‘/

(Established under Karnataka Act No. 16 of 2013)
100-ft Ring Road, Bengaluru – 560 085, Karnataka, India

Internship Report on

Simple Mapper

An Autonomous Mapping Bot

Submitted by:

Manasa Raval (PES1UG21EE074)

Ayush Kumar (PES2UG22CS117)

Kunal Kishore (PES1UG22EC134)

Shriharshith Keshav (PES1UG22EC280)

Vishwajit Okade (PES1UG23EC354)

S. Sunaina (PES1UG23AM249)

June – July 2024

Under the Guidance of

Dr. M J Venkatrangan

Faculty mentor, CRAIS

Professor,

Department of Electrical and Electronics,

PES University

Bengaluru-85

Center for Robotics, Automation and Intelligent Systems (cRAIS)
Room No. B-1212, 100 Feet Ring Road, BSK III Stage, Bangalore - 560085

ABSTRACT

This report presents the development and implementation of an autonomous mapping robot named SimpleMapper. The project incorporates advanced robotic technologies, including Fusion 360 for robot design, Webots for simulation, and sophisticated algorithms for 2D navigation and object detection. The robot aims to make mapping technology accessible to non-technical users while ensuring precise and dynamic obstacle avoidance. The report details the design, simulation, and testing processes, concluding with the successful implementation of autonomous mapping capabilities

CONTENTS

1. Introduction

The advent of autonomous robots has revolutionized various industries, including mapping, surveillance, and exploration. SimpleMapper is a robot designed to bridge the gap between complex mapping technologies and users with limited technical expertise. This report outlines the comprehensive development of SimpleMapper, focusing on key aspects such as robot design, simulation, and autonomous navigation. By leveraging ROS, Webots, and object detection algorithms, the project aims to deliver a user-friendly and efficient mapping solution.

2. Literature review

1) M. B. Alatise and G. P. Hancke, "A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods," in IEEE Access, vol. 8, pp. 39830-39846, 2020, doi: 10.1109/ACCESS.2020.2975643.

Sensor fusion is essential for improving the accuracy and reliability of AMRs, particularly in localization and navigation tasks. Integrating multiple sensors helps overcome individual sensor limitations, leading to better performance in complex environments. Implementing sensor fusion methods is complex and requires significant computational resources and sophisticated algorithms. AMRs are used in a wide range of applications, including surveillance, education, agriculture, firefighting, and search and rescue.

2) Häussermann, K., Zweigle, O. & Levi, P. A Novel Framework for Anomaly Detection of Robot Behaviors. J Intell Robot Syst 77, 361–375 (2015). https://doi.org/10.1007/s10846-013-0014-5.

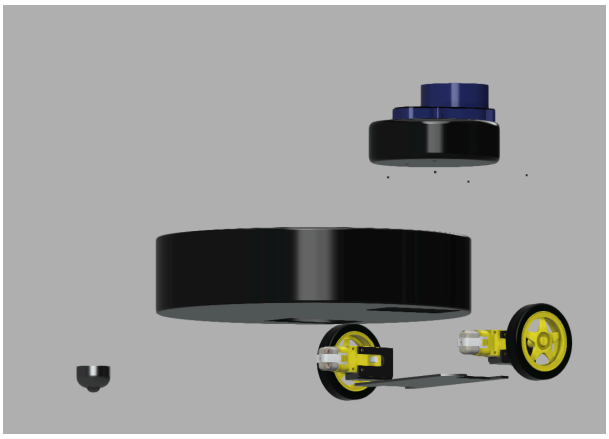
The framework integrates SOMs for spatial modeling and PGMs for temporal modeling, leveraging the strengths of both methodologies to enhance anomaly detection in robotic behaviors. The SOMs are trained in an unsupervised manner, requiring no expert intervention, which is beneficial for handling high-dimensional input spaces and adapting to various robotic applications. Its design allows for a high level of generalization, making it applicable to various types of robots and sensors, and capable of distinguishing between spatial and temporal anomalies.

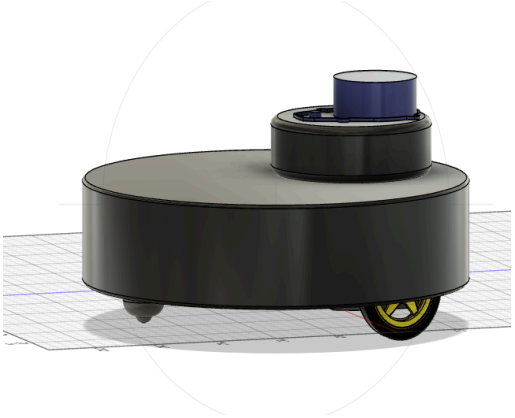
3. Modeling in Fusion360

The design process of the SimpleMapper robot, as visualized in Autodesk Fusion 360, was critical in ensuring that the final product would meet the project's requirements for autonomy and precision in mapping. The model provided in the presentation clearly illustrates the key design elements and the rationale behind them.

A. Design Concept

The chassis contains a circular base design. This shape is essential for enabling smooth, omnidirectional movement, which is particularly beneficial in mapping tight or complex environments. The model shows the differential drive system, which uses two primary wheels for movement and a caster wheel for balance. This setup allows SimpleMapper to rotate on its axis, making it highly maneuverable in confined spaces.



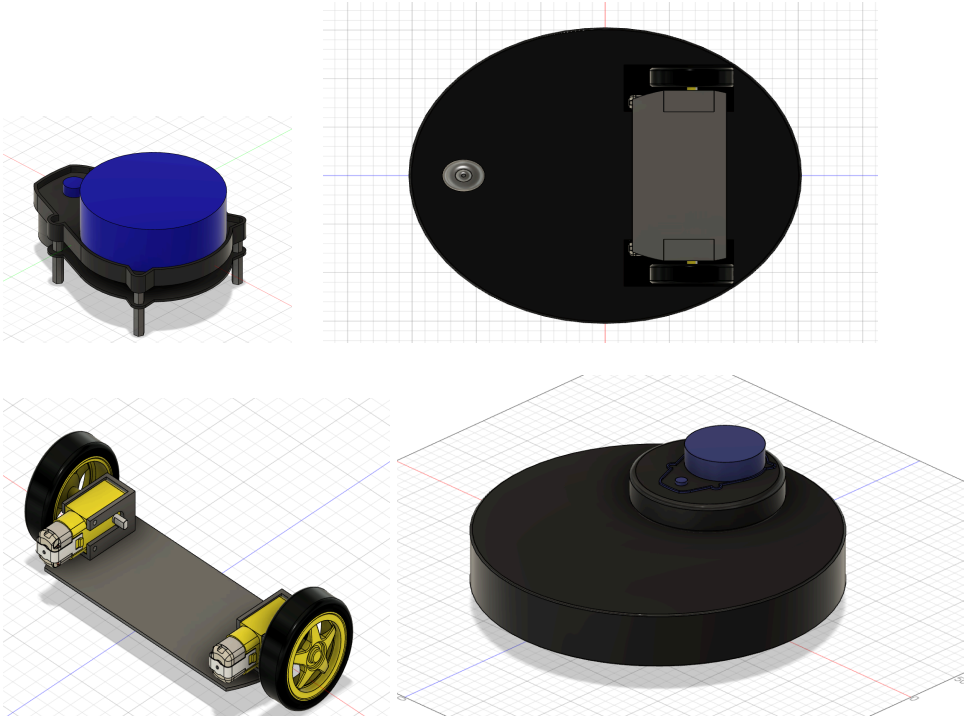


B. Structural Design and Layout

The **Robot Assembly** showcases the assembled view of the robot, emphasizing the elevated platform that houses the sensors. The cylindrical sensor mount visible in the Fusion 360 model is designed to hold the LIDAR sensor securely. The elevated position of the LIDAR ensures an unobstructed 360-degree view, which is crucial for accurate mapping.

C. Component Integration and Accessibility

The robot's design prioritizes accessibility, which is evident in the layout of the components within the chassis. The design demonstrates how each component fits within the overall structure and the bonet type opening which allows for straightforward assembly and disassembly. This design choice is particularly useful for troubleshooting and upgrades, as it minimizes the time required to access and replace parts.



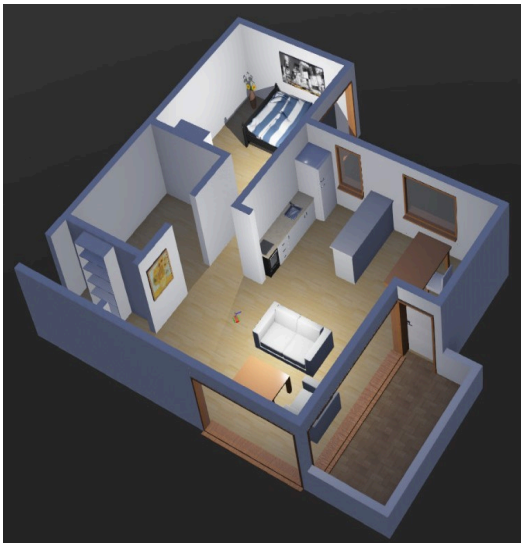
D. Final Adjustments and Design Optimization

Final adjustments were made to optimize the robot's performance. These included refining the sensor placement for optimal coverage, adjusting the battery slots for better balance, and enhancing the overall design to reduce weight while maintaining strength. The result was a robust and efficient design, ready for the next phase of development, which involved physical prototyping and testing in real-world environments.

4. Webots World

The Webots world used for simulation is a simple home-like setup with very basic everyday obstacles that the robot would be dealing with, in a real life scenario.

The simulation environment also tests the robot's capabilities to navigate to get under a certain few obstacles(beds,sofas,etc.) to map the area with a higher precision.

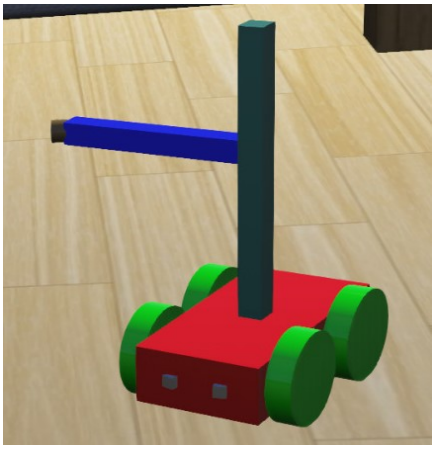


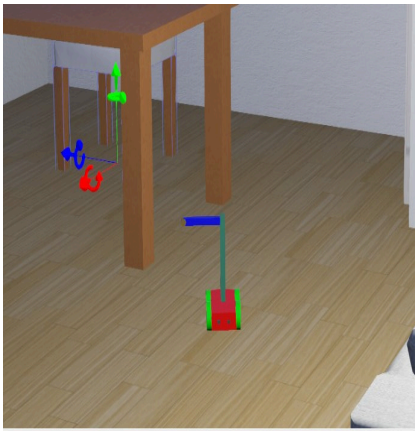
5. Webots Robot

The Webots robot is a 4-wheeled-robot equipped with a lidar sensor which is the main component used by the robot to build an accurate map of the world while also being used to maneuver the robot through tight spaces and obstacle detection and avoidance. Two ultrasonic distance sensors on the front side each covering an angle of 60 degrees,collectively giving a 120 degree field of view for detection of transparent obstacles (glass walls and glass doors). The robot also is equipped with a IMU to help with the localization of the robot in the world.The driving motors for the wheels have wheel encoders which help us in localizing the robot in the world with more precision and accuracy. The webots robot also consists of a camera mounted on to a linear and rotational actuator (future development of the project)

```
void SensorEnable::Initialize_sensors(){
    std::vector<std::string> sensors{"/CAM" , "/Lidar" ,"/ds_left" , "/ds_right" , "/global" , "/IMU" , "/Linear_sensor" , "/Rotation_sensor"};
    std::vector<ros::ServiceClient> vec_client;
    for (auto sensor = sensors.begin(); sensor != sensors.end(); ++sensor){
        vec_client.push_back(nh_.serviceClient<webots_ros::set_int>(SensorEnable::robot_name_+ *sensor +"/enable"));
        ros::service::waitForService(SensorEnable::robot_name_ + *sensor + "/enable");
        vec_client.back().call(srv_timestep);}
    //subscribe_keyboard_ = nh_.subscribe(SensorEnable::robot_name_+"/keyboard/key", 1, &SensorEnable::KeyboardCallBack,this);
    laser_scan_subs=nh_.subscribe(SensorEnable::robot_name_+"/Lidar/laser_scan/layer0",1,&SensorEnable::scannerCallback,this);
    std::vector<std::string> actuators{"/wheel1" , "/wheel2" ,"/wheel3" , "/wheel4" , "/linear" , "/RM"};

    for (auto actuator = actuators.begin(); actuator != actuators.end(); ++actuator){
        vec_client.push_back(nh_.serviceClient<webots_ros::set_float>(SensorEnable::robot_name_+ *actuator +"/set_position"));
        ros::service::waitForService(SensorEnable::robot_name_ + *actuator + "/set_position");
        if(!vec_client.back().call(srv_inf)) ROS_WARN("Position not set");

        vec_velocity_.push_back(nh_.serviceClient<webots_ros::set_float>(SensorEnable::robot_name_+ *actuator +"/set_velocity"));
        ros::service::waitForService(SensorEnable::robot_name_ + *actuator + "/set_velocity");
        if(!vec_velocity_.back().call(srv_zero)) ROS_WARN("velocity not set");
    }
}
```

Sensor initialization of the robot





6. Webots World Mapping using Teleop

In the initial stages of the project we used the Teleop operation to move the robot around the simulation world to generate a map which is being built on the RViz platform.To perform the teleop operation , the ROS package teleop_twist_keyboard was used which enabled us to control the robots movements using our keyboards.This gave us an insight on the various problems faced by the bot while mapping like , glass panels not being considered as an obstacle on processing the LIDAR data, accurate localization not being done on the map due to the unavailability of wheel encoders ,which were later on rectified giving us better results.

The robot also used the slam_gmapping package ,which is a ROS package used to localize the robot to its environment by using the gathered sensor data in real time and producing a map of the environment the robot is situated in, the slam_gmapping package also localized the

robot into the map produced on RViz. This stage helped us massively in eliminating many glitches in the backend code and integration of ROS Noetic with Rviz,Webots and the ROS packages to work simultaneously with coordination. The backend code was programmed using C++ for its ease of integration with Webots and ROS. The Teleop function used the cmd_vel C++ package to publish the velocity commands to the robot along with the geometry_msgs for the poses and angular orientation of the robot with respect to the base plane.

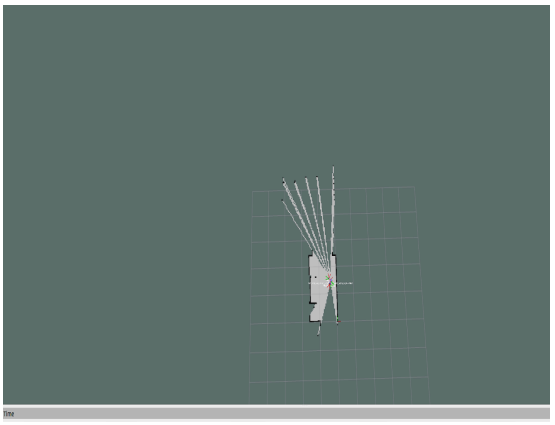


Fig. 6(a) Initial Stage

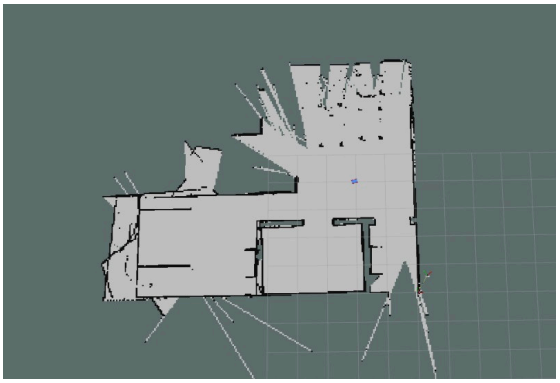


Fig. 6(b) Intermediate Stage

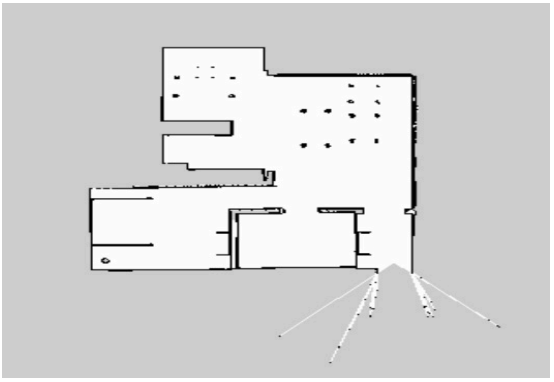


Fig. 6(c) Final Mapped Stage

```
Welcome  dsa.c  void SensorEnable::teleop(int key){ Untitled-1
1   void SensorEnable::teleop(int key){
2       // UP 315
3       // DOWN 317
4       // LEFT 314
5       // RIGHT 316
6       // W 87
7       // A 65
8       // S 83
9       // D 68
10
11      switch(key) {
12        case 87 :
13          srv_act.request.value = 0.1;
14          vec_velocity_[4].call(srv_act);
15          srv_act.request.value = 0;
16          vec_velocity_[5].call(srv_act);
17          break;
18
19        case 83 :
20          srv_act.request.value = -0.1;
21          vec_velocity_[4].call(srv_act);
22          srv_act.request.value = 0;
23          vec_velocity_[5].call(srv_act);
24          break;
25
26        case 65 :
27          srv_act.request.value = -0.4;
28          vec_velocity_[5].call(srv_act);
29          srv_act.request.value = 0;
30          vec_velocity_[4].call(srv_act);
31          break;
32
33        case 68 :
34          srv_act.request.value = 0.4;
35          vec_velocity_[5].call(srv_act);
36          srv_act.request.value = 0;
37          vec_velocity_[4].call(srv_act);
38          break;
39
40        case 315 :
41          srv_act.request.value = 5.0;
42          vec_velocity_[0].call(srv_act);
43          vec_velocity_[1].call(srv_act);
44          vec_velocity_[2].call(srv_act);
45          vec_velocity_[3].call(srv_act);
```

```
Welcome  dsa.c  void SensorEnable::teleop(int key){ Untitled-1
1   void SensorEnable::teleop(int key){
11      switch(key) {
52        case 317 :
60          vec_velocity_[4].call(srv_act);
61          vec_velocity_[5].call(srv_act);
62          break;
63
64        case 316 :
65          srv_act.request.value = 1;
66          vec_velocity_[0].call(srv_act);
67          vec_velocity_[2].call(srv_act);
68          srv_act.request.value = -1;
69          vec_velocity_[1].call(srv_act);
70          vec_velocity_[3].call(srv_act);
71
72          srv_act.request.value = 0;
73          vec_velocity_[4].call(srv_act);
74          vec_velocity_[5].call(srv_act);
75          break;
76
77        case 314 :
78          srv_act.request.value = -1;
79          vec_velocity_[0].call(srv_act);
80          vec_velocity_[2].call(srv_act);
81          srv_act.request.value = 1;
82          vec_velocity_[1].call(srv_act);
83          vec_velocity_[3].call(srv_act);
84
85          srv_act.request.value = 0;
86          vec_velocity_[4].call(srv_act);
87          vec_velocity_[5].call(srv_act);
88          break;
89
90        default :
91          srv_act.request.value = 0;
92          vec_velocity_[0].call(srv_act);
93          vec_velocity_[1].call(srv_act);
94          vec_velocity_[2].call(srv_act);
95          vec_velocity_[3].call(srv_act);
96          vec_velocity_[4].call(srv_act);
97          vec_velocity_[5].call(srv_act);
98      }
99  }
```

7. 2D Nav Goal & Dynamic Object Avoidance

The robot in this project is capable of autonomous navigation within a pre-mapped environment. The navigation process involves moving the robot from a starting point (Point A) to a destination (Point B) by integrating two distinct path-planning strategies: **Global Path Planning** and **Local Path Planning**.

Overview of the MoveBase Package

The **MoveBase** package in ROS serves as a comprehensive framework that facilitates the robot's navigation by allowing the integration of different global and local planners, as well as various approaches to cost mapping. This flexibility ensures that the navigation process can be tailored to suit the specific requirements and constraints of the robot and its environment.

MoveBase relies on a pre-existing map, typically provided by the `map_server`, to generate a global plan. The process is as follows:

1. **Global Path Planning**:
    - **Map Input**: The `map_server` provides a static map of the environment.
    - **Global Costmap**: The map is used to create a **global_costmap**, which represents the environment and the static obstacles within it.
    - **Global Planner**: The **global_planner** uses the global_costmap to compute an optimal path from the robot's current position to the target destination. This path is referred to as the **Global Path**.
2. **Local Path Planning**:
    - **Local Costmap**: A **local_costmap** is generated, which focuses on the robot's immediate surroundings and includes dynamic obstacles.
    - **Sensor Data**: The robot's sensors continuously provide real-time data about nearby obstacles.
    - **Local Planner**: Using the local_costmap and sensor data, the **local_planner** makes real-time adjustments to the Global Path, creating a **Local Path** that ensures collision avoidance and smooth navigation.

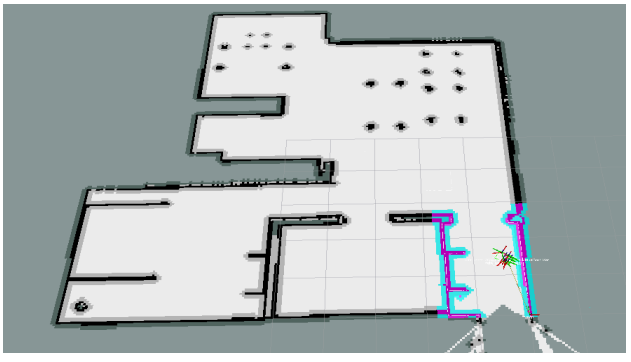
Execution and Control

Once the Local Path is established, the **MoveBase** package outputs a **command velocity**. This command is a set of velocity parameters that the robot follows to move along the computed path. The execution of these commands triggers the robot to navigate from its current location to the target goal while dynamically avoiding obstacles.

Visualization and Interaction with RViz

To interact with the robot and set navigation goals, the **2D Nav Goal** tool in RViz can be used. This tool allows the user to specify the target location on the map, which then prompts the MoveBase package to compute and execute the navigation strategy.

This process of merging the Global and Local Paths enables the robot to navigate effectively within its environment, ensuring that it reaches its destination while adapting to any obstacles that may appear in its path. This capability is crucial for real-world applications where environments are dynamic and unpredictable.



(a)

(b)

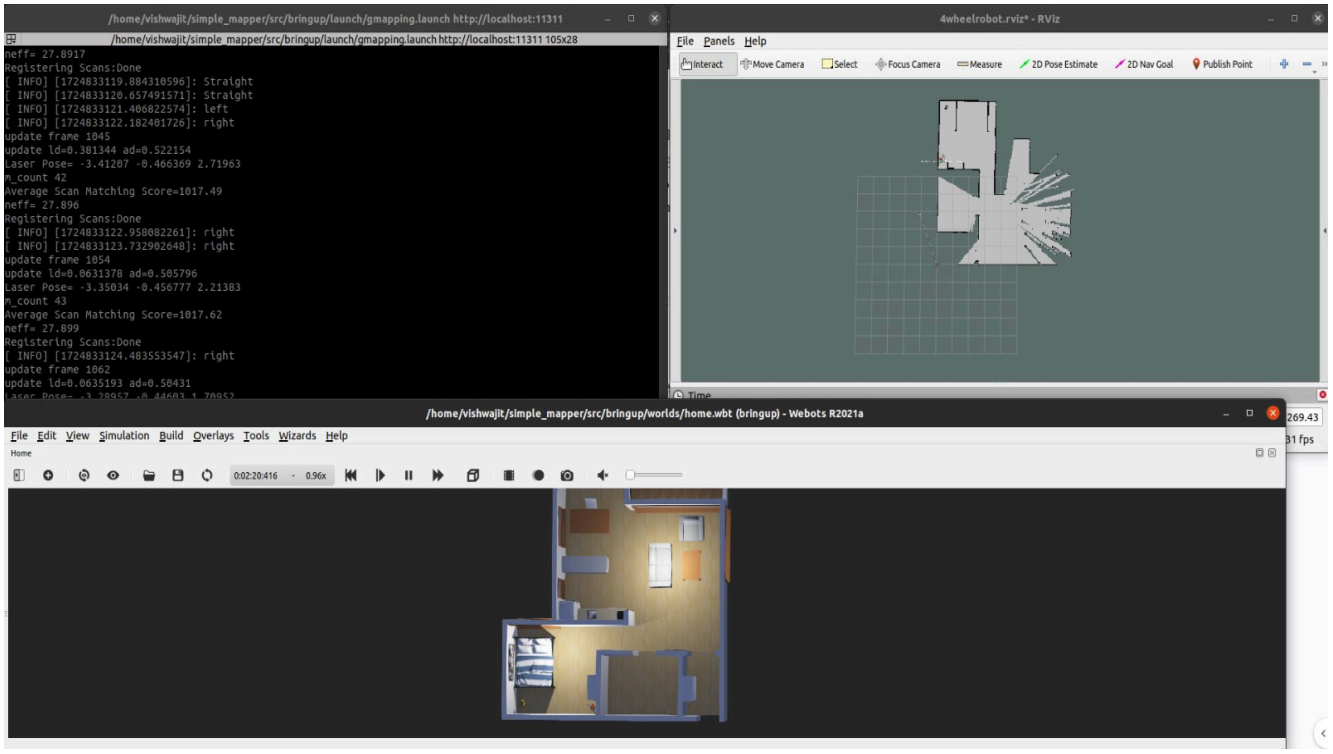
(c)

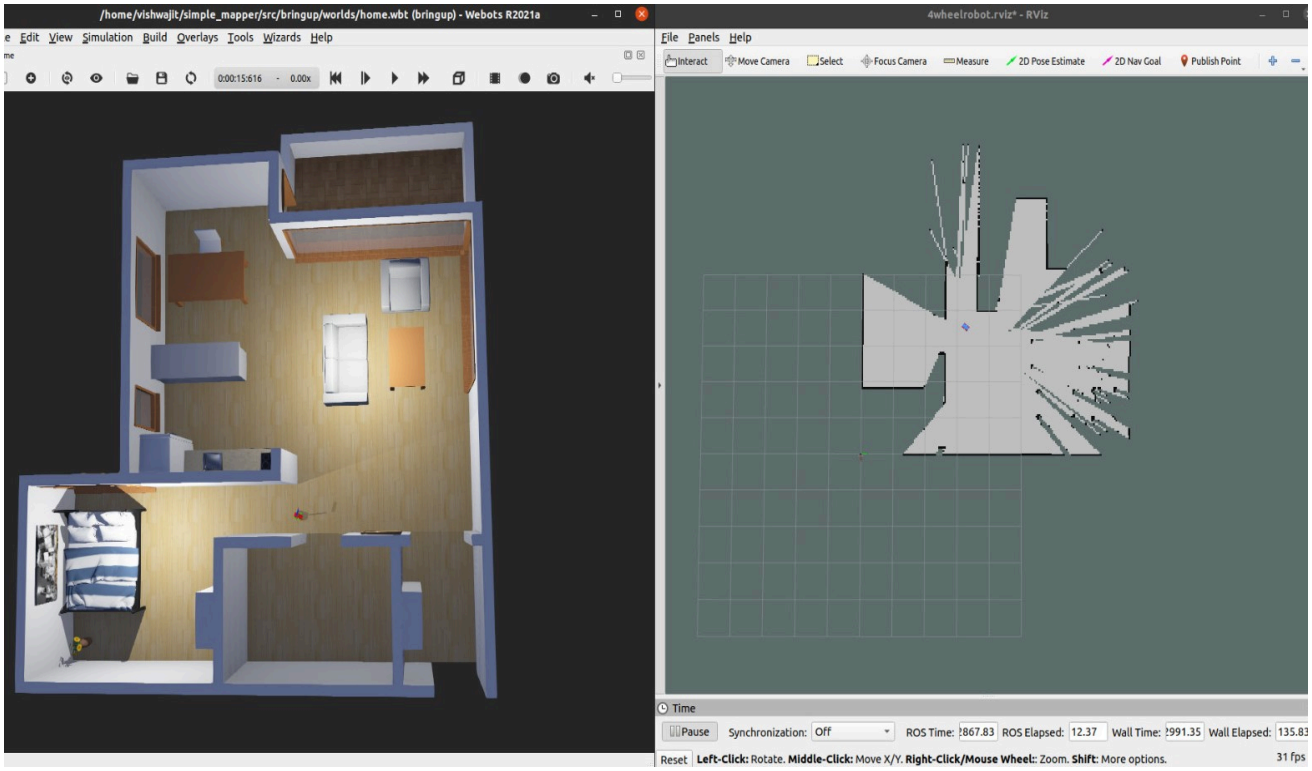
Fig. (a),(b) & (c) 2D Nav Goal and Dynamic Object Avoidance

8. Autonomous Mapping using Object Detection and Avoidance

SLAM (Simultaneous Localization and Mapping):

In this stage, the robot autonomously constructs a detailed map of its environment by leveraging sensors such as Lidar and odometry. These sensors enable the robot to capture intricate details of its surroundings, from static objects to the layout of the space. As the robot explores its environment, it builds a comprehensive and accurate map, which is crucial for subsequent navigation tasks. This map allows the robot to recognize previously encountered areas, thereby improving the efficiency and safety of its navigation.





sensor_enable.cpp

src > bringup > src > sensor_enable.cpp > ...

```
}*/

void SensorEnable :: scannerCallback (const sensor_msgs::LaserScan::ConstPtr & laser_scan_msgs)
{
  flag=0;
  //ROS_INFO("%ld",laser_scan_msgs->ranges.size());
  for ( i=laser_scan_msgs->ranges.size()*0.92;i< (laser_scan_msgs->ranges.size()*1.0);i++){
    if (laser_scan_msgs->ranges[i]<0.5){
      flag=1;

    }
  }
  for ( i= laser_scan_msgs->ranges.size()*0;i<(laser_scan_msgs->ranges.size()*0.08);i++){
    if ((laser_scan_msgs->ranges[i]<0.5)&&(flag!=1)){
      flag=2;

    }
  }


if (flag==1){
  //cmd_vel_command.linear.x=0.1;
  //cmd_vel_command.angular.z=-10;
  srv_act.request.value=1;
  vec_velocity_[0].call(srv_act);
  vec_velocity_[2].call(srv_act);
  srv_act.request.value=-1;
  vec_velocity_[1].call(srv_act);
  vec_velocity_[3].call(srv_act);

  srv_act.request.value=0;
  vec_velocity_[4].call(srv_act);
  vec_velocity_[5].call(srv_act);

  ROS_INFO("right");

}
else if (flag ==2){
  //cmd_vel_command.linear.x=0.1;
  //cmd_vel_command.angular.z=10;
  srv_act.request.value=1;
  vec_velocity_[1].call(srv_act);
  vec_velocity_[3].call(srv_act);
  srv_act.request.value=-1;
```

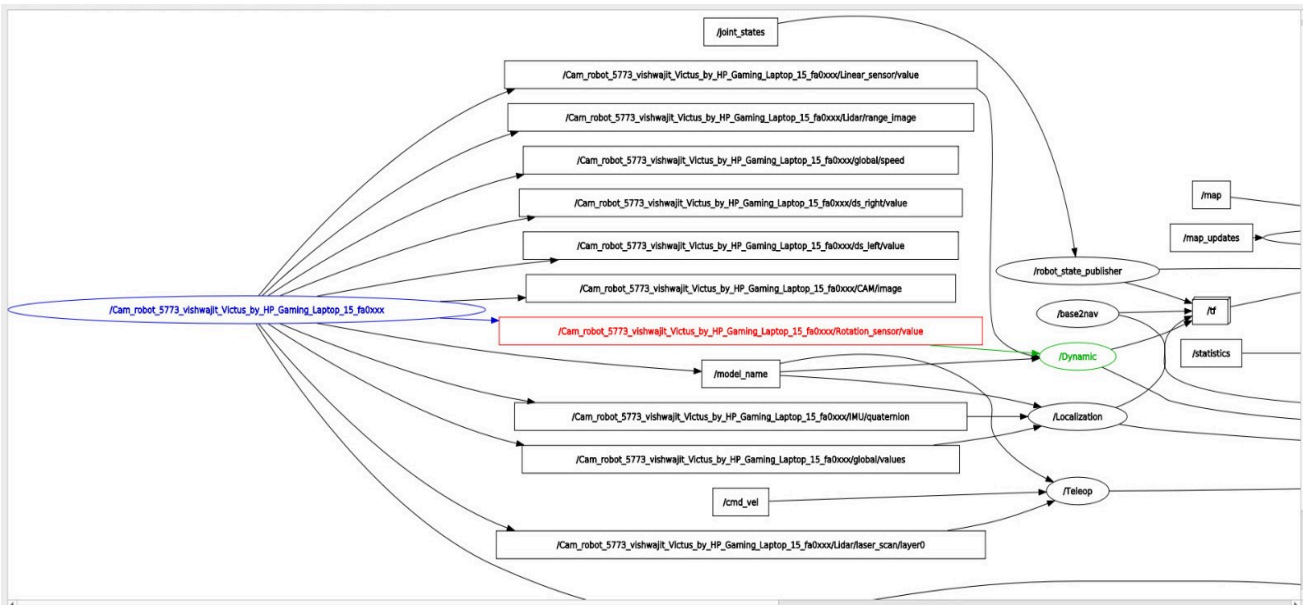
-- NORMAL --

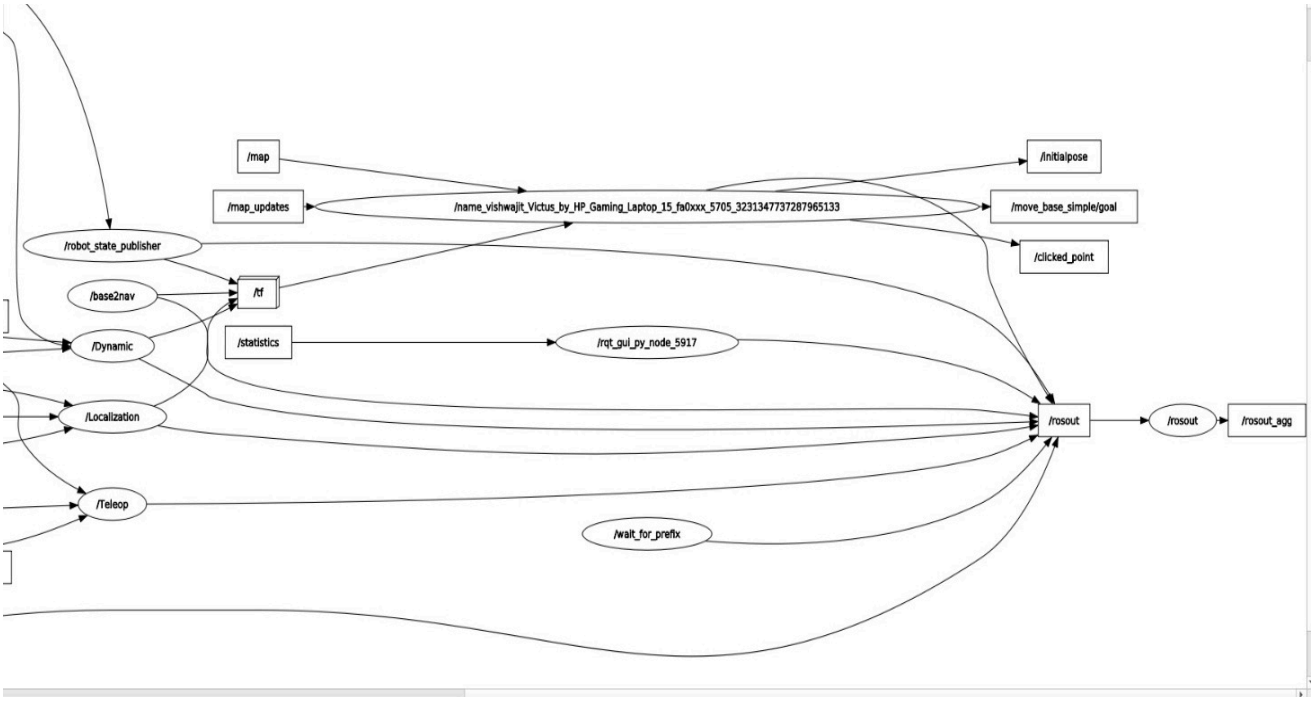
sensor_enable.cpp

src > bringup > src > sensor_enable.cpp > ...

```
void SensorEnable :: scannerCallback (const sensor_msgs::LaserScan::ConstPtr & laser_scan_msgs)
  srv_act.request.value=-1;
  vec_velocity_[0].call(srv_act);
  vec_velocity_[2].call(srv_act);

  srv_act.request.value=0;
  vec_velocity_[4].call(srv_act);
  vec_velocity_[5].call(srv_act);
  ROS_INFO("left");
}
else if (flag==0)
{
  //cmd_vel_command.linear.x=0.5;
  //cmd_vel_command.angular.z=0;
  srv_act.request.value=5.0;
  vec_velocity_[0].call(srv_act);
  vec_velocity_[1].call(srv_act);
  vec_velocity_[2].call(srv_act);
  vec_velocity_[3].call(srv_act);

  srv_act.request.value=0;
  vec_velocity_[4].call(srv_act);
  vec_velocity_[5].call(srv_act);

  ROS_INFO("Straight");
}
  //controlled_cmd_vel.publish(cmd_vel_command);
}
```

RQT Graph


/joint_states
/Cam_robot_5773_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx/Linear_sensor/value
/Cam_robot_5773_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx/Lidar/range_image
/Cam_robot_5773_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx/global/speed
/Cam_robot_5773_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx/ds_right/value
/Cam_robot_5773_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx/ds_left/value
/Cam_robot_5773_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx/CAM/image
/Cam_robot_5773_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx
/Cam_robot_5773_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx/Rotation_sensor/value
/model_name
/Cam_robot_5773_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx/IMU/quaternion
/Cam_robot_5773_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx/global/values
/cmd_vel
/Cam_robot_5773_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx/Lidar/laser_scan/layer0
/map
/map_updates
/robot_state_publisher
/base2nav
/tf
/Dynamic
/statistics
/Localization
/Teleop


/map
/map_updates
/name_vishwajit_Victus_by_HP_Gaming_Laptop_15_fa0xxx_5705_3231347737287965133
/initialpose
/move_base_simple/goal
/clicked_point
/robot_state_publisher
/base2nav
/tf
/Dynamic
/statistics
/rqt_gui_py_node_5917
/Localization
/rosout
/rosout
/rosout_agg
/Teleop
/wait_for_prefix

9. Backend Code

The backend is responsible for initialisation of switches present in the circuit and also mapping them to their respective roles. We have taken a modular approach for initialisation and running our features.

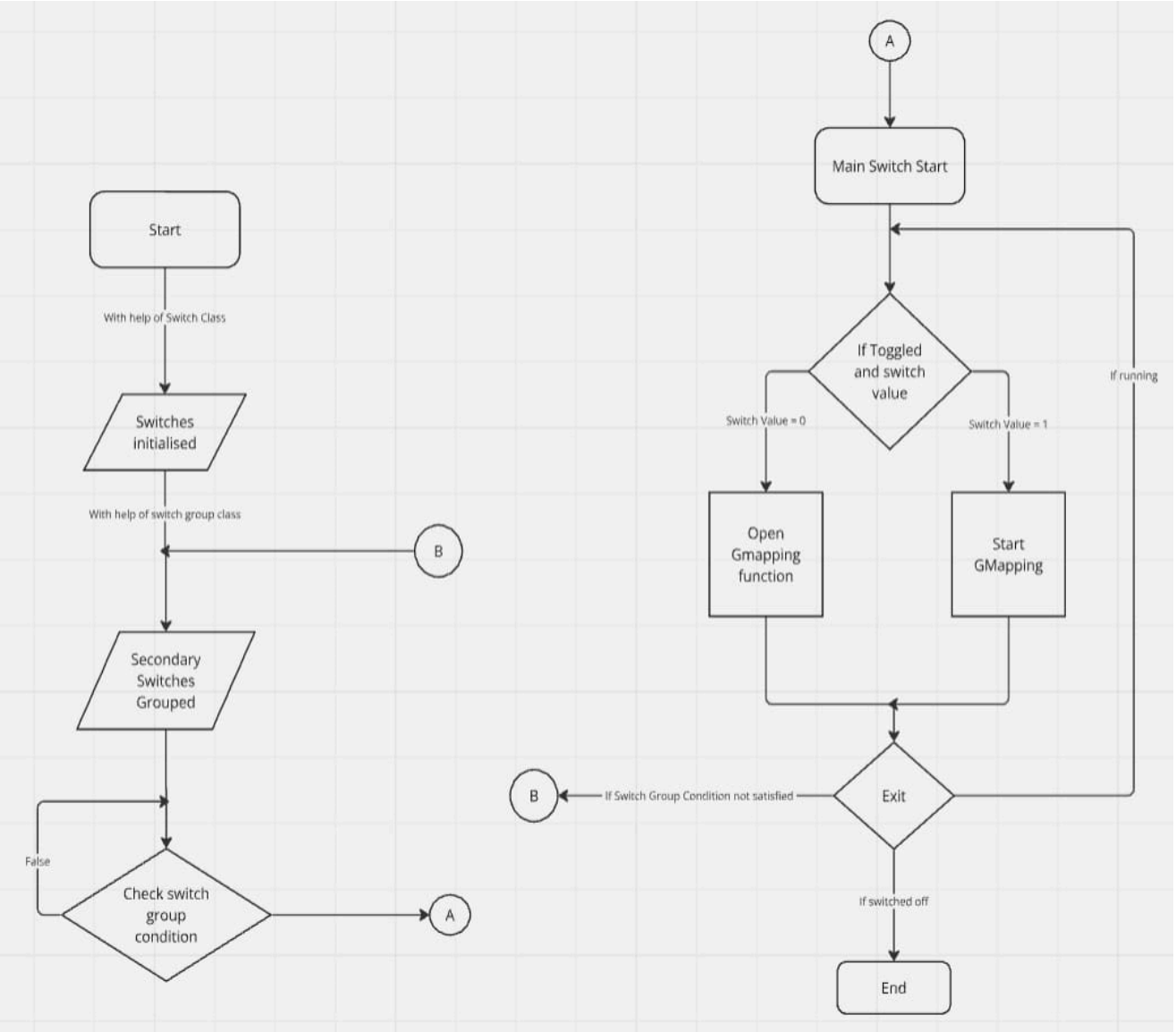
The “Switch” class handles initialisation of a switch. A switch here is defined such that it gives either True or False State. The user has complete flexibility to change it to their desired operation.

The “SwitchGroup” class handles groups of switches with a particular function. Although it appears redundant, this class provides a lot of features to ensure ease in use, improved readability and efficiency. Every switch with a common characteristic, or common functioning can be grouped together as one. In our below example, the code is such that when all the switches in the SwitchGroup are True, it activates the main switch.

The ‘Boilerplate’ class is the main class of this code. This class encases the code that runs the Gmapping features when toggled to one end, and functions when toggled to the other. To run each of our cases, we used bash scripts responsible for their functionality. The user has the flexibility to change the scripts, the boilerplate switch, and more.

The main goal of the backend code is flexibility. Given that our attempt is to make one library for anyone to use, we are trying to ensure that the library can be implemented in any shape or form. The modular approach taken enables any user, from beginner to advanced to tweak the library as per their needs.

The code above is the base example that will be provided to all users initially. They are free to change it as they will.



10. Results and Analysis

The project has achieved several significant milestones, reflecting substantial progress toward the realization of our autonomous robot.

Simulation Accomplishment:

The simulation phase has been successfully completed using Webots in conjunction with ROS 1. The simulation accurately represented the robot's movement and interactions within a controlled virtual environment, providing critical validation of our design and ensuring that the navigation and obstacle avoidance algorithms perform as anticipated.

3D Printing Preparedness:

The 3D printing designs have been finalized and are ready for manufacturing. These designs encompass all structural components, optimized for ease of assembly and durability through 3D printing. The design process involved meticulous iterations to balance material strength with weight, ensuring that the robot will be both robust and efficient in real-world applications

Boilerplate Code Development:

We have also prepared the boilerplate code for the robot's automation, establishing a strong foundation for the system's core functionalities. This code encompasses essential elements such as sensor integration, path planning, and movement control. With this groundwork in place, we are well-positioned to focus on the final development and refinement of the robot's capabilities as we move forward with scripting the ROS package.

Next Steps:

The immediate priorities include the physical construction of the robot following the 3D printing of its components, and the development of the ROS package. This package will integrate all necessary scripts and configurations for autonomous mapping and navigation. Upon completion, the ROS package will be published, contributing to the broader robotics community, particularly for those leveraging Raspberry Pi and ROS for similar autonomous systems.

11. Further Works and Improvements

The design phase of this robotics project has been successfully completed, resulting in a robot that is now ready for manufacturing and testing. The robot has been designed to be 3D printable, which allows for in-house manufacturing with relative ease. The software stack currently uses ROS 1, an older version of the Robot Operating System. However, there are compelling reasons to consider upgrading to ROS 2 in future iterations.ROS 2 brings a host of improvements over ROS 1, particularly in its use of the Data Distribution Service (DDS) as its underlying middleware. DDS offers several advantages, including improved performance, which

is crucial for high-performance data streaming applications like robotics and autonomous vehicles.

It is also designed for scalability, making it ideal for applications that involve a large number of nodes, such as cloud robotics. DDS ensures the reliability of data delivery through features like message loss detection and retransmission. Moreover, ROS 2 provides a rich set of Quality of Service (QoS) settings that allow users to control the behavior of the DDS communication layer, which is important for applications with varying requirements in terms of latency, throughput, and reliability. These enhancements make ROS 2 a valuable upgrade for future versions of the robot's software stack.

Additionally, simulation accuracy can be significantly improved by using advanced simulation packages like SimSpark. Unlike static environment simulations, SimSpark allows for the inclusion of moving objects and humans, providing a more realistic simulation environment that better reflects real-world conditions.To further enhance the robot's functionality and contribute to the ROS community, it is crucial to publish a ROS package specifically designed for robots that utilize Raspberry Pi and ROS for autonomous mapping. This package would facilitate the development process for other developers and researchers working on similar systems, enabling them to take full advantage of the power of ROS and Raspberry Pi for efficient and effective autonomous navigation and mapping.

12. References

1) M. B. Alatise and G. P. Hancke, "A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods," in IEEE Access, vol. 8, pp. 39830-39846, 2020, doi: 10.1109/ACCESS.2020.2975643.
2) Häussermann, K., Zweigle, O. & Levi, P. A Novel Framework for Anomaly Detection of Robot Behaviors. J Intell Robot Syst 77, 361–375 (2015). https://doi.org/10.1007/s10846-013-0014-5.
3) B. Kiran, S. Karthikeyan, M. A. Suhel Pasha, K. N. Manjunatha, S. Manoj Kumar and S. V. Moras, "Design and Development of Autonomous Mobile Robot for Mapping and Navigation System," 2022 IEEE Pune Section International Conference (PuneCon), Pune, India, 2022, pp. 1-5, doi: 10.1109/PuneCon55413.2022.10014944.
4) S. Jain, U. Agrawal, A. Kumar, A. Agrawal and G. S. Yadav, "Simultaneous Localization and Mapping for Autonomous Robot Navigation," 2021 International Conference on Communication, Control and Information Sciences (ICCISc), Idukki, India, 2021, pp. 1-5, doi: 10.1109/ICCISc52257.2021.9484883.
5) S. Noh, J. Park and J. Park, "Autonomous Mobile Robot Navigation in Indoor Environments: Mapping, Localization, and Planning," *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, Jeju, Korea (South), 2020, pp. 908-913, doi: 10.1109/ICTC49870.2020.9289333.
6) K. Vamsi, P. Alle, T. Brichpuria and P. Malarvizhi, "ROS Based Autonomous Disinfectant Mobile Robot for Hospitals," *2021 5th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, Coimbatore, India, 2021, pp. 94-100, doi: 10.1109/ICECA52323.2021.9675920.