Inexpensive Autonomous Rovers for Multi-Agent Applications

*Freedom Rover Units

1st Jordy A. Larrea Rodriguez
Department of Electrical and Computer Engineering
University of Utah
Salt Lake City, USA
Jordy.larrearodriguez@gmail.com

2nd Brittney L. Morales
Department of Electrical and Computer Engineering
University of Utah
Salt Lake City, USA
brittneymrls@gmail.com

3rd Misael Nava
Department of Electrical and Computer Engineering
University of Utah
Salt Lake City, USA
misaelnava812@gmail.com

***Abstract*—The state of the art in autonomous swarms employs a decentralized model consisting of multi-agent networks. These robotic collaborative systems hold the potential to adapt to new environments and optimize individual performance to specific tasks without having to deal with global systems prone to single points of failure. Our team's focus lies therein in developing a multi-agent system capable of a decentralized network. The swarm will incorporates three two-wheel differential drive rovers interfaced through the Robot Operating System (ROS) via wifi telecommunication through the micro-ROS agent service. The central bay system consists of a single laptop to communicate objectives for the agents to complete (carefully designed demos). Our development stack will leverage ROS for project management, simulation capabilities, navigation libraries, and native server-client model in robotics applications. The rovers AI will incorporate simultaneous localization and mapping (SLAM) techniques for RT positioning based on a priori grid or map; thus, facilitating navigation through improved state space mapping. Thus, we introduce FRU-bot, an autonomous platform capable of deployment in decentralized and centralized systems.**

***Index Terms*—Decentralized Communication, Swarm Communication, Multi-agent, ROS, Gazebo**

I. Introduction

Our team's primary objective was to design a networked swarm of rovers capable of collaborative mapping, obstacle avoidance, and cooperative task execution. Our aim is to establish a foundational system that aids in Search and Rescue operations and facilitates research in multi-swarm rover technology. Presently, we've achieved success in creating rovers equipped with mapping capabilities, allowing real-time simultaneous visualization of surrounding areas within a simulation. Moreover, our accomplishment includes developing these rovers at an affordable price point, under $100 each (Fig. 6). This cost-efficient solution stands in contrast to market offerings that typically start at $300 for a turtle bot with a Jetson nano and increase steeply thereafter.

II. Background

A plethora of elements must come together to successfully deploy a swarm of agents. Our system requires a working knowledge in both the physical and abstract. The agents are subject to their physical characteristics: i.e., their circuitry, power consumption, mechanical components, sensors, processing power, and geometry to navigate and respond to an environment. The complexity that drives cyber-physical systems is what largely makes robotics difficult. An agent must be able to function perfectly in a world with a continuous state space. Today, researchers are seeing a high degree of success with robots that incorporate mathematical and data driven modeling that leverages the superior computational capabilities of modern processors to solve complex continuous problems in robotics.

A. Common Robot Autonomous Platforms

Common mobile platforms for autonomous robots encompass a range of designs tailored to diverse applications within the field of robotics. Wheeled platforms, such as differential drive and omnidirectional robots, are popular choices due to their simplicity and efficiency in navigating various environments. Differential drive platforms use two powered wheels, allowing the robot to turn and move forward or backward by adjusting the speeds of the wheels. Omnidirectional platforms, on the other hand, utilize special wheel configurations to achieve unrestricted movement in any direction. Lastly, tracked platforms offer enhanced traction and stability, making them suitable for challenging terrains.

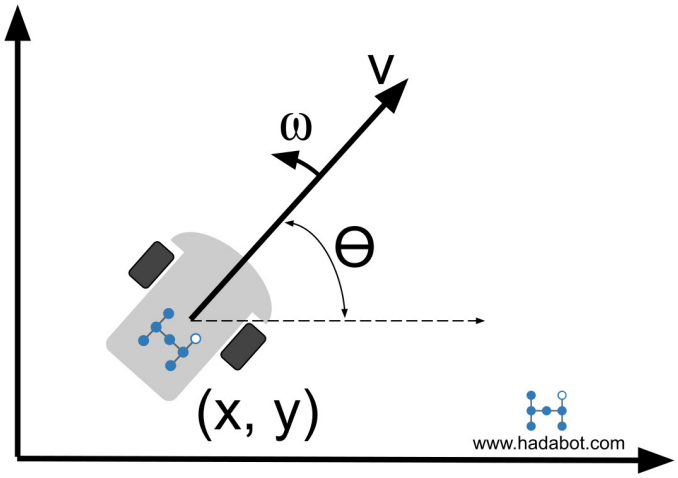

Fig. 1. Graphical representation of 2D robot state configuration standard to 2 wheel differential robots [1].

One notable platform is the TurtleBot 3, a popular open-source robot platform widely used for research and education. TurtleBot 3, developed by ROBOTIS, is a compact and modular robot equipped with various sensors, including a 360-degree lidar sensor, a camera, and inertial measurement units (IMUs). Its differential drive system allows for omnidirectional movement, making it versatile for navigation in dynamic environments. The TurtleBot 3 is often employed in the development and testing of autonomous navigation algorithms, mapping techniques, and obstacle avoidance strategies. Its affordability, ease of use, and vibrant community support make it an attractive choice for researchers, students, and hobbyists interested in exploring the intricacies of autonomous robotics.

B. Robot Localization and Navigation

Normally, robots use a wide range of sensors to inform themselves about the surrounding environment. Our agents, for example, rely on a 6 axis inertial measurement unit (IMU) and rotatory encoders to produce an $\{X, Y, \theta, \dot{X}, \dot{Y}, \dot{\theta}, \ddot{X}, \ddot{Y}\}$ state configuration at every time step by considering physical properties and mechanics (Fig. 1). However, a frequent problem with using rudimentary techniques solely for odometry is that error propagated at each time step due to environmental conditions, sensor specifications, electrical noise, and rounding error can deviate a naive agent during localization. Mobile robots regularly employ an Extended Kalman Filter (EKF) algorithm for state configuration estimation inferred from fused odometry sources such as rotary encoders and IMU data [2].

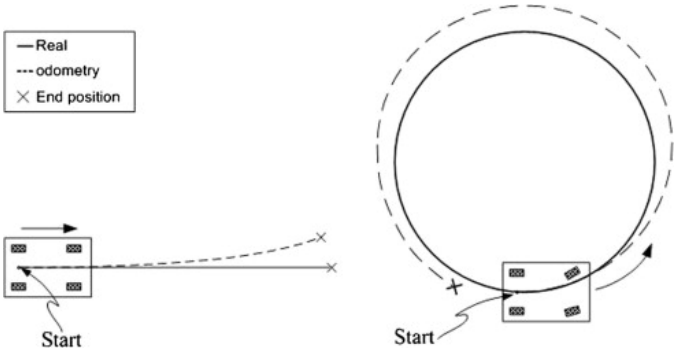

Fig. 2. Graphical representation of basic odometry error as a robot follows a reference trajectory [3].

Odometry error propagates throughout the entire run-time of a robot which can lead to poor real-time localization and navigation. As seen in Fig. 2, over time, a robot that relies solely on wheel odometry will eventually delineate from its target trajectory. 2WD robots often only consider the angular velocity in the yaw from IMU sources to fuse with the wheel odometry. Furthermore, a 2D state space is normally assumed to reduce the complexity of the robot's kinematics and to offset the error of approximated angular velocity by the odometry. Localization via EKF can become more reliable with more sources of odometry. However, the problem with error persists even with the use of a known map and highly accurate sensors.

Consider how a human might traverse a city by using a map. The human could ensure their own position by simply pondering their place in respect to features of the map like buildings, intersections, or street signs. Even if the human absent mindlessly navigated the city, they could triangulate their location by simply calculating their positions based on observations (such as the coffee shop across the street). For an agent, SLAM leverages known knowledge of poses at past time steps and emissions/observations (sensor readings) made as the agent accumulates error through basic odometry to calculate probabilities of where the agent can be in space. One can think of a pose as the transformations needed to get from the previous state configuration to the current. Any number of sensors that detect 'emissions' from the environment can be used for SLAM; however, 2D lidar and cameras are regularly used for the high degree of precision awarded. Agents are capable of both learning a new environment and updating a known map with SLAM; thus, justifying our deployment of SLAM to improve the navigation capabilities of each agent [4].

C. ROS2 and Open Source Frameworks

Robot Operating System 2 (ROS 2) is an open-source middleware framework designed to facilitate the development of robotic software. Building on the success of its predecessor, ROS, ROS 2 was initiated to address the limitations and enhance the capabilities of the original system. ROS 2 is developed and maintained by the Open Robotics organization,

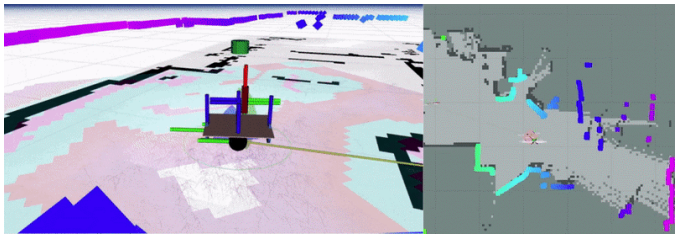

Fig. 3. SLAM simulation on a robot by the Linorobot package [5].

which aims to provide a flexible and robust platform for building advanced robotic systems [6].

The framework supports multiple programming languages, including C++, Python, and others, allowing developers to work with the language of their choice. ROS 2 fosters interoperability, scalability, and maintainability, making it well-suited for a variety of robotic platforms, from small embedded systems to large, complex robots. With its emphasis on distributed systems, real-time capabilities, and improved security, ROS 2 has become a pivotal framework in advancing the development and deployment of autonomous robots across various industries. Its active and growing community continues to contribute to its expansion and refinement, solidifying ROS 2 as a key player in the realm of robotic software development.

A multitude of packages, simulation/visualization tools, software stacks, and robot interface drivers are made available. Of note, ROS2 has notable localization, SLAM, and navigation packages for mobile robots and manipulators. The combined power of free and open source industrial quality packages and the framework's data distribution service (DDS) allow for seamless implementation of new robotic models. The event driven publisher-subscriber implemented by the ROS2 DDS allow for soft RT that supports large and complex robotics projects.

D. Robot Description and Transformation Trees

In ROS, robot descriptions play a pivotal role in defining the physical characteristics and kinematics of a robotic system. Robot descriptions are typically represented using the Unified Robot Description Format (URDF) or the Robot Description Format (SDF). These XML-based formats allow developers to specify the robot's geometry, joint properties, sensor configurations, and other essential details.

The robot description serves as a crucial component for simulation, motion planning, and control within the ROS framework. It provides a comprehensive and standardized way to encapsulate the robot's structural and dynamic properties, enabling seamless integration with various ROS tools and libraries. This description becomes particularly valuable in the context of robotic simulations, where developers can accurately model and visualize the robot's behavior before deploying it to physical hardware. The clarity and consistency offered by well-defined robot descriptions in ROS contribute to the framework's effectiveness in fostering collaborative and interoperable development across diverse robotic applications.

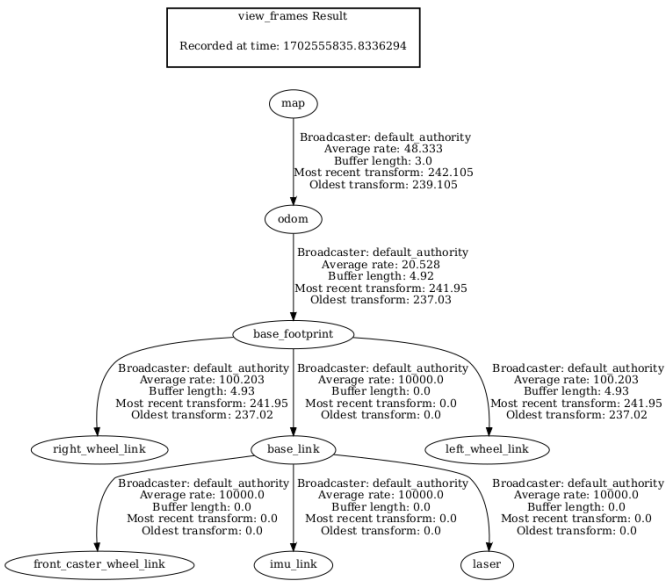

Fig. 4. Transformation dependency tree of the FRU-bot.

Sensor noise distributions, world friction parameters, sensor specifications (lidar range and resolution eg), and available kinematic controllers (and more) are definable via the robot description. ROS uses the URDF to calculate transformations vital for robot localization and navigation. These transformation dependencies are sensitive to noisy sensor readings and lack of odometry sources, and require that the robot node network is dutifully designed. Robot name-spaces and frame ID prefixes are often invoked to isolate nodes on a particular robot or to separate frame-IDs delegations for the robot state publishers respectively.

E. Gazebo for Robot Simulation

Gazebo is a powerful physics simulation environment for simulating robotic systems. Developed by the Open Source Robotics Foundation, Gazebo provides a dynamic and realistic 3D simulation platform for testing and validating robotic algorithms, control strategies, and system behaviors. It supports the simulation of various sensor inputs, physics-based interactions, and complex robotic scenarios, offering an invaluable tool for researchers, developers, and engineers working on robotic projects.

Gazebo enables the deployment of detailed robot models and their environments, allowing users to simulate and visualize robot movements and interactions before deploying code to physical hardware [7]. Its integration with the Robot Operating System (ROS) further enhances its utility, facilitating seamless communication between simulated robots and other ROS-enabled components. Gazebo's flexibility, extensibility, and open-source nature make it a widely adopted tool in the robotics community, fostering innovation, collaboration, and the efficient development of robotic systems [8].

F. Multi-Agent Swarm Systems

Collaborative robots in the context of ROS involve the coordination and collaboration of multiple robots or agents working together to achieve common objectives. ROS provides a flexible and modular framework for developing and implementing multi-agent systems, allowing for the seamless integration of individual robotic entities into a cohesive network. One key advantage of using ROS for multi-agent systems is its robust communication infrastructure, which enables efficient exchange of information among agents. The publish-subscribe model in ROS facilitates inter-agent communication, allowing robots to share sensor data, coordinate movements, and collectively solve complex tasks.

In a multi-agent ROS environment, each robot typically runs its own instance of ROS, allowing for decentralized control and decision-making. This decentralized approach enhances system scalability and fault tolerance, as the failure of one agent does not necessarily disrupt the entire system. ROS facilitates the creation of custom message types and services, enabling agents to communicate specific information tailored to the multi-agent system's requirements. This adaptability is crucial in scenarios where diverse robotic platforms with varying capabilities need to collaborate, such as in search and rescue missions or swarm robotics applications.

The integration of multi-agent systems with ROS also extends to simulation capabilities. Gazebo, allows developers to model and simulate multiple robots simultaneously. This capability is instrumental in testing and refining multi-agent algorithms, ensuring their effectiveness in real-world scenarios. Given the complexity of node dependency graphs for MAS, the ability to debug, test, and validate simulated system networks and algorithms is monumental to decreasing overall development time and wear/tear on the physical robots.

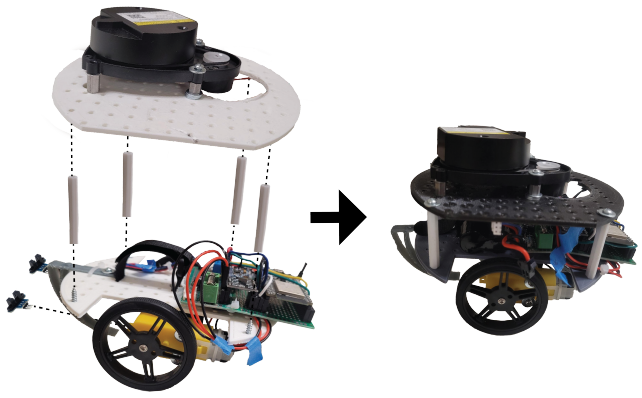

Fig. 5. Parts of FRU-bot, including the chassis, wheels, and electronics on the left. The final build of FRU-bot on the right.

III. Project Implementation

A. Preliminary Robot Chassis

Our rover is designed with two distinct levels, both 3D-printed boards with holes to screw in different components. The upper level features the LiDAR for a clear view, with the wires going through the center of the 3D-printed board. On the lower level, our lithium batteries are securely attached using Velcro at the rear. The power board is positioned at the front, fastened by two of the four poles securing both levels. Beneath the bottom board, two TT motors are firmly fixed to the 3D-printed board using screws, alongside encoders secured with a hot glue gun adjacent to the motors. Wires from the encoders, motors, and LiDAR are routed towards the rear, adjacent to the batteries, and connected to our power board. This central hub houses vital components such as the ESP32-wroom, IMU, and H-bridge.

The two levels are connected using 3D-printed poles, approximately 1½" in length. Additionally, a roller ball bearing is positioned at the rover's front, crucial for even weight distribution and smoother turns. Its absence previously caused calibration errors as the rover rocked back and forth during testing.

Part	Quantity	Cost
LDS-01 LiDAR	1	$39.99
7.4V 2000mAh li-ion	1	$11.99
ESP32-WROOM-DevKitC	1	$10.00
Rotary Encoders	2	$1.60
TT Motor	2	$1.37
Dual Motor Driver	1	$1.37
Caster Wheel	1	$1.06
Power Interface Board	1	~$12.99
3D Printed Chassis	1	~$5.00
Misc. Hardware	NA	~$5.00
Total Cost Per Robot		~$94.61

Fig. 6. Breakdown of part costs per FRU-bot rover.

B. Preliminary Electronics Design and Testing

One of the hardware considerations is what board will run the rover firmware. Luckily, the choice of using ROS2 and Mircoros narrowed the choices down substantially. Microros provides a list of supported development boards. These boards range from Arduino, STM32, and Teenys to ESP32. Looking through the list of specific boards, it quickly became apparent that many were out of the budget. To give an idea, the two cheapest boards are the Raspberry Pi Pico (averages a cost of 6 dollars) and the ESP32-DevKitC-32UE/E (10 dollars). While the next cheapest board is the Teensy 4.1, which clocks in at about 30 dollars per board (a heavy price increase). With the choice coming down to either the Pi Pico or the ESP32, the final choice is ESP32 since it has WiFi built within the chip and is an entirely new option for experimenting. ESP32 developers, Espressif, drivers for I2C, UART, PWM, and ADC. These drivers are essential for interfacing with the sensor to determine pose.

For the rovers to fully function, they must take advantage of odometry, kinematics, and PID. To handle these calculations, the feedback/data from wheel movement and inertial acceleration allows for deriving the rover's pose. The infrared rotary encoder provides feedback to determine the speed of each motor, handling PID, and parts of odometry/kinematics. DAOKI offers a Slotted Optical IR encoder that is within the budget and a part that attaches itself to the TT motors for easier reading of motor speed. Although this encoder provides feedback, it is not enough to accurately determine the rover's pose, which is why the MPU 6050 gives us additional information for deriving the accuracy of its movement by measuring inertial acceleration. Between the MPU6050 and the DAOKI encoder's pose, odometry, kinematics, and PID are accurate enough to serve the parameters of the project. Now, concerning SLAM we could stick with just those two sensors, but the mapping and location would not be satisfactorily accurate without a lidar sensor. Our rovers feature the LiDAR HLS-LFCD2, LiDAR, an acronym for Light Detection and Ranging, utilizes light pulses to gauge distances from its location. To interface with the LiDAR, we use a serial connection to the microcontroller, where data received undergoes conversion to extract ranges and intensities. The processed data are then transformed into Laser Scan messages and relayed to the host computer. On the host side, this information serves as pivotal input for simulating a detailed map reflecting the rover's surroundings. This simulation acts as an invaluable debugging resource, ensuring alignment between the rover's real-world operations and the expected functionality within the simulated environment.

Testing for the hardware and sensors is divided into three steps: reference projects, ESP driver development, and testing on the ESP32 themselves. The first step for testing all sensors was finding projects/tutorials online. These tutorials offer a path to narrow down what parts of the datasheet are necessary for our interaction with the sensors. One example is the MPU6050 tutorial showing which addresses are relevant for XYZ gyro acceleration, setup for MPU before reading from those registers, along the math needed to be on the data to determine its angle. Many of these tutorials were in Arduino and meant for either the UNO or PI Pico, which the team had on hand from previous classes/projects, making the preliminary testing of seeing if the components were not defective a simple process. The next step is reshaping the tutorials into C using the Espressif drivers. Once refactored, the code is tested on the component again to ensure functionality is the same as the original testing, the new code is adjusted to fit the needs of Linorobot.

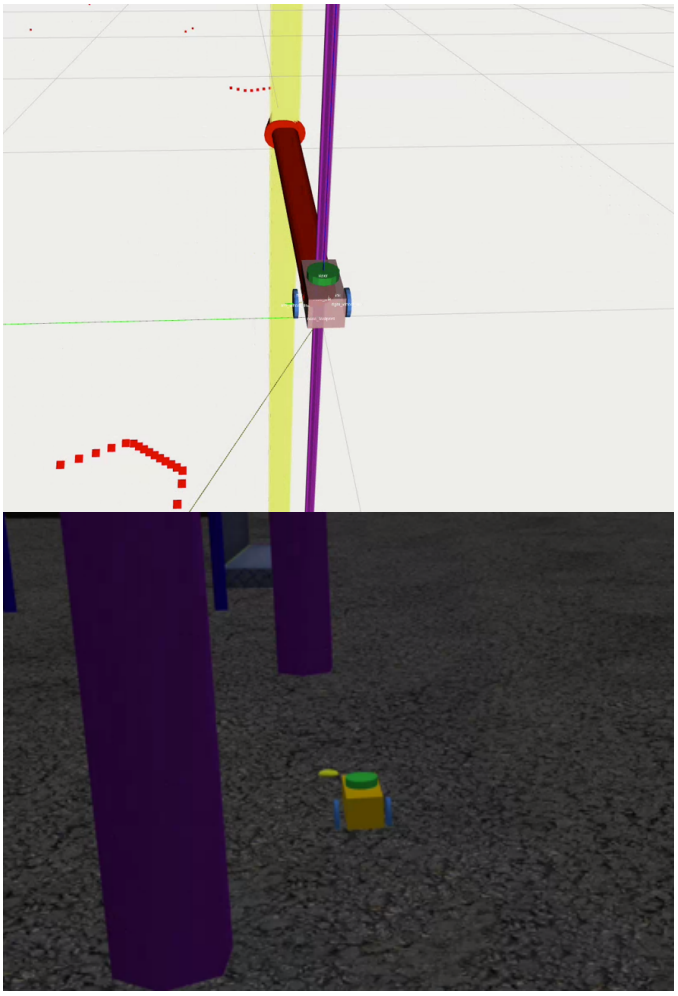

Fig. 7. Visualization of preliminary simulated robot at the top, and the same view in the simulation gazebo environment at the bottom.

C. Preliminary ROS2 Robot Description Design and Simulation

Originally, our team used Linorobot's "linorobot2" package to experiment with Gazebo's simulation capabilities on default parameters and robot models. Thereafter, the robot description package for linorobot2 was altered to fit the physical dimensions of FRU-bot's chassis (visual and collision geometry) and its sensor specifications: min/max lidar ranges, motor torques, update frequencies, etc. In Fig. 7, one can see the odometry direction vector depicted as a red arrow and the detected purple pillars as red dots. Furthermore, the lagging line connecting the base frame to the odometry frame can be seen trailing off the lower left hand corner of the visualized robot.

Our results from preliminary testing and simulation with ROS2 facilitated eventual integration with our eventual prototype. Simulation of FRU-bot at this early stage helped establish a baseline for what the node network should look like at the later stages.

D. Hardware/Firmware

Linorobot is the skeleton for the hardware of our little rovers. We attempted to use the original code built on Arduino and PlatformIO, but the project started with the team developing drivers on the Espressif IDF. The issue, however, is Arduino did not have any official documentation on the supported Espressif libraries/boards. A lack of documentation and not understanding Linorobot2 on how to change/add in our drivers left us with a decision. Dive deeper into and translate the drivers into Arduino/C++ or translate the main.c file along with kinematics, odometry, and PID over to Espressif/C and make sure our drivers follow a similar structure to Linorobot's drivers in terms of formatting Microros data. Ultimately, the team found it easier to bring over the skeleton of Linorobot than to redo the drivers.

For interacting with the MPU6050, the communication protocol is I2C. The setup for the MPU is simple; a zero byte is written to the power management register before data is to be read. After writing the zero bytes, the next step is measuring the error. We are only concerned with the GyroZ register. To calculate the error, while the MPU is still and data is being sampled multiple times, the average is taken to know how much drift there is within the system. The next time the register is read, we subtract that error from the reading to get an accurate value. It is also important to note that the value read must first be divided by 131 according to the datasheet to convert the value to scale.

Encoder interactions only require counting pulses. Essentially, the encoders only have three pins: power, ground, and out. Out is either high or low on whether an object is blocking the IR sensor(in our case rotary attachments to the TT Motors). To achieve RPM calculations, use a PCNT(pulse counter) peripheral to count the rising edges of the Out pin outputs. Then it is a simple problem of using the number of pulses recorded since the last sample divided by the time since the last sample (dx/dt) and converting it to an RPM. For the DAKOI attachment, twenty triggerings of the sensor represent a full rotation.

Analog-digital conversions did not require too much setup, mostly calibration steps were required from the Espressif. Very little needed to be changed from the example code keeping in the relevant calibration sequence chosen, which was a linear fitting. The ADC was used to detect battery voltage; sampling was done at a voltage divider since the ADC could only do a maximum of about 3V. If the sample was less than or equal to 2 V, the overall voltage of the battery was low, and continuing usage resulted in damage to the components.

The TT Motors were bought via Amazon and arrived approximately a week after ordering, priced at approximately $6 per motor. Upon arrival, a visual inspection ensured there were no visible defects. Each motor featured a white rod on both ends, creating a T-shaped appearance, facilitating attachment of the wheel on one side and an encoder on the other to track rotations. The initial functionality test involved connecting the red wire to 5 volts and the black wire to ground,

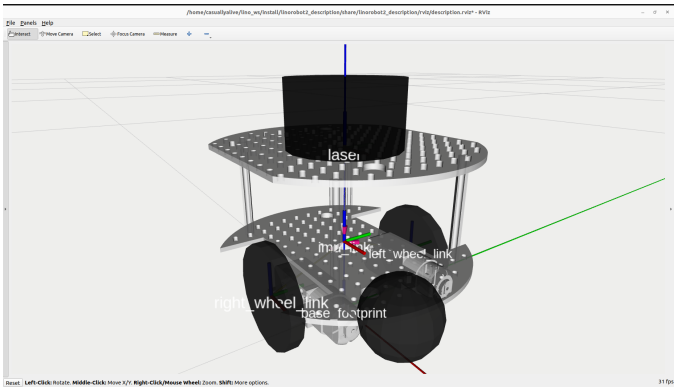

Fig. 8. Visualization of final FRU-bot chassis in rviz2.

which verified proper operation for all motors. Subsequently, assessing compatibility between the microcontroller, h-bridge, and the motors involved validating directional control for forward, backward, and turning movements. Manipulating speed required utilizing PWM within the Espressif framework, which posed challenges in integrating their new PWM drivers. After encountering multiple issues, a pivot to their older PWM driver ensued, demonstrating immediate functionality and seamless integration into the primary rover code.

Our rovers are equipped with the LiDAR HLS-LFCD2, utilizing a usart-based serial connection. Procured from eBay at an approximate cost of $40 per unit, arriving in about one month. Upon arrival, our initial objective was to validate their functionality. To verify functionality, we meticulously followed the instructions provided by Matthew Hogan's tutorial in Ïnteresting Electronic Components #1: HLS-LFCD2.Ïhis guided us through the process of interfacing the LiDAR with a Raspberry Pico, utilizing the Arduino IDE. This comprehensive procedure spanned nearly a month, involving debugging of provided code and checking hardware. Ultimately, the successful operation allowed us to generate x and y data points. Having verified functionality, our focus shifted towards transitioning the system's programming from C++ to C and migrating to the Espressif framework instead of the Raspberry Pico. This strategic shift was a necessity as we are utilizing ESP32-wroom as our microcontroller for each rover.

After extensive refactoring from the Linorobot2 package, and overcoming several challenges, we achieved successful data publication from the lidar to our local host. This data facilitated real-time simulation, enabling mapping of the rover's surroundings. This accomplishment allowed us to showcase a live demonstration during our scheduled demo day.

E. ROS2 Packages for FRU-bot

Further improvements were made to our ROS packages, including complete deviation from the linorobot2 fork and major changes to the robot description, config, and launch files. In order to allow simulation and deployment of multiple robots, two strategies had to be implemented: ie, robot name-spacing and URDF frame prefixing.

Name-spacing essentially encapsulates a robot or specific processes under one name. For example, a name-space can isolate a robot's state publisher, cmd subscribers, and controller to a name. Normally, ROS publishes transformations to the "/tf" topic and sensor data to topics of format "/sensor". This approach functions well during single robot operations or when only one source of data or control is assumed. However, in more complex systems where there are multiple sources of the same sensor data or control (or multiple entities), name-spaces are usually implemented to encapsulate similar processes under the same identity (refer to Appendix D - bottom image).

Launch scripts were implemented to dynamically change configuration files and node instances at run-time (bringup.launch.py). This effectively generalized FRU-bot's configuration files; ultimately, permitting the same parameters for each robot with changes executed at run-time. Furthermore, the launch scripts allow a user to start a simulated or real FRU-bot instance with the same command. Our launch scripts feature optional execution of a robot's nodes at a specific name-space to facilitate execution of multiple robots in simulation and the real world (refer to Appendix C).

Lastly, the robot description features arguments that can assign an optional name-space or frame prefix at run-time. The frame prefix grants dynamic renaming of robot frames. This is useful if the system specifications demand that the same "/tf" topic is used. For example, in order to deploy a centralized model, every agent must be aware of the other robots relative locations, this can be calculated easily when using the same transformation tree. On the other hand, the name-space argument optionally remaps simulated topics to the robot's name-space. Both were implemented, for potentially continued development in multi-agent centralized and decentralized models with FRU-bot.

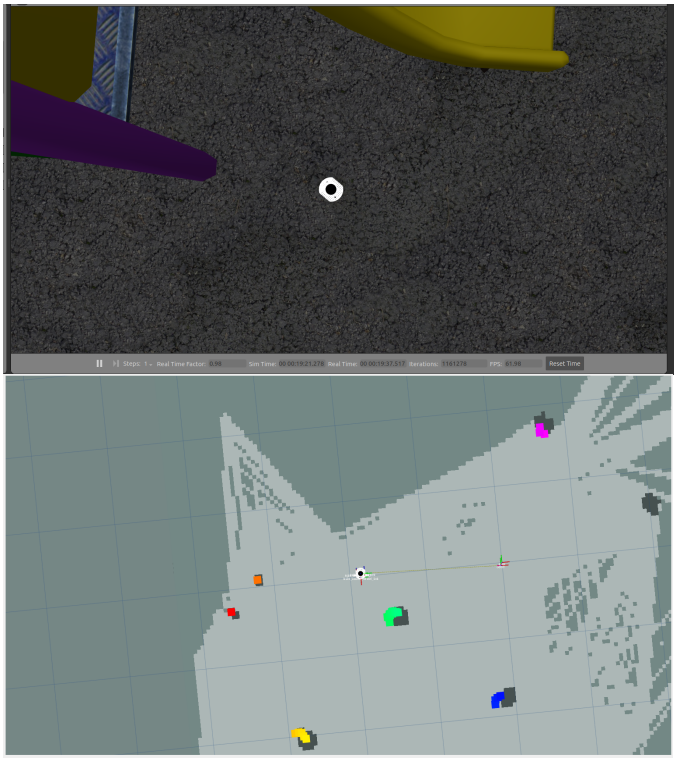

Fig. 9. Simulation of a name-spaced FRU-bot in the top image, and visualization of SLAM in bottom image during the same simulation session.

IV. Evaluation

The IR encoders served their purpose to a usable level. The encoders can determine an RPM for PID and speed control, but with only 20 slots/encoder counts, the readings are most likely not entirely accurate. For the project and with the budget constraints they satisfy our needs. An experiment for future use is to 3D print a new attachment for the rotary piece with more slotted holes. However, issues to look out for in the future for testing more holes are whether the IR sensor can keep up / differentiate the smaller holes and whether the PCNT peripheral is able to keep up with counting the rising edges of the triggers.

For the IMU/MPU6050, the evaluation will fall under the same umbrella as the rotary encoders. It served the purpose of telling us the pose, but it came at the cost of drift to the point the host machine would struggle to understand/run simulations on the data being recieved. The host machine filters data to help with these issues, but it was not prepared to handle how much the data drifted from the MPU6050. As for the reason why the data was not as good as it should have been, we have one theory. When developing the driver for the MPU6050, the steps taken in reading the data and applying math to the data are similar to those in a Mechatronics tutorial [9]. With the only significant difference being the I2C driver reading data. When running the Arduino example code there is no major drift issue, which leads us to believe that the MPU6050 is to run on a microcontroller like the UNO as opposed to one such as the ESP32 deployed in the project. When checking the error value calculated between the two boards, the values from the UNO are closer to the one reported in the tutorial than the value from the ESP32. To figure out this issue, the first step that needs testing is trying complex calibration sequences as opposed to the simple ones used by both the tutorial and Linorobot. If the different calibration sequences do not work out, the next best option is to use the MPU9050. The reason for recommending the MPU9050 as opposed to the 6050 is the Espressif example tutorial for I2C uses MPU9050 and most likely supports it better than the 6050. Meaning the MPU6050 is reliable within the Arduino framework. However, switching to the 9050 is difficult due to budgeting constraints. With the 6050 a set of 5 costs about 10 dollars, while a single 9050 is about 12 dollars.

How effective was the LiDAR for our team? Its functionality met our project requirements by successfully extracting information from the LiDAR and transmitting it as messages to the host computer. We achieved mapping of the rover's surroundings at intervals of 1 second. For our specific project objectives, it fulfilled its designated tasks precisely and efficiently.

Regarding the performance of the Old PWM Driver, it efficiently regulated motor speed by managing the duty cycle. Implementing this older driver proved simpler compared to adopting the newer PWM library within Espressif.

As for the USART pins on the microcontroller, they functioned suitably for transmission and reception of data from the LiDAR. However, improved documentation would have been beneficial, as the process involved a considerable amount of trial and error to identify which USART pins were available for our purposes.

On another note, full system integration with ROS2 met some resistance due to the complexity and overhead. There was considerable experimentation with optimally setting configuration parameters for the SLAM and Navigation tool-set. Furthermore, a fully debugged name-spaced launch sequence was not fully finished by demo-day. Leading to poor outcomes with SLAM and autonomous navigation. However, overall the robots functioned well given sharp deadlines and the complexity of this project.

V. Conclusion

FRU-bot required the integration of multiple software frameworks, stacks, and open source projects. Our team developed a set of open-source packages specific to FRU-bot: firmware, hardware/sensor, navigation, simulation, visualization, and deployment packages. By demo-day, each FRU-bot was cabable of SLAM and autonomous navigation; however, time constraints, noisy inexpensive sensors, and our agent communication strategy ultimately led to poor outcomes during demonstration. Furthermore, the system complexity of multiple robots overwhelmed our central station when attempting to deploy more than one robot with active SLAM features. Overall, we were able to demonstrate what an inexpensive

mobile platform was capable of and the various powerful tools of the ROS2 framework.

VI. Discoveries and Pitfalls

During the refactoring from Arudino/C++ to Espressif/C, the Microros libraries for Arduino are surprisingly very similar, and the developers did well at keeping the name of function calls to Macros and functionality. Unfortunately, there are differences that need mentioning. In the Arduino Microros, runtime variables do not need to be initialized, but in the Espressif and C it is required before use. Here is an example where in Arduino you would only need the second line, but in C you need the two:

```
rosidl_runtime_c__String__init(odom_msg_.child_frame_id));
odom_msg_.child_frame_id =
micro_ros_string_utilities_set(odom_msg_.child_frame_id, "base_footprint");
```

Another example is the pinging of the Microros agent. In the original Linorobot's state machine states are Wait for Agent, Agent available, Agent Connected, and Agent Disconnected. We found that in the state Wait for the Agent, struggles to ping the host machine agent and gets stuck in that state. When that state is removed and goes straight for Agent Connected, the code is executed without issue. A delay is also placed in the Agent Connected so that the Watchdog timer does not trigger when the rover is idle for long periods. The last pitfall with Microros that the team discovered was the limitation with nodes.

ROS and Microros allow Nodes to have multiple publishers and subscribers, but the Microros library for ESP32 has the default max for using nodes to one node, two subscribers, and two publishers. Luckily, these are not hard-set max values and require you to go to the library's colcon.meta file to change the max value then clean/recompile the library.

An unexpected pitfall encounter was with the WiFi router used in the project. Essentially, the firewall on the router would block the microcontroller from initializing a ROS network if the controller is reset more than three times. To circumvent this issue, power cycle the router or turn off the firewall on the router. Turning off the firewall allowed for multiple resets of the board, but it still can/will get stuck with initializing the network if done too often.

Refactoring the lidar code base from C++ to C posed significant challenges, primarily due to the use of a different microcontroller with an entirely distinct framework. While working on the Pico, the capability to read a single byte at a time was feasible. However, transitioning to the Espressif framework posed a problem as it inherently reads multiple bytes, lacking the option to read singular bytes using USART. Also, identifying available USART pins on the microcontroller involved substantial trial and error. However, implementing the lidar onto the microcontroller presented numerous difficulties. The initial challenge surfaced as a watchdog warning during the first trial, triggered because the main system lacked resources to oversee its overall functionality, attributable to a while loop that didn't stop. The remedy involved incorporating a delay of 5 ticks at the end of the loop, to allow the system to check it's status.

Subsequently, encounter issues while retrieving the complete stack of data from the lidar became apparent. Although we were receiving information from the serial line, our code expected the first byte read was the start of the frame, which was consistent only when starting the LiDAR to send data. After a while, the alignment in data we were reading was inconsistent. The solution to ensure consistent data framing,was to forcefully sending the character 'b', which is used to start sending data to the microcontroller. This was called in the polling method within our code, please refere to Appendix A line 1.

Further complexity arose when attempting to publish lidar data to our local host computer. It was able to publish data in it's own project folder, however integration into our primary project file alongside other components resulted in the system to fail. The underlying issue stemmed from microROS, which imposed a limit on node creation set by a configuration file. After extensive research lasting 2-3 weeks, the solution emerged: modifying the configuration file.

VII. Appendices

APPENDIX A

Inside the Lidar Polling Method

```
error_tx = uart_write(test_str);

while(!successful_scan){
    uart_read_size = uart_read(data);

    if(uart_read_size > 0){
        sync[0] = data[0];
        sync[1] = data[1];
    }

    //First capture the start byte of a frame:
    //Once start byte captured, read remaining frame into array:
    if (sync[0] == 0xFA && sync[1] == 0xA0) {
        frame[0] = 0xFA;
        frame[1] = 0xA0;
        for (int v = 2; v <= 2520; v++) {
            frame[v] = data[v]; //copying the data from data array to frame
        }
        ready = true;
    }

    //Once frame captured, extract range/angle and convert to x/y:
    if (ready) {
        lidar_msg_->angle_increment = (2.0*M_PI/360.0);
        lidar_msg_->angle_min = 0.0;
        lidar_msg_->angle_max = 2.0 * M_PI - lidar_msg_->angle_increment;
        lidar_msg_->range_min = 0.12;
        lidar_msg_->range_max = 3.5;

        for (uint16_t i = 0; i < 2520; i = i + 42) {
            if (frame[i] == 0xFA && frame[(i + 1)] == 0xA0 + (i / 42)) {
                good_sets++;
                motor_speed += (frame[i+3] << 8) + frame[i+2]; //accumlate count for avg. time increment
                for (uint16_t j = i + 4; j < i + 40; j = j + 6)
                {

                index = 6*(i/42) + (j-4-i)/6;

                uint8_t rangeA = frame[j + 2];
                uint8_t rangeB = frame[j + 3];
                uint8_t rangeC = frame[j];
                uint8_t rangeD = frame[j+1];

                uint16_t range = (rangeB << 8) + rangeA;
                uint16_t intensity = (rangeD << 8) + rangeC;

                lidar_msg_->ranges.data[359-index] = range/1000.0;
                lidar_msg_->intensities.data[359-index] = intensity;
            }
        }
    }

    rpms = motor_speed / good_sets / 10;
    lidar_msg_->time_increment = (float)(1.0 / (rpms*6));
    lidar_msg_->scan_time = lidar_msg_->time_increment * 360;

    ready = false;
    num_of_times_it_made_it++;
    successful_scan = true;
}
else {
    sync[0] = 0;
    sync[1] = 0;
    did_not_work++;
    }

if(did_not_work >= 5 )
{
    num_reset++;
    ESP_LOGI(TAG_LIDAR, "Number of resets of lidar: %d\n Current value of conversions: %d",
```

```
            num_reset,
            num_of_times_it_made_it);
        vTaskDelay(5);
        free(data);
        return false;
    }
}

free(data);
return successful_scan;
```

APPENDIX B

FRU-bot Power Interface Board

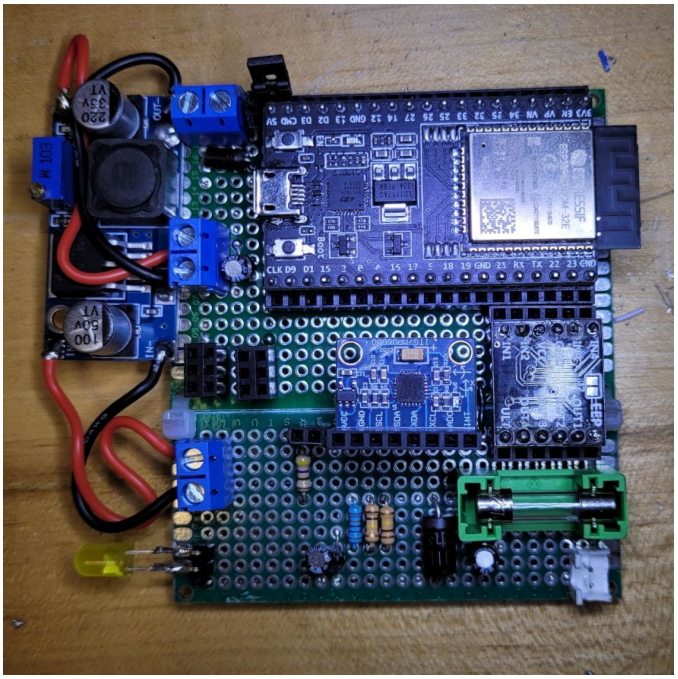

Fig. 10. Power interface board (PIB) with hardware interfaces. FRU-bot's PIB has suport for 3.3V, 5V, and 7.4V supplies. A 4A fuse and flyback diodes provide protection against over-current and reverse voltage damage. The LED light and voltage divider sensor for battery voltage monitoring work in unison to prevent battery cell damage.

APPENDIX C

Launching FRU-bot Using the Bringup Script

```
# Starts all nodes for a simulated FRU-bot instance in namespace 'FRU_bot0' with remappings of state publisher (set sim:=false for physical robot).
# Starts Robot State Publisher, Gazebo server and client, rviz, and robot localization toolbox nodes.
ros2 launch fru_bot_bringup bringup.launch.py use_rviz:=true sim:=True use_ns:=True idx:=0 remap_tf:=True
```

APPENDIX D

Multi-Agent Gazebo Simulation

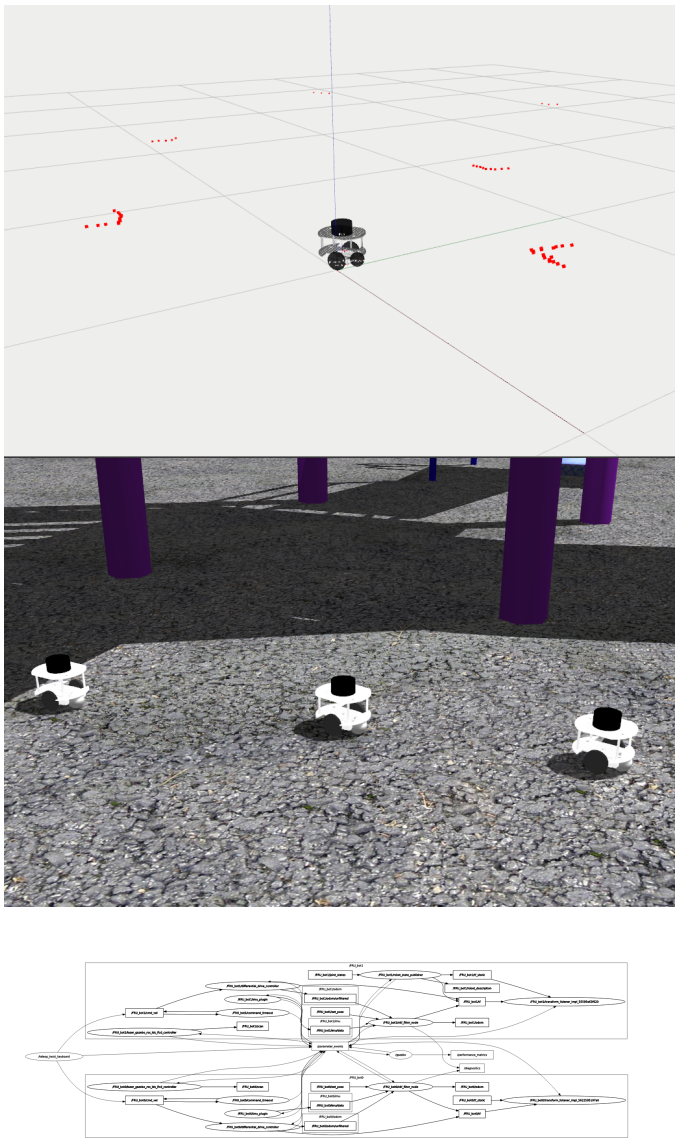

Fig. 11. Visualization of FRU_bot1 at the top, and the same view in the simulation gazebo environment at the bottom with the other two agents in the center image. The bottom image features the dependency graphs of two out of the three total agents.

REFERENCES

[1] Jack, "Implement ros2 odometry using vscode running in a web browser," 05/05/2023 2022. [Online]. Available: https://blog.hadabot.com/implement-ros2-odometry-using-vscode-in-web-browser.html

[2] T. Moore and D. Stouch, "A generalized extended kalman filter implementation for the robot operating system," in *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer, July 2014.

[3] K. Lee, W. Chung, and K. Yoo, "Kinematic parameter calibration of a car-like mobile robot to improve odometry accuracy," *Mechatronics*, vol. 20, no. 5, pp. 582–595, 2010.

[4] S. Macenski and I. Jambrecic, "Slam toolbox: Slam for the dynamic world," *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, 2021. [Online]. Available: https://doi.org/10.21105/joss.02783
[5] Linorobot, "Linorobot/linorobot2: Autonomous mobile robots (2wd, 4wd, mecanum drive)." [Online]. Available: https://github.com/linorobot/linorobot2
[6] "Wiki," Aug 2018. [Online]. Available: http://wiki.ros.org/ROS/Introduction
[7] D. S. Drew, "Multi-agent systems for search and rescue applications," *Current Robotics Reports*, vol. 2, pp. 189–200, 2021.
[8] Sep 2021. [Online]. Available: https://gazebosim.org/
[9] Chris, Terenc, C. Sinha, L. Akindele, Dejan, J. Green, A. Richards, Manuel, Martin, Feroce-Lapin, and et al., "Arduino and mpu6050 accelerometer and gyroscope tutorial," Feb 2022. [Online]. Available: https://howtomechatronics.com/tutorials/arduino/arduino-and-mpu6050-accelerometer-and-gyroscope-tutorial/
[10] Robotis-Git, "Robotis-git/turtlebot3: Ros packages for turtlebot3." [Online]. Available: https://github.com/ROBOTIS-GIT/turtlebot3
[11] S. Khaliq, S. Ahsan, and M. D. Nisar, "Multi-platform hardware in the loop (hil) simulation for decentralized swarm communication using ros and gazebo," in *2021 IEEE 22nd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE, Conference Proceedings, pp. 310–315.
[12] "esp32-wroom-32e_esp32-wroom-32ue_datasheet_en_2023," 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf