in a finite duration, the speed of creation is influenced by chance due to the method of movement selection. A challenge arises when the robot attempts to move forward in the presence of an obstacle; localization will inaccurately assume that the robot has successfully moved the predetermined distance.

The second phase of the algorithm involves bifurcating the rotation to the left and the right, while the rotation angle remains unchanged. Thus, the algorithm now has three options to choose from. This additional option allows the robot to generate a more precise map, although this remains contingent upon a random factor.

In the subsequent phase, sensor data is incorporated to mitigate the localization issue. The robot now reads and processes data before moving forward. If an obstacle is detected directly ahead, the robot refrains from advancing and selects one of the three options. This enhancement improves localization, albeit with a minimal gain in overall efficiency.

To further enhance efficiency, algorithms are designed to process data after selecting any option. If an obstacle is detected solely in the direction of rotation, the robot will not execute a rotation. Conversely, if obstacles are detected on both sides, the robot will continue to rotate in the selected direction, thereby preventing movement into an obstruction.

The selection process must undergo a significant transformation to refine the algorithm further. The robot's movement will now be entirely contingent on sensor readings. The robot will advance until an obstacle is detected in its path. Upon detection, the robot will rotate in one direction and continue rotating until no obstacle is identified in front of it.

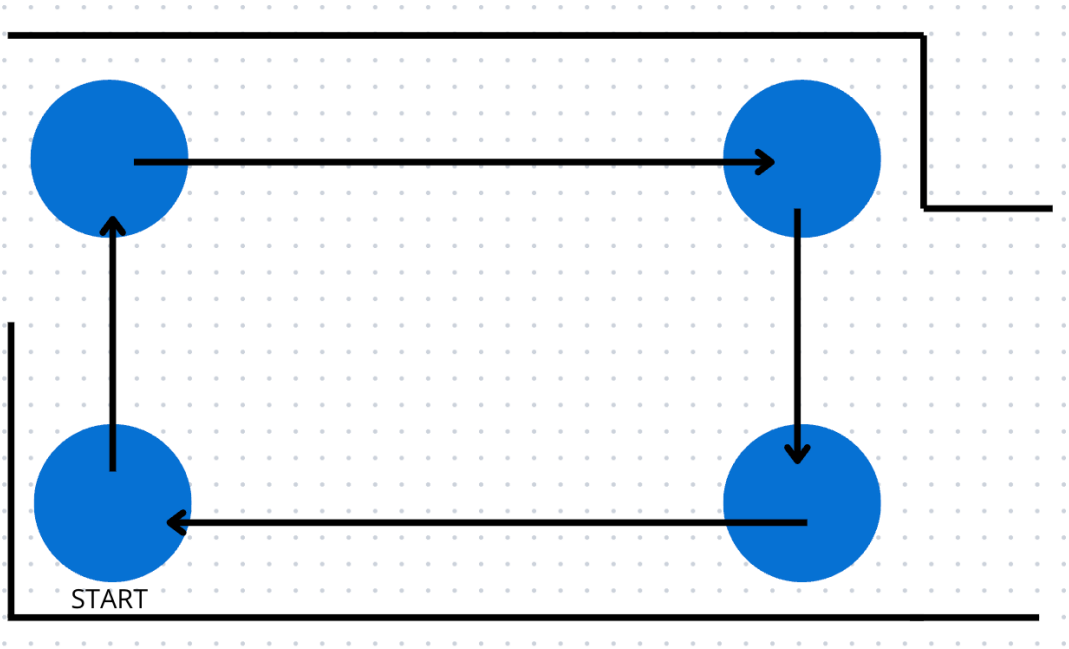

Image 7.1Example of closed loop

This revised approach introduces a new issue; when navigating an environment as depicted in the illustration above, the robot may enter a perpetual loop, preventing the completion of the entire mapping process. To circumvent this, an edge-following algorithm will be employed. The implementation of this algorithm is as follows: when an obstacle is detected, the robot will rotate 90 degrees in relation to the edge of the obstacle, utilizing sensor data to guide this action. Thereafter, the robot will move forward. Following each movement, it will verify whether a 90-degree alignment remains between itself and the obstacle's edge. If this alignment is not maintained, corrective action will be taken.

When the robot encounters a corner, it will detect an obstacle both in its front and side. In such circumstances, the robot will rotate in the opposite direction to the obstacle until achieving a 90-degree angle relative to the obstacle detected in front of it. It will then proceed to move forward. A further scenario may involve the robot reaching the terminus of an edge. In this case, the robot must execute several steps to continue following the edge of the object. Initially, it needs to cover a forward distance equal to its length plus a small constant.

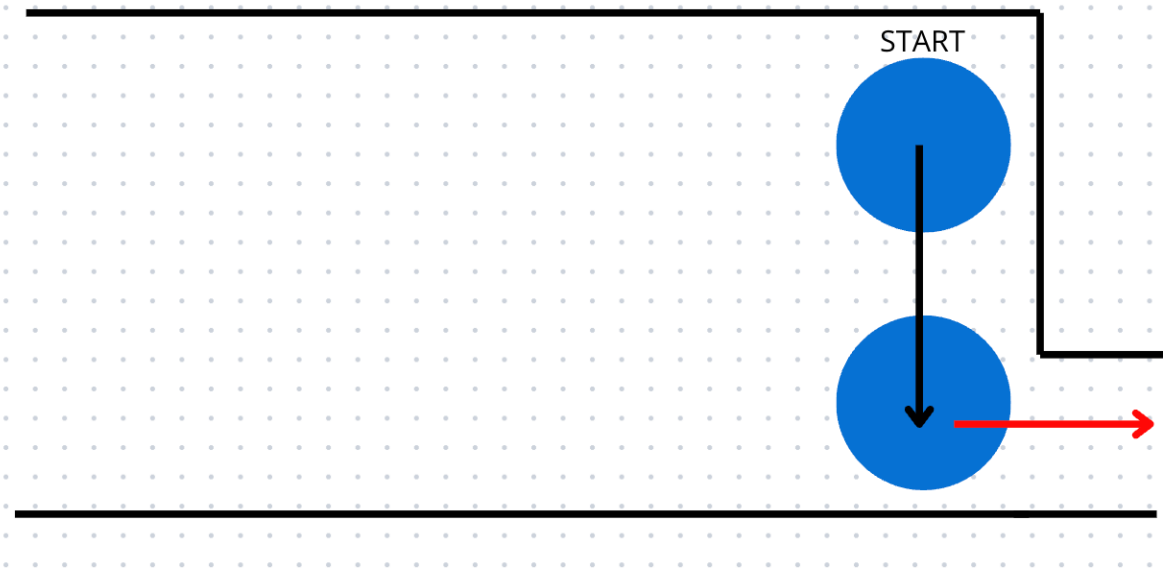

Image 7.2 Example of too narrow space

If an obstacle is encountered ahead of time during this distance, the robot will initiate the edge-following process. This precaution is warranted because the space is potentially insufficient for the robot to navigate through. If the robot identifies no obstacle ahead, it will rotate 90 degrees towards the detected obstacle and subsequently repeat the process of covering the forward distance.

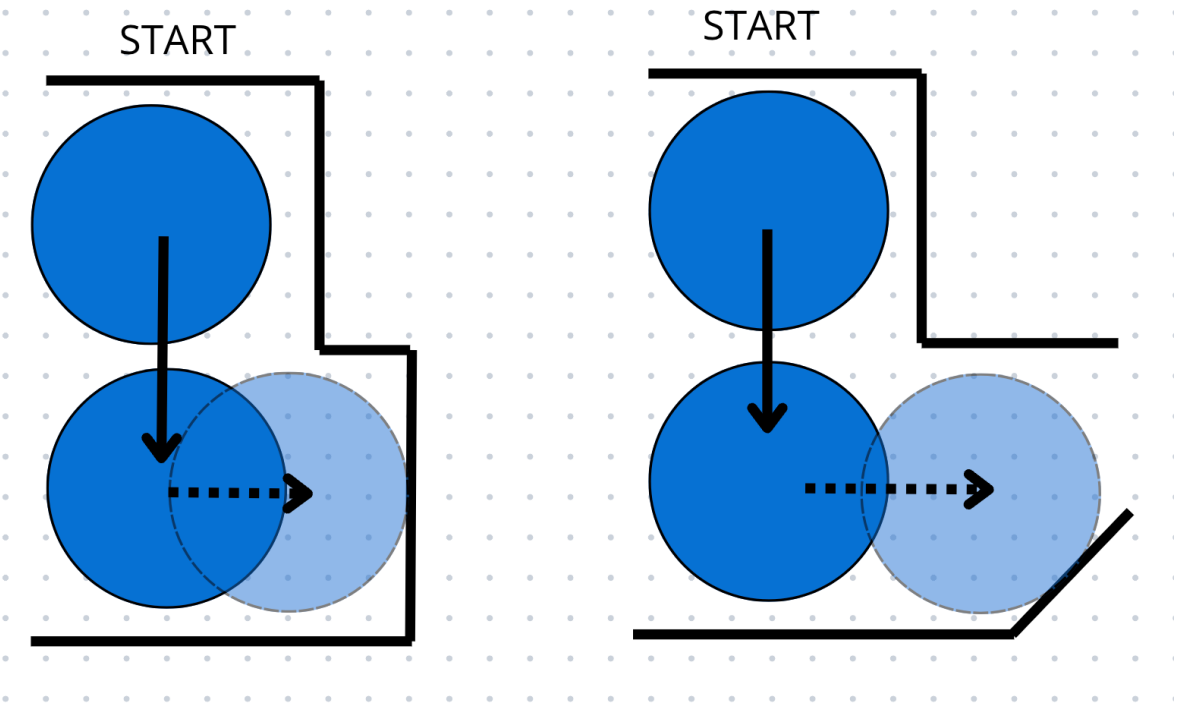

Image 7.3 Example of narrow passage

If an obstacle is encountered in front of or directly opposite the obstacle, the robot will rotate until it achieves a 90-degree alignment with the edge of the opposing obstacle and continue to follow that edge. If this alignment does not occur, the robot will check for

detecting the obstacle's edge. If it remains undetected, the robot will move in a zig-zag pattern until the sensor successfully detects the edge.

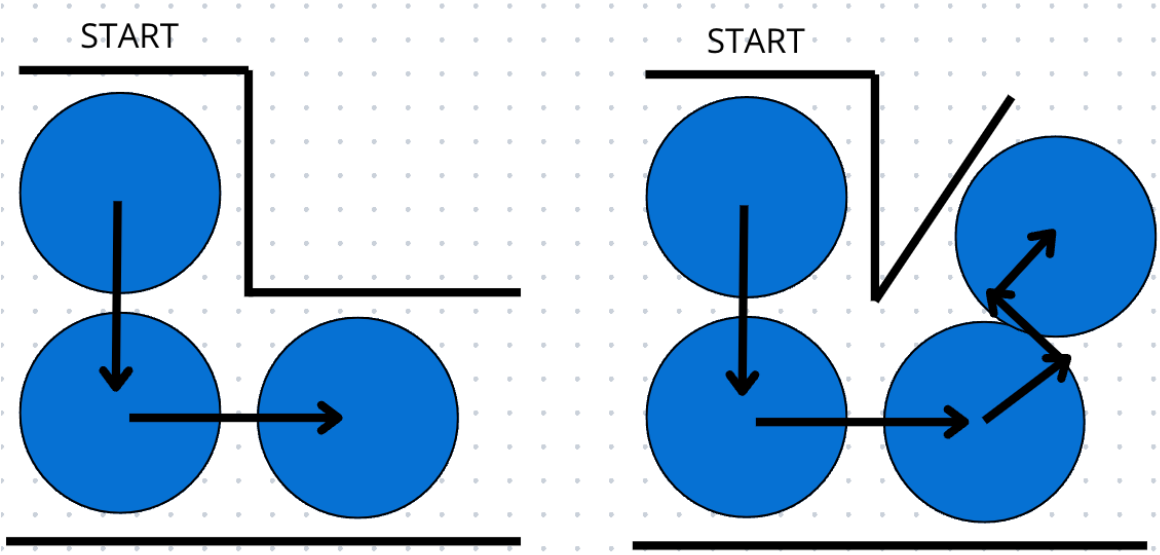

Image 7.3 Example of successfully traversed corner

The final algorithm is implemented in the method move_to_empty_space ().

7.1. Code Implementation

In this section, there will be code snippets and explanation. First part will be helper classes, and the second part explains the algorithm implementation.

```
...
class PositionEnum(Enum):
    LEFT = -1
    NONE = 0
    RIGHT = 1
...
```

This enum contains possible positions of the object relative to the robot. It is also used to determine in which direction the robot should rotate.

```
...
class CellStatus(Enum):
    UNKNOWN = 0
    EMPTY = 127
    OCCUPIED = 100
...
```

This enum contains possible statuses of each cell of the grid.

```
...
class Position():
    def __init__(self):
        self.position = PositionEnum.NONE

    def is_left(self):
        return self.position == PositionEnum.LEFT

    def is_right(self):
        return self.position == PositionEnum.RIGHT

    def is_none(self):
        return self.position == PositionEnum.NONE

    def set_position(self, value: PositionEnum):
        if not isinstance(value, PositionEnum):
            raise TypeError(
                f"value must be of type 'PositionEnum', not '{type(value).__name__}'")
        self.position = value

    def switch(self):
        if self.position == PositionEnum.NONE:
            return

        if self.position == PositionEnum.LEFT:
            self.position = PositionEnum.RIGHT
```

```
        elif self.position == PositionEnum.RIGHT:
            self.position = PositionEnum.LEFT
    ...
```

Purpose of the helper class Position is to maintain where the obstacle is located relative to the robot. It contains three methods is_*() to determine if an obstacle is located at a different position. The obstacle can be located on only one of three positions at any time. Method switch() enables moving the obstacle position from one side to another. This is useful when the robot is located in the corner.

```
    ...
    class Rotation(Position):
        def __init__(self):
            super().__init__()
            self.rotation = 0

        def increase_counter(self):
            self.rotation += 1

        def reset_counter(self):
            self.rotation = 0

        def get_counter(self):
            return self.rotation

        def set_position(self, value):
            super().set_position(value)
            self.reset_counter()

        def switch(self):
            super().switch(self)
            self.reset_counter()
    ...
```

Class Rotation memorizes in which direction the robot needs to rotate. The robot can't continuously rotate for a large angle because of security reasons. With the help of this class, that problem is overcome. It has a private variable that counts how much time the rotation command was sent to the robot. This is used to reset the robot's status if it gets stuck in infinite rotation.

```
    ...
    class CreateMap(Node):
    ...
```

Next snippets of code will be part of the main class CreateMap that contains all logic for exploration strategy.

```
...
    def __init__(self):
        super().__init__('create_map')

        self.map_pub = self.create_publisher(
            OccupancyGrid,
            'map',
            10
        )
        self.cmd_vel_publisher = self.create_publisher(
            Twist,
            'cmd_vel',
            10
        )

        self.subscription = self.create_subscription(
            Bumper,
            'bumper',
            self.bump_callback,
            10
        )
        self.create_subscription(
            Odometry,
            '/odom',
            self.odom_callback,
            10
        )
...
```

First part of the initialization method consists of creating publishers and subscribers. First publisher is map_pub. Its purpose is to maintain the occupancy grid map. Publisher cmd_vel_publisher takes care of commanding the robot's movement. Method bump_callback() is part of the bumper subscription and method odom_callback() is a part of the odom subscription.

```
...
    def __init__(self):

        # Robot pose
        self.robot = Pose()

        self.robot_offset = {
            (0, 0),
            (0, -1),
            (-1, 0),
            (-1, -1)
        }

        # Grid parameters
        self.grid_size = 100
```

```
        self.resolution = 0.1
        self.grid = [CellStatus.UNKNOWN.value] * \
            (self.grid_size * self.grid_size)

        self.origin = Pose()
        self.origin.position.x = -self.grid_size * self.resolution / 2.0
        self.origin.position.y = -self.grid_size * self.resolution / 2.0

        self.light_offsets = {
            "is_light_left": (0, 1),
            "is_light_front_left": (1, 1),
            "is_light_center_left": (1, 0),
            "is_light_center_right": (1, -1),
            "is_light_front_right": (1, -2),
            "is_light_right": (0, -2),
        }

        self.bumper = Bumper()

        self.obstacle = Position()
        self.rotation = Rotation()

        self.edge_detected = 0
        self.forward_counter = 0
        self.random_number = -1

        timer_period = 0.1

        self.timer = self.create_timer(timer_period, self.check_status)

        self.get_logger().info("Create Map Initialized")
    ...
```

The robot occupies four cells at any time. This is defined with the robot_offset variable. Occupancy grid is defined as a 100x100 grid with the starting cell's value as unknown. This is represented as a list with 1000 elements. Starting position of the robot is in the middle of the map. Light sensors detect an obstacle in one cell in front, the middle, and the sides of the robot position. This is defined with self.light_offsets. Variable bumper stores current sensor readings, variables obstacle, rotation, and instances of the Position and Rotation classes. Variable edge_detected is used as a flag to determine if the robot encountered the obstacle's edge. Variable forward_counter counts how much the robot moved forward. Variable random_number is used to break infinite rotation if the robot gets stuck. Timer_period is used in the timer, which every 0.1 seconds invokes the check_status() method.

```
...
    def odom_callback(self, msg):
        current_pose = msg.pose.pose
        self.robot.position.x = (current_pose.position.x -
                                 self.origin.position.x) / self.resolution
        self.robot.position.y = (current_pose.position.y -
                                 self.origin.position.y) / self.resolution
        self.robot.orientation.z = current_pose.orientation.z
...
```

Method odom_callback updates robot's current position with odometry results.

```
...
    def bump_callback(self, msg):
        self.bumper = msg
        self.process_light_sensors(msg)
        self.publish_map()
...
```

Method bump_callback updates bumper values with sensor data, processes that data, and updates the map.

```
...
    def process_light_sensors(self, msg):
        for sensor, offset in self.light_offsets.items():
            if getattr(msg, sensor):
                self.mark_obstacle(offset)

    def mark_obstacle(self, offset):
        self.mark_on_map(offset, CellStatus.OCCUPIED)
...
```

If any light sensor detects an obstacle, it will mark that spot as occupied.

```
...
    def mark_on_map(self, offset, cellStatus: CellStatus):
        dx, dy = offset
        position_x = self.robot.position.x + dx
        position_y = self.robot.position.y + dy

        center_x = self.robot.position.x - 0.5
        center_y = self.robot.position.y - 0.5

        if (-0.2 >= self.robot.orientation.z and self.robot.orientation.z >= -0.8):
            temp_x = position_x - center_x
            temp_y = position_y - center_y

            temp_x_2 = -temp_y
            temp_y_2 = temp_x
```

```
            position_x = temp_x_2 + center_x
            position_y = temp_y_2 + center_y

        if (0.8 >= self.robot.orientation.z and self.robot.orientation.z >= 0.2):
            temp_x = position_x - center_x
            temp_y = position_y - center_y

            temp_x_2 = temp_y
            temp_y_2 = -temp_x

            position_x = temp_x_2 + center_x
            position_y = temp_y_2 + center_y

        if (-0.8 > self.robot.orientation.z or self.robot.orientation.z > 0.2):
            temp_x = position_x - center_x
            temp_y = position_y - center_y

            temp_x_2 = -temp_x
            temp_y_2 = -temp_y

            position_x = temp_x_2 + center_x
            position_y = temp_y_2 + center_y

        index = self.grid_index(position_x, position_y)
        if 0 <= index < len(self.grid) and self.grid[index] != cellStatus.value:
            self.grid[index] = cellStatus.value
...
```

Method mark_on_map() finds appropriate cells and marks them. Position of the cell is calculated with the robot's current position and required offset. If the robot's orientation is 0.5, that means that the robot is rotated for 90 degrees clockwise, and if the value is -0.5, then the robot is rotated for 90 degrees counter-clockwise. Value of 1 represents a rotation of 180 degrees. If the rotation value is between 0.2 and 0.8, then it is assumed that the robot is rotated for 90 degrees clockwise. The value between -0.8 and -0.2 assumes that the robot is rotated for 90 degrees counter-clockwise. This is not the real rotation, but this assumption makes easier calculation. Rotation matrix is used to determine the correct cell. The middle of the robot is used as the center of rotation. The cell is updated if the index is in range and the new value differs from the old one.

```
...
    def grid_index(self, x, y):
        return int(y) * self.grid_size + int(x)
...
```

Method grid_index() transfers 2D coordinates to one-dimensional space.

```
...
  def mark_robot(self):
      for offset in self.robot_offset:
          self.mark_on_map(offset, CellStatus.EMPTY)
...
```

Method mark_robot() marks robot position in the grid as empty.

```
...
  def check_status(self):
      self.move_to_empty_space()
      self.publish_map()
...
```

Method check_status() implements exploration strategy and publishes updated map.

```
...
   def publish_map(self):
      map_msg = OccupancyGrid()
      map_msg.header.frame_id = "map"
      map_msg.info.resolution = self.resolution
      map_msg.info.width = self.grid_size
      map_msg.info.height = self.grid_size
      map_msg.info.origin = self.origin

      map_msg.header.stamp = self.get_clock().now().to_msg()
      map_msg.data = self.grid

      self.map_pub.publish(map_msg)
...
```

Method publish_map() publishes the current grid with timestamp.

```
...
   def reset_status(self):
      self.get_logger().info("Resetting status")
      self.rotation.set_position(PositionEnum.NONE)
      self.forward_counter = 0
      self.edge_detected = 0
...
```

Method reset_status() resets all variables relevant for edge detection.

```
...
    def move_to_empty_space(self):
        # rotate for random amount and reset status
        if self.rotation.get_counter() == self.random_number:
            self.reset_status()
            self.obstacle.set_position(PositionEnum.NONE)
            return

        # robot rotated for full circle
        # start reset proccess
        if self.rotation.get_counter() >= 100:
            if self.rotation.get_counter() == 100:
                self.random_number = random.randint(100, 200)
                self.rotation.increase_counter()
                return

            if self.rotation.get_counter() == self.random_number:
                self.reset_status()
                self.obstacle.set_position(PositionEnum.NONE)
                self.move_forward
                return
...
```

If the robot gets stuck and is not able to move forward after rotating for a full circle, then a random value that can be approximately between 0 and 360 degrees is chosen. When that value is reached, the robot's status is reset and it stops following the obstacle edge. It will start moving forward until it finds new obstacle.

```
            ...
                def move_to_empty_space(self):
                    ...
                    # robot came to end of object edge
                    # start the process of moving around the edge in L direction
                    if self.edge_detected == 1 and self.rotation.get_counter() > 1:
                        # narrow corner, go back
                        if self.obstacle.is_right() and self.bumper.is_left_pressed or
            self.obstacle.is_left() and self.bumper.is_right_pressed:
                            self.obstacle.switch()
                            self.reset_status()
                            return

                        # object edge is detected again
                        # exit detection of object edge
                        if self.obstacle.is_right() and (self.any_right_lights_pressed()) or
            self.obstacle.is_left() and (self.any_left_lights_pressed()):
                            self.reset_status()
                            return

                        if self.obstacle.is_right() and self.bumper.is_right_pressed:
                            self.turn_left()
                            return
```

```
            if self.obstacle.is_left() and self.bumper.is_left_pressed:
                self.turn_right()
                return

            # move forward 9 times
            if self.forward_counter < 10:
                self.move_forward()
                self.forward_counter += 1
                return

            # go to next step
            self.rotation.reset_counter()
            self.forward_counter = 0
            self.edge_detected = 2
            return
    ...
```

When a robot detects an edge, it needs to take three different steps to move past it. Before the first step, it rotates into the obstacle direction and tries to find an obstacle with the light sensor. If that fails, then the first step begins. It moves forward 9 times. If at any point it encounters an obstacle with its bumper sensor on the same side, it needs to move away from it before continuing to move forward. If it encounters an obstacle on the opposite side, then that means that it cannot enter the other side of the edge. It stops following the current obstacle's edge and starts following the new encountered edge.

```
    ...
        def move_to_empty_space(self):
            ...
            if self.edge_detected == 2 and self.rotation.get_counter() > 14:
                # narrow corner, go back
                if self.obstacle.is_right() and self.bumper.is_left_pressed or
    self.obstacle.is_left() and self.bumper.is_right_pressed:
                    self.obstacle.switch()
                    self.reset_status()
                    return

                # object edge is detected again
                # exit detection of object edge
                if self.obstacle.is_right() and (self.any_right_lights_pressed() or
    self.bumper.is_right_pressed) or self.obstacle.is_left() and
    (self.any_left_lights_pressed() or self.bumper.is_left_pressed):
                    self.reset_status()
                    return

                self.edge_detected = 3
                self.rotation.reset_counter()
                self.forward_counter = 0
                return
    ...
```

The second step is to rotate 15 times. If it manages to encounter an obstacle from the opposite side, then the whole process stops and steps from previous step are implemented.

```
...
    def move_to_empty_space(self):
        ...
        # start moving in zig-zag pattern towards objects edge
        # rotate three two times, then move forward
        if self.edge_detected == 3:
            # narrow corner, go back
            if self.obstacle.is_right() and self.bumper.is_left_pressed or
self.obstacle.is_left() and self.bumper.is_right_pressed:
                self.obstacle.switch()
                self.reset_status()
                return

            # object edge is detected again
            # exit detection of object edge
            if self.obstacle.is_right() and (self.bumper.is_light_right or
self.bumper.is_light_center_right or self.bumper.is_right_pressed) or
self.obstacle.is_left() and (self.bumper.is_light_left or
self.bumper.is_light_center_left or self.bumper.is_left_pressed):
                self.reset_status()
                return

            self.rotation.increase_counter()

            if self.rotation.get_counter() % 3 != 0:
                self.move_forward()
                return

            self.turn_right()
            return
...
```

Third step is for every three movements forward, the robot needs to rotate itself until it reaches the obstacle's edge. If it encounters the obstacle on the opposite side, then this process is stopped and it implements the case from the first two steps.

```
...
def move_to_empty_space(self):
        ...
        # rotate robot right until obstacle is only on the left side
      if (self.rotation.is_right()):
        if self.obstacle.is_left() and self.only_left_light_pressed() or
self.obstacle.is_right() and self.only_right_light_pressed():
            self.reset_status()
            self.move_forward()
            return

        self.turn_right()
        self.rotation.increase_counter()
        return

       # rotate robot left until obstacle is only on the right side
      if (self.rotation.is_left()):
        if self.obstacle.is_left() and self.only_left_light_pressed() or
self.obstacle.is_right() and self.only_right_light_pressed():
            self.reset_status()
            self.move_forward()
            return

        self.turn_left()
        self.rotation.increase_counter()
        return
...
```

When the robot has rotation set in one direction, it will rotate in this direction until only the side light sensor closest to the obstacle is detecting it.

```
...
def move_to_empty_space(self):
      ...
      if self.bumper.is_left_pressed and self.bumper.is_right_pressed:
        # obstacle is not detect then find where it is and rotate robot until only its
edge sees it
        if self.obstacle.is_none():
          # obstacle on the right
          if self.bumper.is_light_front_right:
            self.get_logger().warn("Obstacle detected on the right")
            self.rotation.set_position(PositionEnum.LEFT)
            self.obstacle.set_position(PositionEnum.RIGHT)
            return

          # obstacle on the left
          if self.bumper.is_light_front_left:
            self.get_logger().warn("Obstacle detected on the left")
            self.rotation.set_position(PositionEnum.RIGHT)
```

```
                    self.obstacle.set_position(PositionEnum.LEFT)
                    return

            # obstacle already detected and both bumpers are pressed
            # robot  approached the corner of obstacle
            # rotate itself in opposite direction to continue following objects edge
            if self.obstacle.is_left():
                self.get_logger().warn("obstacle already detected and both bumpers are
    pressed")
                self.rotation.set_position(PositionEnum.RIGHT)
                return

            if self.obstacle.is_right():
                self.get_logger().warn("obstacle already detected and both bumpers are
    pressed")
                self.rotation.set_position(PositionEnum.LEFT)
                return

            self.get_logger().info("Both bumpers are pressed, but obstacle is not
    detected")
            exit()
            return
    ...
```

When both bumper sensors are pressed, the robot has a couple of choices. If it did not detect the obstacle until now or it forgot about it, then it checks which side is closest to the obstacle and sets it to that side. It needs to rotate itself in the opposite direction. If an obstacle was already detected, that means that it got too close to its edge. It needs to rotate in the opposite direction to move away from it.

```
...
def move_to_empty_space(self):
    ...
    if self.bumper.is_right_pressed and self.obstacle.is_none():
        self.get_logger().info("Obstacle detected on the right")
        self.rotation.set_position(PositionEnum.LEFT)
        self.obstacle.set_position(PositionEnum.RIGHT)
        return

    if self.bumper.is_left_pressed and self.obstacle.is_none():
        self.get_logger().info("Obstacle detected on the left")
        self.rotation.set_position(PositionEnum.RIGHT)
        self.obstacle.set_position(PositionEnum.LEFT)
        return
...
```

If the left or right bumper sensor is pressed and an obstacle was not detected until now, then it needs to set and the rotation in the opposite directions begins.

```
...
def move_to_empty_space(self):
    ...
    if self.obstacle.is_left() and self.bumper.is_left_pressed:
        self.get_logger().info("Left bumper pressed, rotating right")
        self.turn_right()
        return

    if self.obstacle.is_right() and self.bumper.is_right_pressed:
        self.get_logger().info("Right bumper pressed, rotating left")
        self.turn_left()
        return
...
```

When one of the bumpers is pressed and the obstacle is already located on that side, robot needs to move from its edge.

```
...
def move_to_empty_space(self):
    ...
    # it is possible that robot came to end of the edge
    # rotate to the opposite side up to two times and try to detect object
    if self.obstacle.is_left() and not self.any_left_lights_pressed():
        self.get_logger().info("it is possible that robot came to end of the edge")
        self.rotation.set_position(PositionEnum.LEFT)
        self.edge_detected = 1
        return

    # it is possible that robot came to end of the edge
    # rotate to the opposite side up to two times and try to detect object
    if self.obstacle.is_right() and not self.any_right_lights_pressed():
        self.get_logger().info("it is possible that robot came to end of the edge")
        self.rotation.set_position(PositionEnum.RIGHT)
        self.edge_detected = 1
        return
...
```

When a robot loses an obstacle, it assumes that it encounters the end of the edge. It starts the process of edge detection.

```
...
def move_to_empty_space(self):
    ...
    if self.obstacle.is_left() and self.bumper.is_light_front_left and not
self.bumper.is_light_left:
        self.get_logger().info("Lost left light rotate left")
        self.rotation.set_position(PositionEnum.RIGHT)
        return
```

```
            if self.obstacle.is_right() and self.bumper.is_light_front_right and not
        self.bumper.is_light_right:
                self.get_logger().info("Lost right light rotate right")
                self.rotation.set_position(PositionEnum.LEFT)
                return

        def move_to_empty_space(self):
            ...
            # robot left side lost tract of objects edge
            if self.obstacle.is_left() and not self.bumper.is_light_left:
                self.get_logger().info("robot left side lost tract of objects edge")
                self.rotation.set_position(PositionEnum.LEFT)
                return

            # robot right side lost tract of objects edge
            if self.obstacle.is_right() and not self.bumper.is_light_right:
                self.get_logger().info("robot right side lost tract of objects edge")
                self.rotation.set_position(PositionEnum.RIGHT)
                return
        ...
```

In this case, the robot starts rotation to try to get back on the track.

```
        ...
        def move_to_empty_space(self):
            ...
            self.move_forward()
        ...
```

By default, the robot should move forward.

```
        ...
            def only_left_light_pressed(self):
                return self.bumper.is_light_left and not self.bumper.is_light_front_left and not
        self.bumper.is_light_center_left

            def only_right_light_pressed(self):
                return self.bumper.is_light_right and not self.bumper.is_light_front_right and
        not self.bumper.is_light_center_right

            def any_left_lights_pressed(self):
                return self.bumper.is_light_left or self.bumper.is_light_front_left or
        self.bumper.is_light_center_left

            def any_right_lights_pressed(self):
                return self.bumper.is_light_right or self.bumper.is_light_front_right or
        self.bumper.is_light_center_right
        ...
```

This are the couple of helper methods.

```
...
 def turn_left(self):
        self.move_robot(z=0.5)
        self.get_logger().info('Rotating left.\n')

    def turn_right(self):
        self.move_robot(z=-0.5)
        self.get_logger().info('Rotating right.\n')

    def move_forward(self):
        self.mark_robot()
        self.move_robot(x=0.1)
        self.get_logger().info('Moving forward.\n')
...
```

Methods turn_left(), turn_right() and move_forward() use different positioning to move the robot.

```
...
    def move_robot(self, x=0.0, z=0.0):
        move_msg = Twist()
        move_msg.linear.x = x
        move_msg.angular.z = z
        self.cmd_vel_publisher.publish(move_msg)
...
```

Method move_robot() publishes appropriate message to cmd_vel topic.

```
...
def main(args=None):
    rclpy.init(args=args)
    node = CreateMap()
    rclpy.spin(node)
    rclpy.shutdown()


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

This is the default startup of ROS2 node.

Conclusion

This study successfully demonstrated the application of theoretical knowledge to enhance and expand the functionality of older models of robotic vacuum cleaners. By integrating advanced mapping capabilities, it has become possible to adapt features typically found in newer, high-end models to older devices. Key improvements include creating and utilizing environmental maps. It enables functionalities such as detecting and marking soiled areas, locating the robot in case of interruptions, and displaying cleaned and remaining sections of the space. These advancements extend the lifespan and utility of older robotic vacuum cleaners and provide a cost-effective solution for users seeking to upgrade their devices without purchasing new models. Implementing these features within the Robot Operating System (ROS) framework underscores the feasibility and effectiveness of the proposed methodology. By leveraging sensor fusion and optimized exploration strategies, this research has overcome the inherent limitations of older hardware, resulting in improved map accuracy and coverage. This work contributes to the broader field of consumer robotics by bridging the gap between theoretical advancements and practical applications, paving the way for future innovations in enhancing legacy robotic systems. Ultimately, this study highlights the potential for integrating modern robotics principles into older devices, offering a sustainable and efficient approach to improving their performance and user experience.

Literature

[1] Computer Science Department and Robotics Institute, Carnegie Mellon University, Pittsburgh, Learning metric-topological maps for indoor mobile robot navigation

[2] Huang S., Dissanayake G. *Robot Localization: An Introduction*

[3] Mahony R., Hamel T., Trumpf J. *An homogeneous space geometry for simultaneous localisation and mapping*

[4] Thrun S., *Learning metric-topological maps for indoor mobile robot navigation*

[5] Balch T., *Grid-based Navigation for Mobile Robots*

[6] Diginsa Umar M., Shafie N., Review: Issues and Challenges of Simultaneous Localization and Mapping (SLAM) Technology in Autonomous Robot

Summary

Enhancing Robotic Vacuum Cleaner Performance through Improved Map Building

This thesis aims to enhance the performance of older models of robotic vacuum cleaners by improving the space mapping procedure despite limited sensor capabilities and exploration strategies. The work primarily focuses on developing a method for creating maps of their environment despite limited sensor data. The aim is to design a space exploration strategy suitable for older robotic vacuum cleaners' computational and sensor capabilities. The proposed methodology uses sensor fusion to overcome sensor limitations, resulting in improved map accuracy while maximizing coverage. The implementation of this approach will be carried out within the Robot Operating System (ROS) environment.

keywords: mapping, localization, robot vacuum cleaner, ROS, mapping algorithms, odometry, sensors

Sažetak

Unaprjeđenje performansi robotskih usisavača putem poboljšane izgradnje karata

Cilj ovog diplomskog rada je unaprijediti performanse starijih modela robotskih usisavača poboljšanjem procedure izrade karata prostora uz ograničene senzorske sposobnosti i strategije istraživanja prostora. Rad se primarno fokusira na razvoj metode stvaranja karata njihove okoline unatoč ograničenim senzorskim podacima. Cilj je osmisliti strategiju istraživanja prostora prikladnu za računalne i senzorske mogućnosti starijih robotskih usisavača. Korištenjem fuzije senzora, predložena metodologija nastoji prevladati senzorska ograničenja, što rezultira poboljšanom točnošću karata uz maksimizaciju pokrivenosti. Implementacija ovog pristupa provest će se unutar okruženja Robot Operating System (ROS), a potom će biti validirana putem simulacija.

ključne riječi: izgradnja karata, lokalizacija, robotski usisavač, ROS, algoritmi izrade karata, odometrija, senzori