M.Eng in Information Technology
Autonomous Intelligent Systems
Supervised by: Prof. Dr. Peter Nauth

Localization in autonomous navigation (including kidnapped robot problem) with ROS 2

Md Zahid Hasan (1396470)
md.hasan5@stud.fra-uas.de

Md Mosharraf Hossain (1386448)
md.hossain3@stud.fra-uas.de

Abstract— This paper presents an approach for autonomous navigation on a Turtlebot3 that includes detecting and handling the "Kidnapped Robot Problem" using a SLAM algorithm. The robot's logic, motion control, and data processing are all managed through the Robot Operating System (ROS2), which simplifies creating multi-threaded programs that communicate over a high-speed TCP/IP network. In this work, the Turtlebot3 is simulated in Gazebo for testing, and Rviz is used to display sensor data. The robot's map is generated by the Gmapping package, which uses inputs from LIDAR sensors (laser scans) and odometry. To navigate in this mapped environment, the Turtlebot3 relies on open-source navigation tools, and its position is estimated using Monte Carlo Localization (MCL). MCL does not need any initial knowledge of where the robot starts; instead, it constantly updates the robot's position by analyzing sensor information and using probabilistic methods. A key contribution of this paper is detecting when the robot has been "kidnapped" (moved unexpectedly). The proposed detection strategy evaluates the difference in particle weights, the maximum weight among particles, and the standard deviation of these weights. By combining these parameters, we can recognize unexpected movements more accurately than if we only looked at a single factor, like the maximum weight. Six different similarity-based measures are also tested and compared with particle-weight-based detectors. The results indicate that these similarity-based detectors more reliably recognize both normal and kidnapped conditions. Furthermore, they do not reduce the accuracy of relocalization, meaning the robot can still correctly re-establish its position after it is moved.

Keywords—ROS2, MCL kinetic, Gazebo, TurtleBot3, LIDAR, Kidnapped Robot Problem (KRP).

I. INTRODUCTION

Autonomous navigation robots need to figure out where they are in order to move safely and complete tasks. Their use is growing in industrial settings, partly because they can operate in dangerous environments with minimal human involvement. To function correctly, these robots must address several key challenges, including creating maps of their surroundings (mapping), figuring out their own positions (localization), and planning routes (path planning). Each of these problems can be approached with different methods.

In many cases, robots cannot rely on the Global Positioning System (GPS). While GPS is widely used for locating and navigating both people and robots, it does not work well in certain environments. For instance, GPS signals may be blocked by the thick walls of a large building (like a hospital or school), and tall buildings in city centers can interfere with GPS reception. Even when GPS signals are available, they may not be accurate enough or may not provide updates often enough for the robot's needs. To overcome these limitations, autonomous robots often use techniques from robot localization. These techniques let the robot figure out where it is at any moment, even without GPS, by combining: A map of the robot's environment, Sensor data (for example, from wheel encoders or laser scanners), Motion models based on the robot's design (kinematics), which describe how the robot's movements affect its position.

One particular challenge is called the "kidnapped robot problem," which happens when a robot is suddenly moved to a new place without its internal systems knowing. Detecting this event is difficult, especially for methods like Monte Carlo Localization (MCL) [7]. MCL uses particle filters, where many possible positions (particles) are tested. When these particles converge on a single location, the robot assumes it knows where it is. But if the robot is unexpectedly moved to a different area with few or no particles, the robot may have trouble correcting this mistake.

In real life, a robot does not get "kidnapped" very often; however, this scenario is frequently used to test if a localization method can recover from errors. Additionally, certain mechanical or sensor malfunctions can mimic a "kidnapping" scenario. Recognizing these situations can also serve as a form of fault detection, alerting the robot's control system that something unusual has happened and needs attention.

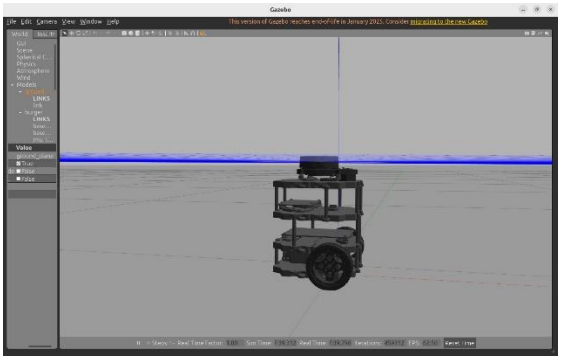

Fig. 1: Turtlebot3 in Gazebo

II. LITERATURE REVIEW

A. Cherubini et al. [1], introduce a new techniques for identifying the "kidnapped robot" problem within Monte Carlo

Localization (MCL). Their approach analyzes the robot's sensor data to determine whether a sudden movement is a typical, expected change or an unusual event—such as when the robot is unexpectedly moved to a different location. To validate their method, the authors designed a series of simulations that evaluate detection accuracy and compare its performance against other existing techniques.

In their Research Article, B. M. F. da Silva et al. [2], delineates a solution for the global localization of a kidnapped robot by adapting 2D SURF (Speeded Up Robust Features) image features to create 3D landmarks for the environment. The proposed algorithm constructs a prior map consisting solely of these 3D SURF landmarks. When the robot is reactivated in an unknown location, it begins by searching for these landmarks, which are then compared to the pre-mapped ones using SURF descriptors. The displacement between the detected online landmarks and the offline landmarks is estimated using the RANSAC random sampling method. This approach not only robustly estimates the robot's position but also effectively reduces false detections by eliminating incorrectly associated landmarks.

A. Majdik et al [3], in their research paper 'New approach in solving the kidnapped robot problem' proposed an approach that explicitly integrates obstacle velocities into the position estimation process using a Kalman-based observer. This method leverages the measured velocities to predict obstacle positions through a tentacle-based strategy, enhancing the responsiveness and accuracy of navigation decisions. The system's implementation employs a combination of sensors, including cameras for visual navigation along predetermined paths during the training phase and a LIDAR sensor to detect and avoid both static and moving obstacles that were not encountered during training.

In their paper 'Detection of kidnapped robot problem in Monte Carlo localization based on the natural displacement of the robot' I. Bukhori et al. [4], conduct an experimental analysis comparing GMapping and RTAB-Map, focusing on their SLAM accuracy, the quality of the maps they produce, and how effectively these maps support navigation tasks. The study specifically targets ground robots equipped with RGB-D sensors operating in indoor environments, offering valuable insights into the practical performance and applicability of these SLAM techniques in real-world scenarios.

F. Dellaert et al [11]. Used algorithm for localization is Monte Carlo Localization (MCL), which is often favored over other methods due to its demonstrated accuracy and robustness in addressing the robot localization problem. For instance, MCL was implemented with a Laser Range Finder (LRF) and sonar sensors, outperforming Kalman filters thanks to its capability of representing multimodal distributions, which is essential for global localization. Additionally, it surpassed grid-based Markov localization algorithms by providing higher localization accuracy.

III. Methodology of kidnapped robot problem

The SLAM (Simultaneous Localization and Mapping) algorithm is integral to autonomous navigation, as it estimates a robot's position while it explores unknown environments. The accuracy and quality of the generated map improve with increased exploration of uncharted areas. In this context, the SLAM algorithm is implemented using the GMapping tool, which relies on odometry data from wheel encoders and laser data from sensors like the Kinect to construct a 2-D occupancy grid map resembling a building floor plan. Robot software development is carried out within the Robot Operating System (ROS2), which facilitates inter-process communication, low-level device control, and the implementation of common functions through a node-based graph architecture. Furthermore, complex 3D simulations are performed using GAZEBO, a simulation environment that includes components such as Models, GZ Server, GZ Client, and GZ Web. GAZEBO offers features such as built-in robot models, cloud simulation, dynamics simulation, advanced 3D graphics, command line tools, and TCP/IP transport, all of which support the visualization and analysis of inertia, forces, and sensor information.

IV. Simulation Environment

A. Robot Operating System (ROS2)

ROS 2 is an open-source framework designed for building advanced robotic applications. It extends the original ROS by incorporating improved real-time performance, enhanced security, and robust support for distributed systems and multi-robot coordination. Its modular architecture and communication backbone system released under the BSD (Berkeley Software Distribution) license, designed specifically for robotics. It provides a comprehensive set of tools and libraries that enable developers to write, build, and test code across multiple machines, supporting scalable, distributed systems. ROS 2 offers essential services including low-level device control, common functionalities, robust inter-process communication through DDS (Data Distribution Service), and effective package management.

B. Gazebo

Gazebo is a free, open-source 3D simulator for robotics that lets us build and test robots in a virtual world. It uses real-world physics—like gravity, light, and movement—so that the simulated robot behaves much like a real one. This means we can quickly try out new designs and algorithms without worrying about damaging expensive equipment such as robots.

In addition, Gazebo simulates sensor data from devices such as cameras, lasers, and more. This helps us to see how our robot's sensors perform in different conditions, even in extreme environments that might be too risky for actual hardware.

Robot designs are defined using an XML file called the Universal Robotic Description Format (URDF), which details the different parts of the robot and how they fit together. For example, one image might show a 3D model of a TurtleBot3 in Gazebo, while another shows the testing environment where the robot operates.

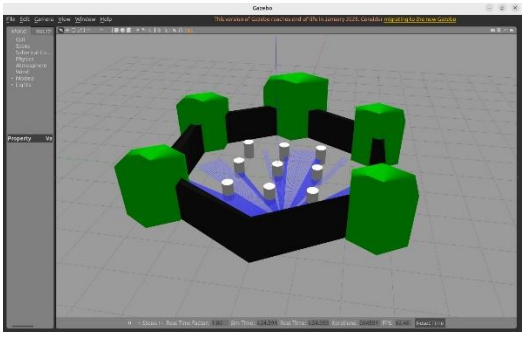

Fig. 2: Gazebo Simulation in ROS World

C. Rviz2

Rviz2, the advanced ROS2 visualization tool, is an essential resource for students and developers alike. It allows us to view our robot's 3D model and monitor sensor data in real time, just like its predecessor, but with enhanced performance and flexibility. With Rviz2, we can record sensor outputs—such as point clouds from stereo cameras, laser scanners, and depth sensors—and replay these logs to diagnose and fine-tune your robotic systems.

In addition to visualizing 3D data, Rviz2 supports 2D image feeds from webcams and RGB cameras, offering a comprehensive view of what the robot perceives. Its robust plugin system enables custom displays and interactive markers, so we can tailor the interface for our specific project needs. Moreover, Rviz2's improved multi-threaded rendering and dynamic configuration features help us analyze the robot's decision-making process, monitor sensor accuracy, and debug both planned and unplanned actions in real time.

This powerful tool not only bridges the gap between simulation and real-world testing but also deepens our understanding of robotic behaviors and sensor integration. Whether we're troubleshooting localization issues or optimizing navigation algorithms, Rviz2 is our go-to platform for visualizing and refining the inner workings of our robot.

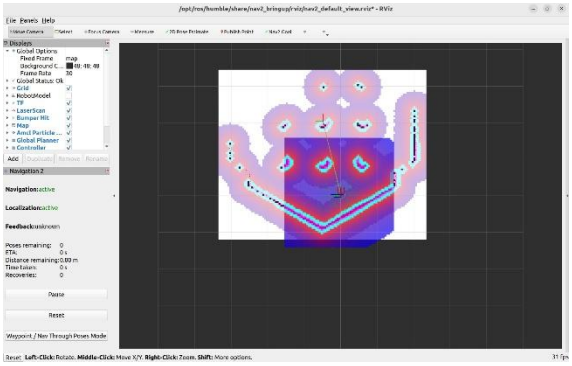

Fig. 3: Gmapping Simulation in rviz2

V. SLAM ALGORITHM

Autonomous robots are capable of navigating both indoors and outdoors while avoiding obstacles through a technique known as Simultaneous Localization and Mapping (SLAM). SLAM allows a robot to continuously create and update a map of its environment while determining its own location within that map. Various algorithms have been developed to address SLAM challenges, including Particle Filters, Extended Kalman Filters, FastSLAM, Covariance Intersection, and Graph-based SLAM, each integrating mapping, sensing, kinematic modeling, and the management of data from multiple sources and moving objects. A crucial component in SLAM is the range measurement device, which collects essential information about the surroundings. This data enables the robot to detect landmarks—distinctive features that help refine its position—using sensors and measuring devices. Upon sensing a landmark, the robot processes the input to better understand its environment and update its map. Among the several SLAM algorithms available, such as CoreSLAM, KartoSLAM, Lago SLAM, and HectorSLAM, the project described here utilizes the Gmapping algorithm, which relies on a Particle Filter. The Particle Filter computes the posterior distributions of the states in a Markov process, even when observations are noisy or incomplete, by employing approximate prediction and update techniques. This comprehensive integration of mapping, sensor fusion, and probabilistic estimation makes SLAM an effective solution for enabling autonomous robot navigation in dynamic and complex environments.

Samples from the distribution are represented as particles, with each particle assigned a weight that indicates the probability of it being selected. Gmapping is a highly efficient Rao-Blackwellized particle filter that generates grid maps from laser data. This package includes a ROS wrapper for OpenSlam's Gmapping and provides a laser-based SLAM solution as a ROS node known as slam_gmapping. By using slam_gmapping, you can create a 2-D occupancy grid map from the LIDAR sensor and the robot's pose data. [5]

When the robot moves, it obtains a new perspective of its surroundings. However, this movement is inherently accompanied by noise and errors, which increases the uncertainty in the robot's localization. To manage this uncertainty automatically, a mathematical representation of the robot's motion is employed—this is known as the motion model [6]. The motion model captures the variations and imperfections in the robot's movement, enabling the system to better predict its new state based on previous information and control inputs.

As the robot navigates its environment, it identifies key features, known as landmarks, that need to be added to the map. However, due to errors in its exteroceptive sensors, the positions of these landmarks are not precisely determined. Additionally, the uncertainty in the robot's own location compounds this problem, making it necessary to carefully combine both sources of uncertainty. To address this challenge, an automated solution employs a mathematical model known as the inverse observation model, which calculates the most likely positions of the landmarks based on the sensor data.

When the robot detects landmarks that have already been mapped, it uses these known features to refine both its own position and the positions of all landmarks in the environment. This observation process reduces the uncertainty associated with its localization and the landmarks themselves. To automate this correction, a mathematical model is employed that predicts the expected sensor measurements based on the estimated positions of both the robot and the landmarks. This model, known as the direct observation model, plays a critical role in aligning the robot's internal map with actual sensor data, thereby enhancing the overall accuracy of the navigation system.

By combining the motion model, the inverse observation model, and the direct observation model with an estimator engine, we can construct a fully automated SLAM solution. The estimator is crucial for propagating the uncertainties introduced at each stage—whether from the robot's movement, the sensing of new landmarks, or the re-observation of previously mapped features. This careful management of uncertainty ensures that both the robot's self-localization and the positioning of landmarks are continually refined, resulting in a more accurate and reliable mapping of the environment.

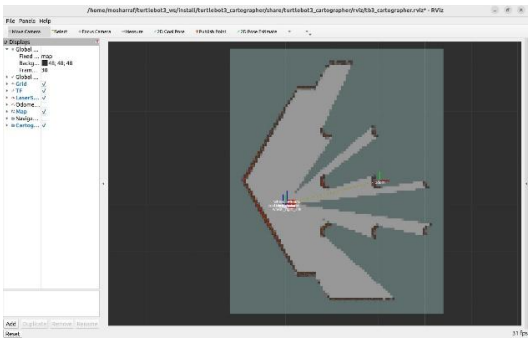

Fig. 4: SLAM simulation of the surrounding area

VI. Monto Carlo Localization

In mobile robot localization, it is virtually impossible for a robot to know its exact coordinates and orientation (collectively known as its pose) on a given map. Instead, the robot must infer this information from its environment. The state it infers is referred to as its belief, which is defined as

$$bel(St)=p(st|z1{:}t,u1{:}t) \quad \ldots\ldots\ldots 1$$

This posterior represents the probability distribution over the robot's state, $s_t= \langle x_t,y_t,\theta_t \rangle$, at time t. It is conditioned on all past measurements, $Z_{1:t}=\{z_1,z_2,\ldots\ldots\ldots z_t,\}$ and all past control $u_{1:t}=\{u_2,\ldots\ldots\ldots u_t,\}$. In some cases, it is also useful to consider the belief before incorporating the current measurement[7].

$$bel(st)=p(st|z1{:}t-1,u1{:}t) \quad \ldots\ldots 2$$

Monte Carlo Localization (MCL) is a Bayesian-based method for localization. It offers an effective framework to compute the posterior belief, denoted as bel(∗), using both measurement and control data. The underlying Bayes filter relies on the Markov assumption, which states that, given the current state s_t, past and future data are independent. By applying Bayes' rule along with this Markov property, the belief posterior can be defined as

$$be(st) = \eta p(zt|st)\ bel(st) \quad \ldots\ldots\ldots\ldots 3$$

The term $p(s_t|\ s_{t-1}u_{1:t})$ is known as the prediction or motion model, as it captures how the robot's state transitions due to its motion. Conversely, the probability $p(z_t|\ s_t)$ is referred to as the correction or sensor model, since it uses sensor readings to update the robot's state. The normalization constant η ensures that the resulting probability distribution sums to one. The Bayes filter framework allows for various representations of the posterior. In Monte Carlo Localization (MCL), the posterior $bel(st)$ is represented by a St of N weighted samples, with the sample density reflecting the likelihood of the robot occupying a particular pose [7].

$$St= \langle s_{t[n]},\omega_{t[n]} \rangle;\ n=1,2,\ldots,N \quad \ldots\ldots\ldots 4$$

Each particle $s_{t[n]}$ represents the hypothesis of the robot's pose at time t. The $\omega_{t[n]}$ is the nonnegative number called weight of particle. It indicates how good particle $s_{t[n]}$ in representing the robot's pose.

One major challenge with particle filters is maintaining a random distribution of particles throughout the state space, a task that becomes increasingly complex as the problem size grows. Consequently, an adaptive particle filter is often preferred because it converges faster and is significantly more computationally efficient than a basic particle filter. Monte Carlo Localization (MCL) is a probabilistic localization method used for robots navigating in 2D environments. It implements the Monte Carlo localization approach, leveraging a particle filter to track a robot's position relative to a pre-existing map.

The process begins with generating a map of the environment. The robot may either be placed at a random location or start with no initial position estimate. As the robot moves, new samples are generated to predict its updated position based on its control commands. If the robot loses track of its position, additional random, uniformly distributed samples can be introduced to help it recover its localization.

VII. Implementation & Result

A. Hardware

1. **TurtleBot3** Burger is a compact, cost-effective mobile robot built on ROS, tailored for education, research, hobby projects, and product prototyping. Its design focuses on reducing both size and cost without compromising on functionality or quality, while also allowing for future upgrades. The robot's modular construction means you can customize it by reconfiguring mechanical parts and integrating optional components such as computers and sensors. With core technologies like SLAM, Navigation, and Manipulation at its heart, TurtleBot3 is especially suited for home care applications—it can build maps and autonomously navigate your room using simultaneous localization and mapping algorithms.

Fig. 5: Turtlebot 3 Burger[6]

2. **Lidar**—short for laser distance sensor—is an optical device used to measure the distance to an object using a non-invasive laser beam. These devices can also function as proximity sensors, detecting objects within a specific range. There are several methods for distance measurement, including:

   - Time-of-Flight (ToF): This is the most common approach, where a narrow pulse of laser light—focused by a lens system—is emitted toward the object. The sensor then measures the time it takes

for the light to bounce back, calculating the distance based on this travel time.

- Triangulation and Similar Techniques: These methods use geometric principles to determine distance, often involving the angle of incidence of the reflected light.

Each method has its applications, but the Time-of-Flight principle is widely favored for its simplicity and effectiveness.

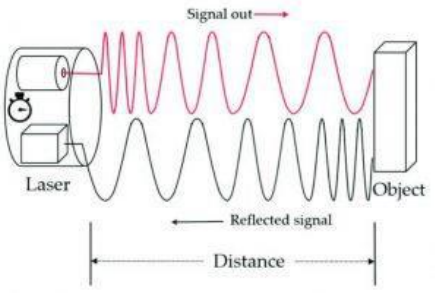

Fig.6: Principle of Lidar Sensor

When the laser beam reaches an object, it reflects off its surface and returns to the sensor, which then measures the duration of the round trip. The LDS-01 is a 2D laser scanner that offers a full 360-degree view, collecting data from the surrounding environment to support SLAM (Simultaneous Localization and Mapping). This sensor is integrated into the TurtleBot3 Burger, Waffle, and Waffle Pi models.

Fig.7: Lidar Sensor - LDS 01 [08]

3. **The Raspberry Pi** is a compact, low-cost computer about the size of a credit card, designed to connect to a monitor or TV and work with a standard keyboard and mouse. This small yet powerful device opens up opportunities for users of all ages to explore computing and learn programming languages such as Scratch and Python. It handles everyday desktop tasks—from internet browsing and high-definition video playback to creating spreadsheets, word processing, and gaming.

   The Raspberry Pi 3 Model B+ represents the latest addition to the Raspberry Pi 3 family. It features a 64-bit quad-core processor running at 1.4GHz, supports dual-band wireless LAN (2.4GHz and 5GHz), includes Bluetooth 4.2/BLE, offers faster Ethernet connectivity, and provides PoE capability with an optional PoE HAT[12].

Fig.8: Raspberry pi 3 b+ Model [09]

4. **OpenCR 1.0** (Open-source Control module for ROS2) is an open-source robot controller powered by a high-performance MCU from the ARM Cortex-M7 series. It supports RS-485 and TTL interfaces for controlling Dynamixel motors, along with UART, CAN, and various other communication protocols. Additionally, it is compatible with development tools such as the Arduino IDE. One significant advantage is its ability to operate more effectively when paired with a host controller like an SBC (Single Board Computer). OpenCR 1.0 also provides specialized ROS-based resources to fully leverage its capabilities within the ROS ecosystem[13].

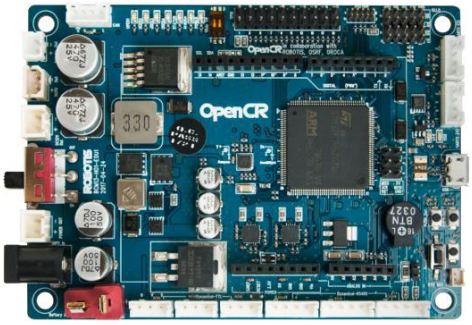

Fig.9: OpenCR 1.0 [10]

B. Software

Linux Machine - Unix/Linux Machine It is the machine required for all heavy computations. The robot sends data from its sensors to the computer. The information is fed into MCL by the computer. Simultaneously, a movement command is delivered from the computer to the robot.

The Linux Machine corresponds to the Remote PC, which will control TurtleBot3. Ubuntu 22.04.5 LTS (Jammy)is used for Simulating the turtle bot 3 burger (Kinetic)

ROS2 - The Robot Operating System (ROS2) is a message-passing service that makes robotics easier. Every system component (sensors and actuators) is run by individual nodes that handle the low-level software that runs the component. We can then send messages to run the SLAM, Teleoperation, Navigation, and Position Estimation nodes.

C. Project architecture

Colcon - Colcon is a command-line tool used for building sets of software packages in ROS 2. It expands upon existing build systems like CMake and is designed to facilitate the build, test, and installation processes across multiple packages with minimal user configuration.

Colcon Workspace - A Colcon workspace is the directory that organizes all the packages and is compiled with the Colcon tool. The src folder stores the packages' source code. During compilation, the build folder contains intermediate build files, while the install folder contains the final installed artifacts, including headers, libraries, and executables. The log folder holds build logs and other output related to the build process. Typically, all development is done within the src folder.

Package - A ROS 2 package built with Colcon can house one or more executables (nodes). Each package must include a package.xml file, which defines the package's attributes and dependencies. If a package uses CMake, it also has a CMakeLists.txt file outlining the build configuration. Additional resources, such as launch files, reside in the launch folder, while YAML configuration files are found in the config folder. The launch file enables multiple nodes to be started at once, supporting more complex, multi-node applications.

D. Communication framework

Master – the node manager - Every node registers with the master before starting. The master oversees communication between nodes. We must run 'ros2core' each time we start ROS2 because it launches the master. (REMOTE PC)

Node – process in ROS2 - A node is a live instance of an executable file from a package. Use 'ros2run [pkg_name] [node_name]' to start a node. Use 'ros2node list' to display all currently running nodes. Use 'ros2launch [pkg_name] [file_name.launch]' to launch the master and multiple nodes simultaneously. The launch file defines the rules for starting nodes.

E. Communication methods

ROS2 offers a variety of communication methods including Topics, Services, Parameters, and Actions. Topics utilize a publish-subscribe model, enabling asynchronous communication between nodes via channels with message types defined in '.msg' files. Services employ a synchronous request-reply mechanism, which is particularly useful for tasks invoked on demand, with service data types stored in '.srv' files. In ROS2, parameters are managed locally within each node rather than through a centralized parameter server, allowing individual nodes to maintain their own configuration dictionaries. Additionally, Actions provide a server-like communication method with status feedback, making them ideal for time-consuming and preemptive tasks, and the action data types are defined in '.action' files.

F. Map Generation

The map generation process begins by creating the environment and importing the TurtleBot3 into Gazebo, where the robot model, featuring two wheels and a mounted Lidar sensor, is set up. The mapping is executed using the Gmapping package: the sensor captures depth images of the environment via infrared sensors acting as rangefinders, which are then converted into a 3D point cloud using the depth image proc package in ROS2, and subsequently transformed into a 2D laserscan with the point cloud to laserscan package. By providing all required parameters to the Gmapping packages, a map is generated in RViz (as shown in Figure 10). Initially, the robot is navigated in the simulated environment using the teleop_twist_keyboard key package in ROS2 by sending velocity commands via the keyboard, and the rqt graph of Gmapping is depicted in Figure 11. Additionally, the Gmapping package provides the /map topic, which the map server package uses to save the generated map, making it ready for autonomous navigation.

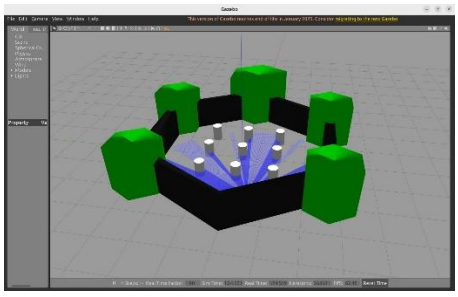

Fig.10: Turtlebot3 Gazebo World Simulation

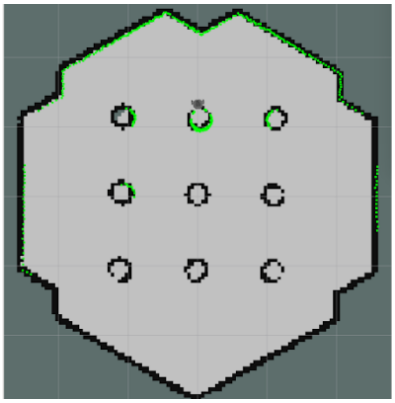

Fig.11: Turtlebot3 rqt graph of Gmapping of turtlebot3 world

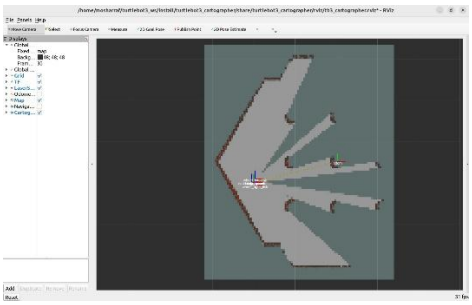

Fig.12: Turtlebot3 rqt graph of Gmapping of turtlebot3 real world

G. Localization

Localization estimates the position and pose of the robot relative to its environment. For accurate localization, laser data, odometry, and an environmental map are essential. The TurtleBot3 employs Monte Carlo Localization (MCL), a probabilistic approach for 2D robots, where the robot's pose is tracked against a known map using a particle filter. To reduce computational requirements, KLD (Kullback–Leibler divergence) sampling is utilized to automatically adjust the sample size, as illustrated in Fig. 2, which shows the map visualized in RViz. Fig. 3 displays the self-localized robot in RViz.

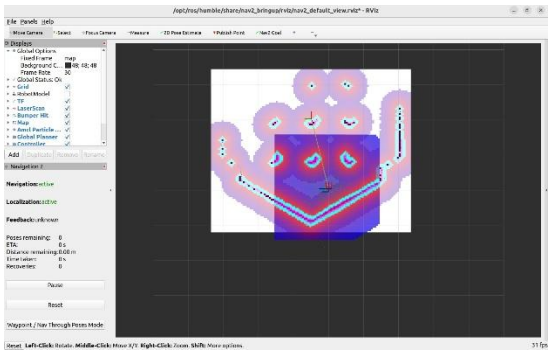

Fig.13: Autonomous Visualization in rviz

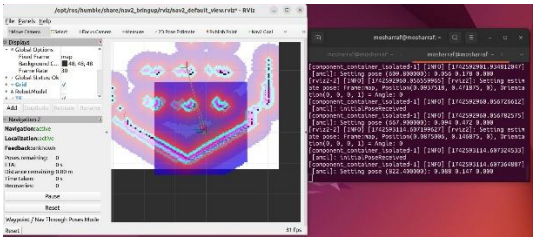

Fig.14: Turtlebot3 Visualization and teleop mode

H. Autonomous navigation

Autonomous navigation is achieved once mapping and localization are complete. The ROS2 navigation stack packages implement AMCL, which provides the node for localizing the robot on a static map. This node subscribes to TF data, the robot's 2D laser scan data, and the previously generated static map, subsequently publishing the robot's pose and estimated position relative to the map. RViz then visualizes both the global and local costmaps, enabling the robot to move autonomously. A 2D navigation goal can be set in RViz, assigning a destination and orientation to the robot on the map. The robot then plans its path to the desired destination and sends corresponding velocity commands to its controller, as illustrated in Figure 13 & 14, which shows the path from the initial position to the assigned goal where in fig 14 the data also changing that means it changes its location.

VIII. CONCLUSION

In conclusion, the Kidnapped Robot Problem poses a significant challenge for autonomous robots when their initial pose is unknown or when they are unexpectedly relocated. Various techniques, including particle filters and Kalman filters, have been developed to estimate the robot's pose with high accuracy.

The issues of the Kidnapped Robot Problem, localization, and SLAM remain critical topics in robotics that demand ongoing research and development. By leveraging advanced algorithms and platforms like TurtleBot3, we can enhance the capabilities of autonomous robots, enabling them to operate more efficiently and effectively in real-world scenarios.

We conducted an experimental analysis of an autonomous navigation system using ROS2, employing the TurtleBot3 Burger and a Linux machine (Ubuntu 22.04) to simulate, test, and validate our project.

Utilizing the Robot Operating System (ROS2), we simulated localization to help the robot determine its position and orientation in both the Gazebo simulation and the real-world environment. SLAM was used to create maps of the surroundings as the robot moved through unknown areas, and we validated autonomous navigation and position estimation with the generated maps in Gazebo and RViz.

Overall, our findings demonstrate the effectiveness of the Robot Operating System as a platform for experimentation with various algorithms—such as SLAM, Localization, and MCL—that significantly improve position estimation and support autonomous navigation.

IX. ACKNOWLEDGEMENTS

We would like to acknowledge Frankfurt University of Applied Sciences for providing the necessary resources such as Robot for this project. Special thanks to Prof. Dr. Peter Nauth for his guidance throughout the Autonomous Intelligent Systems module.

We also appreciate the contributions of the ROS2 and TurtleBot3 communities, whose resources supported the development of our obstacle avoidance system.

REFERENCES

[1] "A. Cherubini, F. Spindler, and François Chaumette, "Autonomous Visual Navigation and Laser-Based Moving Obstacle Avoidance," IEEE Transactions on Intelligent Transportation Systems, vol. 15, no. 5, pp. 2101–2110, Apr. 2014, doi: https://doi.org/10.1109/tits.2014.2308977.

[2] B. M. F. da Silva, R. S. Xavier, and L. M. G. Gonçalves, "Mapping and Navigation for Indoor Robots under ROS: An Experimental Analysis," Jul. 2019, doi: https://doi.org/10.20944/preprints201907.0035.v1.

[3] Andras Majdik, M. Popa, L. Tamas, and Gheorghe Lazea, "New approach in solving the kidnapped robot problem," ResearchGate, pp. 1–6, Jul. 2010.

[4] I. Bukhori and Z. H. Ismail, "Detection of kidnapped robot problem in Monte Carlo localization based on the natural displacement of the robot," International Journal of Advanced Robotic Systems, vol. 14, no. 4, p. 172988141771746, Jul. 2017, doi: https://doi.org/10.1177/1729881417717469.

[5] J. Solà, "Simulataneous localization and mapping with the extended Kalman filter 'A very quick guide... with Matlab code!,'" 2014. Available: https://www.iri.upc.edu/people/jsola/JoanSola/objectes/curs_SLAM/SLAM2D/SLAM%20course.pdf

[6] Y. Name, "ROBOTIS e-Manual," ROBOTIS e-Manual. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/ (accessed Mar. 21, 2025).

[7] I. Bukhori, Z. H. Ismail and T. Namerikawa, "Detection strategy for kidnapped robot problem in landmark-based map Monte Carlo Localization," *2015 IEEE International Symposium on Robotics and Intelligent Sensors (IRIS)*, Langkawi, Malaysia, 2015, pp. 75-80, doi: 10.1109/IRIS.2015.7451590.

[8] Media-amazon.com, 2025. https://m.media-amazon.com/images/I/51vjGC3M8BL.jpg (accessed Mar. 21, 2025).

[9] R. P. Ltd, "Buy a Raspberry Pi 3 Model A+," Raspberry Pi. https://www.raspberrypi.com/products/raspberry-pi-3-model-a-plus/ (Last accessed 21.03.25).

[10] "ROBOTIS e-Manual," ROBOTIS e-Manual. https://emanual.robotis.com/docs/en/parts/controller/opencr10/ (Last accessed 21.03.25).

[11] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte carlo localization for mobile robots," in Robotics and Automation (ICRA), 1999 IEEE International Conference on, vol. 2, 1999, pp. 1322–1328.