Integration of ROS Navigation Stack with Dynamic Environment Information in Gazebo Simulation

Pedro H. F. Mendes
Dep. Acadêmico da Elétrica Universidade Tecnológica Federal do Paraná (UTFPR), Cornélio Procópio, Brasil, Instituto Politécnico de Bragança, Bragança, 5300-253, Portugal
pedromendes@alunos.utfpr.edu.br

André Mendes
Research Center in Dig. and Int. Robotics (CeDRI), Lab. Sust. e Tec. em Regiões de Montanha (SusTEC), Instituto P. de Bragança, 5300-253, Portugal, Dep. de Ing. de Sist. y Automática, Universidade de Vigo, 36.310, Spain
a.chaves@ipb.pt

Luís F. C. Duarte
Departamento Acadêmico da Elétrica, Universidade Tecnológica Federal do Paraná (UTFPR), Cornélio Procópio, Brasil
lfduarte@utfpr.edu.br

2022 Smart Technologies, Communication and Robotics (STCR) | 978-1-6654-6047-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/STCR55312.2022.10009315

Abstract— Sensing the environment is a crucial task that robots have to perform to navigate autonomously. Furthermore, it must be well executed to make navigation safer and collision-free. As autonomous mobile robots are being deployed in several applications, they often encounter dynamic habitats, where sensing and perceiving the environment becomes harder. This work proposes integrating a wireless sensor network with the Robotic Operating System to incorporate data into layered costmaps used by the robot to navigate, feeding the algorithms with advanced information about the territory. The architecture was tested in simulation, where we could validate the structure and collect data showing improved paths calculated and reduced computational load through better parametrization. Thus, this strategy ensures that the advanced information about the environment has improved the navigation process.

Keywords— **Autonomous Mobile Robot; Wireless Sensor Network; Robot Operating System; Gazebo Simulator; TurtleBot3**

I. Introduction

Robotics research has been evolving and growing since it was first defined. Its various applications boosted the developments from the beginning. Following the field tendency, mobile robotics has been in the spotlight for some time. Giving the robot the capability of movement through navigation only increases the possible tasks it can perform. However, navigation is not an easy mission to complete. It involves the integration of several systems that work closely together, as it can be shown in [1], in which the authors review the basics of mobile robotics as well as the sensor fusion techniques, citing the challenges of the field, and in [2], which presents basic concepts of mobile robots, algorithms applied, types of sensors, and future challenges.

Indoor guidance is one of the fields where wheeled autonomous mobile robots (AMRs) are deployed. Museums, hospitals, restaurants, shopping centers, schools, and universities are only a few examples where a guidance robot could be beneficial and make people's lives easier. The work developed in [3] reviews guidance robots and the techniques used to navigate autonomously. Navigation in these dynamic environments always implies avoiding obstacles that were not previously expected, such as a group of people, objects left out of place, and several other dangerous situations. In industrial sites, there are other possible dangers to the robots, such as temperature or dirt, depending on the industry. That noted, the more information about the environment the AMR could get, the better it will be able to navigate. The work developed in [4] uses multi-layer maps to incorporate dynamic information to achieve human-aware behavior.

Although several works have proposed navigation solutions and sensory data integration, the vast majority of these approaches only consider sensory data coming from sources on the robot's frame. Thus, this initial work proposes and simulates the robot subsystem of an architecture integrating information from distant sources, such as a wireless sensor network (WSN), which would allow the robot to have advanced knowledge of its habitat. As a result, the robot senses the environment in advanced sections of the path it is traveling, far from the range of its platform's sensor array. We achieve that kind of navigation through the Robot Operating System (ROS) tools.

The remaining of this paper is structured as follows. Section II presents the system architecture. Section III aims for the application developed for the simulation tests. Section IV-B introduces the results and discussion, and finally, a conclusion in Section V.

II. System Architecture

ROS is one of the most popular open-source platforms providing a dynamic middleware with publisher/subscriber communication, and a remote-procedure-call mechanism [5]. The different modules in the control architecture implement the functionalities mentioned above, also basic actions and report events about their state by subscribing and publishing messages, all controlled by a master acting as a broker, making it much easier to integrate different executable codes.

ROS packages are a union of software that performs some tasks together. For example, the navigation stack package. The navigation stack is a meta-package in ROS, which means it is composed of a set of packages. This set of software integrates

This work was supported through Portugal's national funds FCT/MCTES (PIDDAC) to CeDRI (UIDB/05757/2020 and UIDP/05757/2020) and SusTEC (LA/P/0007/2021).

information such as odometry, laser scans, and mapping data to control the robot. The navigation stack integrates all software needed for a robot to navigate autonomously. It includes mapping, sensing, perceiving, planning, and acting.

Although we can provide static maps to the platforms running ROS, the environments in which these robots operate usually are not static. Because of that, ROS allows us to use layered costmaps. Costmaps are grid-based maps that store information about their cells, if the cell is free, occupied, or its state is unknown. The work in [6] shows the basic idea for the implementation in ROS, comparing it with other techniques and presenting the benefits. Figure 1 illustrates this concept. The final map is a sum of all the values stored in the layers. There are three default layers in ROS, the static, the obstacles, and the inflation layer. The proposed system acts into this feature. We added a new costmap layer as a ROS plug-in, explained in [7], which receives information from any source published into a ROS topic called WSN.

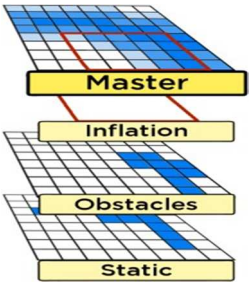

Fig. 1. Layered costmap adapted from [6].

The new layer is a C++ ROS plug-in class to which the navigation stack has access. Every layer class has its functions. The principal function of the layers is the update costs process. It is responsible for setting the values of the cells. Those values will be summed with the other layers' values and compose the master costmap. They may be free, unknown, and lethal. This work's architecture adapts the standard layer class to receive information from a topic that takes data from a WSN monitoring the robot's habitat and updating the map accordingly. The communication between the sensor nodes and the robot will be implemented with Message Queuing Telemetry Transport (MQTT) [8].

This proposed architecture allows the robot to gain much more knowledge of the environment since the source may be anywhere as long as the robot can communicate with it. The idea is that a WSN would gather the information that the robot's sensors cannot sense yet. Once the robot has this advanced data, it can optimize its actions and calculate better routes perceiving and avoiding dangerous areas much faster. Besides producing safer navigation, this advanced information about the environment allows us to tune the parameters of the navigation algorithms much better. A suit- able parametrization may lead to improved behavior and also computational load reduction. Note that this decentralization of sensory information increases the sensor coverage without boosting the robot's CPU burden since the nodes control the sensors. The data will be processed before arriving at the robot, which will only have to receive and set the costmap values.

Figure 2 illustrates the purpose of the system. It shows a map of an indoor environment. The robot was sent to the target point represented by the blue pin. At first, the robot has information about the static map and its surroundings (depending on the sensor's range). Then, an unexpected "danger", represented by the red and yellow icon, appears outside the range of the robot's sensors. With the system's advanced information, given by the wireless sensors, the robot can recalculate its routes before it gets too close to the problematic area, possibly saving time, computational load, and energy, making a safer path around the site. Also, it's possible to create temporarily blocked areas for the robot not to pass through, which could be useful in some environments (e.g., industrial plants).

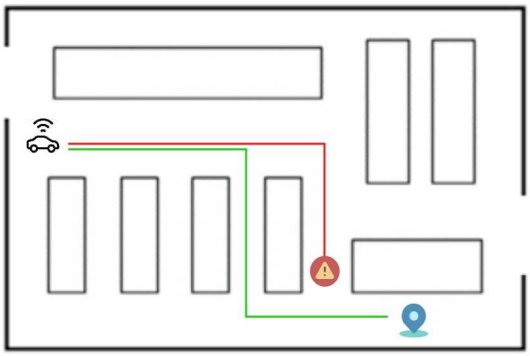

Fig. 2. System illustration.

In this paper, we will focus on developing the integration and validation (through simulation) of the new layer and the WSN ROS topic. The subsystems outside the robot (WSN) will not be developed yet. They will be addressed in future works. The simulation tests will deal with the system presented in Figure 3. The block "identify obstacles in the monitored area" in the robot system would receive the WSN information sent via wireless communication. At this point, the obstacles will be perceived in the simulation, so the MQTT and WSN systems are not in focus. As one can see, besides the C++ plug-in, two other scripts in Python were implemented. The first takes the sensed information and loads an array of data to publish into the WSN ROS topic. The second is responsible for triggering a path recalculation when the costmap is updated by the WSN.

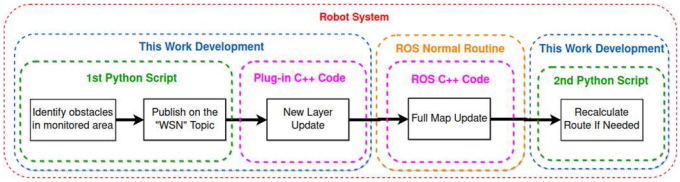

Fig. 3. System Architecture.

A. First Python Script – Array Loader

The first script is represented in Algorithm 1. It receives the simulation information (which area received the obstacle) and organizes it to be published on the WSN topic. The data

structure created is an array. It receives the coordinates of the sensed areas' locations if an obstacle is sensed. Invalid coordinates values (-999) are set if the area is clear. Figure 4 shows the array. After it is loaded, it calls the ROS function to publish that array into the WSN ROS topic.

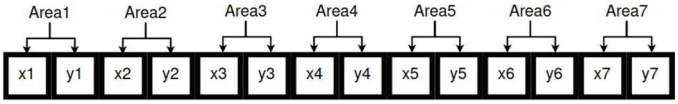

Fig. 4. Data Array.

Algorithm 1 *Array Loader*

Input: Area where the obstacle was placed (A)

```
function ARRAY_LOADER(A)
    area ← A
    array[0 : 14] = −999
    if area = 1 then
        array[0] = x_area_1
        array[1] = y_area_1
    else
        ...
        if area = 7 then
            array[13] = x_area_7
            array[14] = y_area_7
        end if
    end if
    Publish_On_WSN_ROS_Topic(array)
    end
```

B. *Plug-In C++ Code – New Layer Function*

ROS plug-ins are usually C++ classes that the ROS system can reach, but it builds and works independently from the core software. Algorithm 2 presents the New Layer logic. The layer plug-in subscribes to the WSN topic and receives the data array mentioned above. It stores the information and waits for the navigation stack to call. Once it is called, it reads the area's coordinates from the array and sets the cost of each cell in the covered area accordingly, using the ROS standard routine for that. The navigation stack then merges those costs with the other layers and updates the master costmap.

Algorithm 2 *New Layer Function*

Input: Array with the areas' information (A) and Costmap Cells.

```
function UPDATE_COSTS(A, COSTMAP_CELLS)
    for each area do
        if area_x = −999 and area_y = −999 then
            for each area cell do Set_Cost(area_cell, FREE_COST)
            end for
        else
            for each area cell do Set_Cost(area_cell, LETHAL_COST)
            end for
        end if
    end for
    end
```

C. *Second Python Script – Recalculate Route*

The second Python script is responsible for triggering the path planner recalculation. This script is necessary to adjust and optimize the route accounting for the newest environment information received. Algorithm 3 presents the logic behind that script. The method accesses the robot's data, such as odometry and the navigation goal, through ROS topics and, if needed, triggers the path planner. If the goal previously set is not valid anymore, for example, when the goal is inside a dangerous area, this script stops the robot and cancels the mission. With this implemented logic, the frequency of the global path planner may be adjusted to zero. The local planner triggers the recalculation if it detects obstacles near the robot, and this script triggers when the layer detects a change in the global environment.

Algorithm 3 *Recalculate Route*

```
function RECALCULATE()
    Goal ← current_goal()                          ▷ //X,Y, and YAW.
    Pose ← current_pose()                          ▷ //X,Y, and YAW.
    Linear_diff ← Linear_Distance_Between_Pose_and_Goal()
    Angular_diff ← Angular_Distance_Between_Pose_and_Goal()
    if Linear_diff > tolerance AND Angular_diff > tolerance then
        Plan ← Calculate_Path()
        if Plan is valid then
            Execute_Path()
        else
            Stop_Robot()
        end if
    else
        Pass
    end if
    end
```

III. APPLICATION DEVELOPMENT

To test the architecture and validate the possible gains of the proposed system, a simulation environment was prepared in Gazebo [9]. At first, the world model that simulates the blueprint of a school floor was built, which will be the site of the study. Figure 5 and Figure 6 show the simulated world and the floor's blueprint.

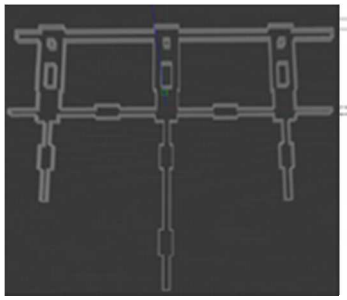

Fig. 5. Gazebo world.

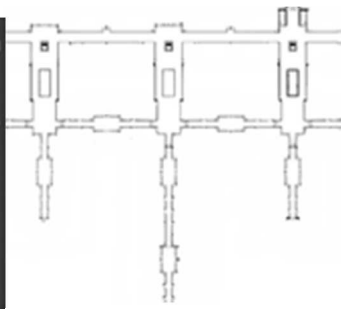

Fig. 6. Real blueprint.

The robotic platform used in the simulations was the TurtleBot3 (TB3). The TB3 is an open-source and collaborative effort led by ROBOTIS, the Open Robotics organization, and other partners. The TB3 is not only a ROS standard platform, it is the most popular one, having an active community of developers that facilitate educational projects and research [10].

Inside that world, we proposed three routes the robot would have to perform as if it was guiding someone around the floor. Besides the routes, we have chosen seven areas on the map where the monitoring would be done. They were called "zones". Considering where people usually stand, these seven zones were chosen, creating a possible danger for the robot by blocking the corridors. Figure 7 shows the routes and

monitored zones. It also shows the sensor nodes (ESP8266 controlled) and the MQTT broker (Raspberry Pi 3 B), which will be used in future works to collect information. With this centralization, one can remove some computational load from the robot and make the system scalable to any

number of robots since they would only have to subscribe to the MQTT topic. In the figure, we have points “A” to “D”. The first route goes from “A” to “B”, the second “B” to “C” and the third “C” to “D”. The algorithm raffles the location to place the obstacles for each route, once the robot gets to the goal the obstacles are cleared for the following route to begin. The red-colored area represents the monitored zone 4 (Z4). Each route will be performed 50 times for two configurations explained in the following paragraphs.

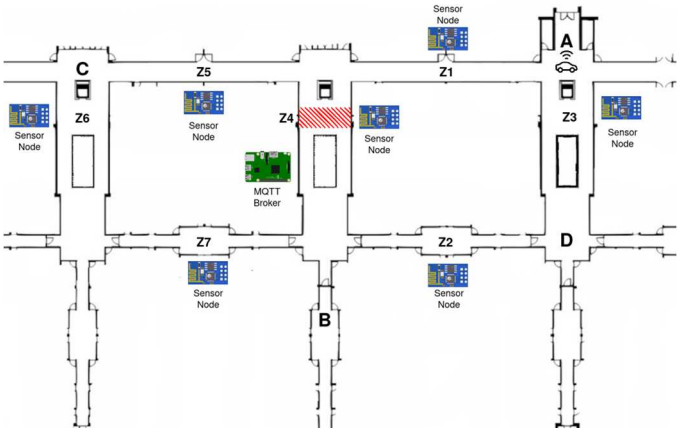

Fig. 7. Simulation scenario.

IV. EVALUATION

A. Simulation

To facilitate the gathering of information and to produce a high number of tests, a Python script was developed to control aspects of the simulation and automatize the runs. Each run is counted as the completion of the three routes. That way, we were able to simulate each configuration several times and have more accurate results. This script handled tasks such as the launch of all ROS structures, the placement of obstacles (randomly inside each route), setting navigation goals, and the creation of log files for analysis. Each test was conducted with fifty runs. The danger to the robot is simulated as obstacles blocking the passage completely. For the first route, the obstacles are placed either in the first monitored zone or the fourth. For the second path, the algorithm randomly places the obstacles in zones four or five. The third route gets zones one, five, and three randomly obstructed. A second Python script was produced to analyze all the log information generated by the first one. We monitored data such as computational load and memory usage of the ROS processes, time to complete each route, and distance traveled. All this information was treated using Python libraries such as Pandas and Numpy.

The simulations compared two configurations of the TB3 navigation stack. The first (called “default") is a parametrization with minimal changes from the default values and without the architecture’s advanced information. The second (called “full”) will have our architecture, and some parameters were optimized considering the proposed system. The most rewarding ones in terms of CPU consumption were the reduction of the local costmap size and the path planner calculations.

B. Results and Discussion

The first thing we see as a result is behavior change. Figure 8 (default) and Figure 9 (full) show the visualization tool used in ROS to illustrate the change in the conduct of the robot. In both, the first moments of the simulation are presented. Here, obstacles obstructed the passage in zone 4 (in both), but only the second had already placed the danger on the map (pink-colored area). The full configuration already avoids the problematic area, and the default hasn’t yet detected the danger in the route. We can say that the integration of the ROS topic with the costmap through the new layer worked well.

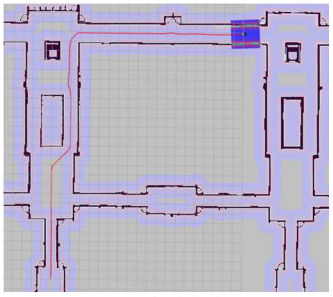

Fig. 8. Default behavior.

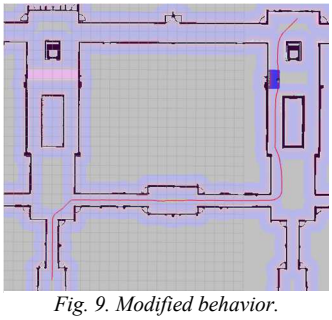

Fig. 9. Modified behavior.

Now, focusing on the system performance, we analyzed Figure 10 and Figure 11. They show, respectively, the mean and 95% confidence interval of the distance traveled, and the time spent on each route. The blue bar represents the full configuration. The orange bar illustrates the default configuration. These results show that in the case that the danger makes the crossing impossible for the robot, the advanced information permits anticipation and correction of the routes in time to reduce the time and distance traveled. The ability to “predict” what is going to be in the way, in this case, produces these reductions as the obstacles or danger does not have to be inside the sensor’s range, so if the robot has to change the path, it saves time and distance traveled to detect the problematic area. To compare the parametrization, the script monitored the computational load used by the move base process, which is the central node that integrates the algorithms and sends the controlling commands to the drivers.

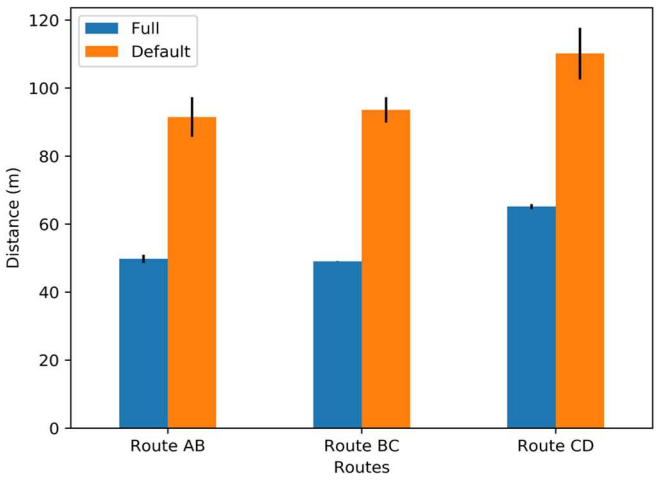

Fig. 10. Distance traveled for each route.

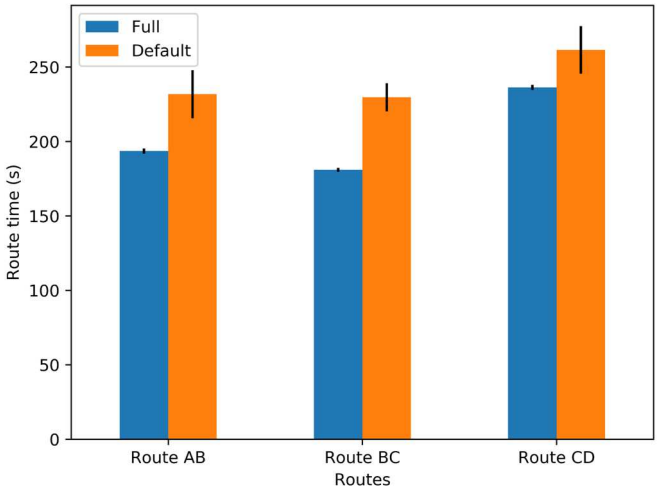

Fig. 11. Time to complete each route.

In Figure 12, we can see a substantial reduction in computational load. This achievement was possible by tuning some parameters considering the advanced information of our architecture. For example, the sizes of the costmaps were reduced, and the frequency of path planning and map updates. It is a notable result since embedded boards with limited processing power usually control mobile robots. The chart below shows CPU percentage values over one hundred, which happens in multicore systems when the process runs in more than one core.

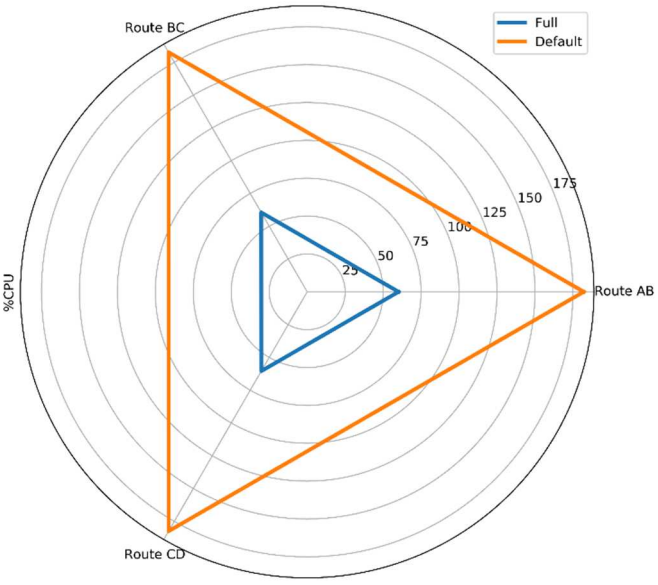

Fig. 12. Move base percentage of CPU consumption.

V. Conclusion and Future Works

This paper presented and demonstrated aspects of a different concept to integrate advanced dynamic environment information with AMRs navigating with ROS. The introduced architecture was designed with layered costmaps and ROS topics that receive information about the environment from sources outside the robot's sensory structure. Algorithm 1 successfully introduced the areas' information into the ROS framework. Algorithm 2 allowed the navigation stack to account for the new cell costs, and Algorithm 3 guaranteed the global path recalculations only when it was necessary. Through the simulation produced, we validated this fusion and showed some of the possible gains. The system produced behavior that avoids danger zones with faster detection and reaction. This conduct opened the possibility of adjusting critical parameters of the navigation stack, reducing the computational load. In future works, the system will be implemented on a robotic platform. Also, the WSN, with its sensor nodes and the central broker. That way, we can compare the simulated and real results.

Acknowledgment

The authors are grateful to the Research Centre in Digitalization and Intelligent Robotics (CeDRI) for sharing its facilities and material, and also to the Foundation for Science and Technology (FCT, Portugal) for financial support as mentioned earlier.

References

[1] Mary B Alatise and Gerhard P Hancke. A review on challenges of autonomous mobile robot and sensor fusion methods. IEEE Access, 8:39830–39846, 2020.

[2] Md AK Niloy, Anika Shama, Ripon K Chakrabortty, Michael J Ryan, Faisal R Badal, Zinat Tasneem, Md H Ahamed, Sumaya I Moyeen, Sajal K Das, Md F Ali, et al. Critical design and control issues of indoor autonomous mobile robots: A review. IEEE Access, 9:35338–35370, 2021.

[3] Debajyoti Bose, Karthi Mohan, Meera CS, Monika Yadav, and Deven- der K Saini. Review of autonomous campus and tour guiding robots with navigation techniques. Australian Journal of Mechanical Engineering, pages 1–11, 2022.

[4] Fang Fang, Manxiang Shi, Kun Qian, Bo Zhou, and Yahui Gan. A human-aware navigation method for social robot based on multi-layer cost map. International Journal of Intelligent Robotics and Applications, 4(3):308–318, 2020.

[5] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In ICRA workshop on open source software, volume 3, page 5. Kobe, Japan, 2009.

[6] David V Lu, Dave Hershberger, and William D Smart. Layered costmaps for context-sensitive navigation. In 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 709–715. IEEE, 2014.

[7] Creating a new layer. Available: http://wiki.ros.org/costmap 2d/Tutorials/CreatingaNewLayer. Accessed: January 2022.

[8] Biswajeeban Mishra and Attila Kertesz. The use of mqtt in m2m and iot systems: A survey. IEEE Access, 8:201071–201086, 2020.

[9] Jack Collins, Shelvin Chand, Anthony Vanderkop, and David Howard. A review of physics simulators for robotic applications. IEEE Access, 9:51416–51431, 2021.

[10] Robotis e-manual. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/#overview. Accessed: June 2022.