I.G.O.R.

Intelligent General Order-Fulfillment Robot

Johnathan Shentu, CSE; Josh Savard, CSE; Adam Rivelli, ME; Victor Wong, CSE; Alex Yen, CSE

***Abstract*— Delivering objects autonomously in a building is not only an interesting technical problem, but is also a way to allow for more uninterrupted time for officer workers. With I.G.O.R. we demonstrate that a significant portion of this time can be saved by replacing the delivery process with a convenient and reliable robot.**

Package delivery within a building, for humans, requires knowledge of the building topology as well as time to physically deliver the package. Our project implements a robot that autonomously delivers packages within a floor of a building. We hope that this project can be used to reduce the amount of time spent delivering packages in office settings, and to improve the productivity of employees.

I. INTRODUCTION

A. Significance

Every year, significant amounts of time are spent by highly paid individuals in the trivial matter of transporting items within offices and other business environments. While some companies have dedicated paper runners, it can be expensive to hire a person whose sole job is to deliver packages. As a result, the valuable time of crucial employees is not utilised to its fullest potential.

B. Context and Existing Products

There are many delivery robots built today, but they are designed for alternative use cases. The Marble Robot [1] is able to deliver items between two points of interest in an outdoor setting. Similarly, Starship Technologies' food delivery robots [2] also only operate outdoors. One large advantage of restricting operations to the outdoors is the availability of GPS/GNSS signals for localization.

The Savioke Relay [3] is a delivery robot meant to provide hotel room service. As this robot operates indoors, it cannot use GPS/GNSS signals and instead uses LiDAR. Both of these robots must be loaded and unloaded by a human, which is one of the challenges our robot will attempt to address.

C. Societal Impacts

Our project will reduce the time spent on delivering small packages within a building. If we deployed I.G.O.R. in an office building, we expect that it would increase the overall productivity of most employees by decreasing the amount of time spent on tasks not directly related to work. From a manager's perspective, this is an obvious benefit. From an employee's perspective, this may be seen as a benefit because their time is not wasted on trivial tasks. However, relatively minor inconveniences like delivering a file to a coworker's desk may be seen as welcome breaks during a day's work. It is more concerning that I.G.O.R. may reduce the number of people at sites where delivery of intra-office mail is a time-consuming task. But because I.G.O.R. can only deliver packages - not sort or schedule deliveries - human intervention is still very much needed.

A problem that we are unable to address is Human-Robot Interaction. That topic is on a different scope of ongoing research that will be hard to address within our SDP project. If the robot can detect an anomaly within its own system, we can have the robot stop all actions as an emergency operation, and set up hazard lights to signal and error within its own system. However, if a robot encounters an unintentional error in its system that isn't detected, it will not react to the error accordingly. Additional components, such as peripheral sensors, will allow our system to detect errors and allow for internal correction by the main system. However, this is not addressed in our project.

D. Requirements and Specifications

The overall objective of this project is to be able to deliver a package from one location to another autonomously. In order to meet this objective, the robot needs to meet the high level requirements shown in the left column of Table 1.

Table 1: Requirements and Specifications

Requirement	Specification	Value
Receive source and destination	Command-line interface	Display a map that the user can use to select a package source and destination
Path plan route to goal	Time	< 2 sec
Carry a package to destination	Speed	0.5 mph
Autonomous package unloading	Distance from selected destination	3 feet
Battery Life	Time	3+ deliveries in Marcus basement
Collision avoidance	Responsiveness	< 180 ms
Portability	Size / weight	< 4cu.ft. / < 20 lbs

E. Specification breakdown

First, a user needs to send a delivery order to the robot. To make it easy to select a pickup and drop off point, some form of graphical interface is required. We currently plan on displaying a map to the user, such that the user can select pickup and drop off points of interest and then send that data to the robot.

Next, the robot needs to be able to plan a path between two selected points of interest. Here, we've specified that the robot should be able to plan a path in less than 2 seconds. While it is

important that the robot begins its tasks quickly, it is also important to note that the robot needs to continuously re-plan its path in case it goes off course. In order to decrease error, the robot must be able to re-plan the path in less than 2 seconds to prevent significant disparity from the robot's estimate of its position and its actual position. This is accomplished with the Navigation Stack (NavStack), which plans a path trajectory for the robot within our specification of 2 seconds.

In addition, while it is important that the robot can move between pickup and drop off points quickly, it is more important that the robot doesn't harm people or infrastructure while traveling. We decided that the robot should move at about 0.5 mph to increase safety regarding collisions and improve the robot's path following accuracy due to it having reduced momentum. While 0.5 mph is slower than the average person's walking speed, it is still quick enough to fulfill deliveries in a timely manner.

We decided that the package needs to be delivered within 3 feet of the destination because when packages are delivered, it isn't imperative that the package is at the exact position; it is only important that it is left at about the correct area with some level of error tolerance, which we decided was 3 feet. We settled on 3 feet due to doorways being about 3 feet across, and we determined it was not an issue if a user specified a drop off point at the left side of a doorway, and the package was left at the right side.

The battery life was chosen because in order to have an active duty time with sporadic use; we estimate that the duty cycle between delivering and idling will be on average one to three. Thus, completing three deliveries in the Marcus basement on one charge should allow an adequate buffer to allow for recharging between deliveries. Per delivery, the robot should travel the distance of approximately Marcus 5 to the SDP lab.

Collision avoidance was chosen to be under 180 ms so that if the robot is moving at maximum velocity, it can stop in less than 10 cm, which is the maximum distance that our distance sensors can reliably detect objects at.

Portability is the requirement with the broadest range of acceptable values. Because we want our product to be easy for a human to move in case of an emergency, we based the value for the portability requirement on what an average person can easily carry.

II. Design

A. Overview

We've outlined the different subsystems necessary to implement our solution in Figure 1.

Each module is outlined in grey, which we further modularize with specific components. The core computational component within our system is the Raspberry Pi [4]. The Pi controls every component within our system – the motors, unloading mechanism, Pi Camera [5], and our custom PCB with integrated ultrasonic distance sensors; our Pi is the driving microcontroller for our system. The battery was chosen such that there was enough capacity so that the robot could make 3 or more deliveries across the basement of Marcus. Additionally the robot frame and the lifting mechanism were chosen to be able to keep the robot inside of our size and weight constraints.

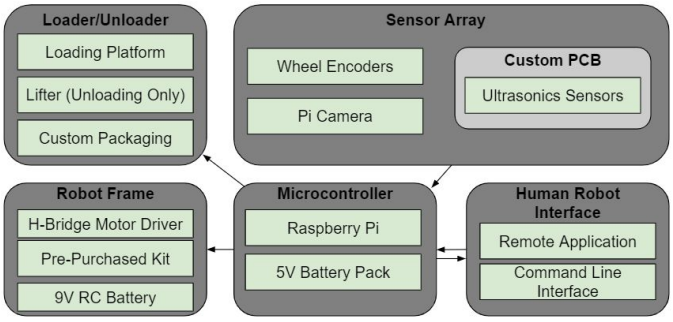

Figure 1. Block diagram describing the components used in our project.

Within our Pi, we are using a middleware called the Robot Operating System (ROS) [6]. Within ROS, I.G.O.R. uses the NavStack and AprilTag [9] packages to do the following: navigate from source to destination given its surroundings, current position, and destination position; use AprilTags as a means for global localization of the robot's position in relation to the map. For our CDR, we have demonstrated that our robot can navigate and drive from a source point to a destination point, although the navigation via NavStack and global localization via AprilTags needs further improvement. The use of NavStack and AprilTags will be discussed more in *II.C.*

We are using a pre-built mecanum robotics kit [7] as the platform for our project. We've also added an H-bridge motor driver module [8] to this kit to allow us to control the motors using the Raspberry Pi. The motors have built-in encoders that are used to calculate the robot's current position using wheel odometry. The robot's updated position is then sent to NavStack, which sends new velocity commands to the motors. New encoder values are then generated via rotation of the motor, and these encoder counts are once again used by NavStack and thereby closes the control loop.

The main focus for the remainder of the semester would have been put into incorporating navigation with AprilTags to make our package delivery more robust and accurate. Additionally, we were implementing a command line interface (CLI) to select the source and destination positions for delivery.

B. State Machine

Upon the completion of all modules, the end system will utilize them to perform the intended tasks of the robot. In order to do so, the system must perform subtasks abstracted from the modules in a specific order.

A model of our system is shown in Figure 2 in the form of

a state machine. In this model, initialization begins by checking if a map file has been preloaded into the system. This is necessary as the robot cannot navigate the floor or receive meaningful directives without the map. If successful, the system requests for a confirmation of the current position it is in.

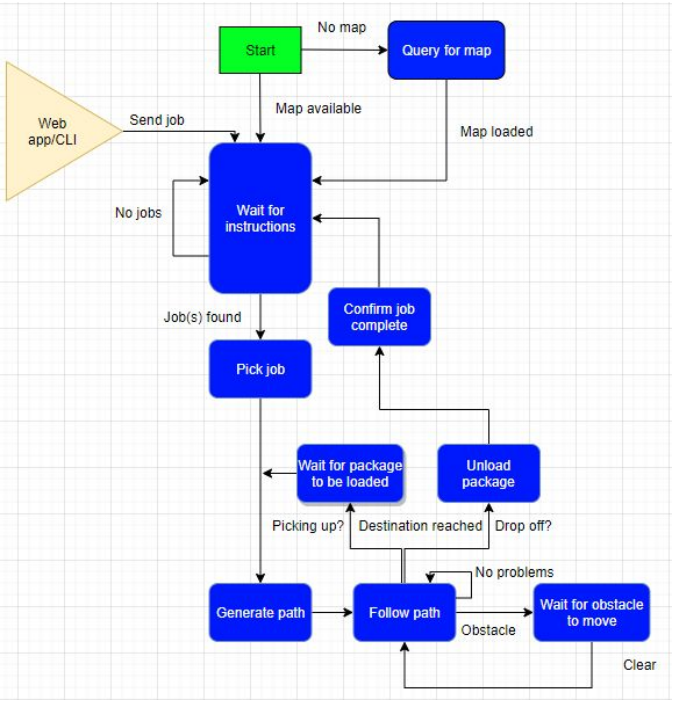

Figure 2: State machine representing the program flow of our robot.

After successful initialization, the system will begin idling and checking for job directives. These directives will be received by either a console on the Raspberry Pi or polled from a server. Job directives consist of a source and destination point, which are then transcribed into two separate source and destination points – the robot must first drive to the package location from its current position, then drive to the dropoff location to unload the package.

In order to perform this task, the system must first generate a path for the robot to follow, then command the motors to follow the path while updating its current location using a combination of odometry and visual cues (April Tags) [9]. In addition, the robot must stop to avoid obstacles detected from the ultrasonic sensors, planning around it if necessary. At the end of each destination, the robot must then align itself to receive or drop the package off at its destination.

C. Robot Operating System

ROS is a software framework and collection of libraries that facilitates internal communication and modularization within a robotic system, as well as simplifying the implementation of a number of robotics algorithms on a new physical platform. In ROS, programs are split into different nodes, each of which can send messages to topics and decide from which topics it wants to receive messages. The architecture of our ROS program is shown in Figure 7. Note that not every node and topic is shown in this diagram; if they were, this diagram would become over-complicated. For example, NavStack uses several nodes and topics internally that aren't directly relevant to our project, such as the motor driver node, so NavStack is simply represented by a single node. The core of our software architecture is the NavStack and the motors – everything else can be interpreted as a data source for the NavStack.

The NavStack is an official ROS repository that sends commands to a robot's motors, provided that it receives the required data to perform its role. At a minimum, the NavStack needs to know the following: (1) what its surroundings are like, (2) where it currently is, and (3) where it should go. The rest of the software we have written is built around providing the NavStack with this information.

There are three ways to accomplish task (1): outfit the robot with the sensors needed to figure out what its environment is like, tell the robot ahead of time about its environment, or a combination of the two. For this project, we limited ourselves to primarily telling the robot what its environment is like ahead of time. To do this, we pre-loaded the robot with a hand-drawn map of its environment, indicating which areas in its environment are freespace and which areas are obstacles. We also pre-loaded the robot with the position of several AprilTags in its environment that it could later use as landmarks.

For task (2), we used two sources of data about the robot's current position: how far each of its wheels has turned (wheel odometry), and the robot's position relative to the AprilTags we placed in the world (visual odometry).

Wheel odometry works by sensing the amount that each wheel rotated during a timestep using encoders, which then uses that information to calculate the direction and distance the robot traveled in during the last timestep. For example, if all of the wheels rotated 90 degrees forwards between the previous and the current timestep and each wheel had a circumference of 4 inches, then one can conclude that the robot moved forwards by one inch. Because our robot uses mecanum wheels, the equations governing our wheel odometry are slightly more complicated than that. These equations are shown in Equation 1.

$$\begin{aligned} \Delta x &= \frac{1}{4}(d_1 + d_2 + d_3 + d_4) \\ \Delta y &= \frac{1}{4}(-d_1 + d_2 + d_3 - d_4) * tan(\alpha) \\ \Delta\theta &= (-d_1 + d_2 - d_3 + d_4) * \beta \end{aligned} \quad (1)$$

Equation 1. The equations governing the wheel odometry of I.G.O.R. Δx, Δy, $\Delta\theta$ represent the change in the robots position forward, to the left, and rotationally, d_i represent the tangential distance that each wheel rotated in the last timestep for the front-left, front-right, rear-left, and rear-right wheels respectively, and α and β are parameters tuned manually depending on the robot and its environment's physical features.

While wheel odometry can be used for localization in many robots, solely relying on this information will result in movement inaccuracies over long periods of time. If the robot runs over a slippery patch of ground, gets pushed, or accidentally overshoots its position, the robot's positional awareness will be unreliable if wheel odometry is its only source of position information.

For this reason, we implemented visual odometry using AprilTags. AprilTags are small grids of pixels, shown in Figure 3. Given the characteristics of a camera (e.g., how much it distorts an image), a software library can be used to calculate the camera's position and orientation in space relative to an AprilTag. Since we already know the position of the camera relative to the robot and the position of the AprilTags relative to the environment, we can use these two relative positions to calculate the robot's actual position based on this data, and correct the errors accumulated in the wheel odometry's estimate of the robot's position. Using a combination of wheel and visual odometry, we can accomplish task (2), and provide the NavStack with an estimate of the robot's current position.

Figure 3. An image of an AprilTag.

Finally, we need to tell the robot where to go (task (3)). To do this, we use the orchestrator node from Figure 7. The orchestrator is largely responsible for implementing the behaviors described by the state machine in Figure 2. The orchestrator is where our robot behaves as a delivery robot, not just a robot that navigates autonomously. The orchestrator node accepts a few pieces of user input and then directs the NavStack where to go based on this input. The orchestrator needs the following as input: the package pickup coordinates, whether or not the package has been placed on the robot yet, and the package dropoff coordinates. After the user enters the pickup coordinates, the orchestrator asks the NavStack to go to the pickup location. Next, the orchestrator waits until the user indicates that the package has been loaded, and then passes the destination coordinates on to the NavStack. After reaching the destination, the orchestrator communicates with the lifter to drop off the package, and then it resumes its idle behavior.

Given that tasks (1), (2), and (3) are accomplished, the NavStack can generate its output: a desired robot velocity. After generating this velocity, the motor node converts this into a set of motor voltages, and sends these to the motors.

III. The Product

A. *Product Overview*

In figures 4 and 5, we show our product design sketch and physical implementation respectively; figure 4 shows the top-down view sketch of our robot.

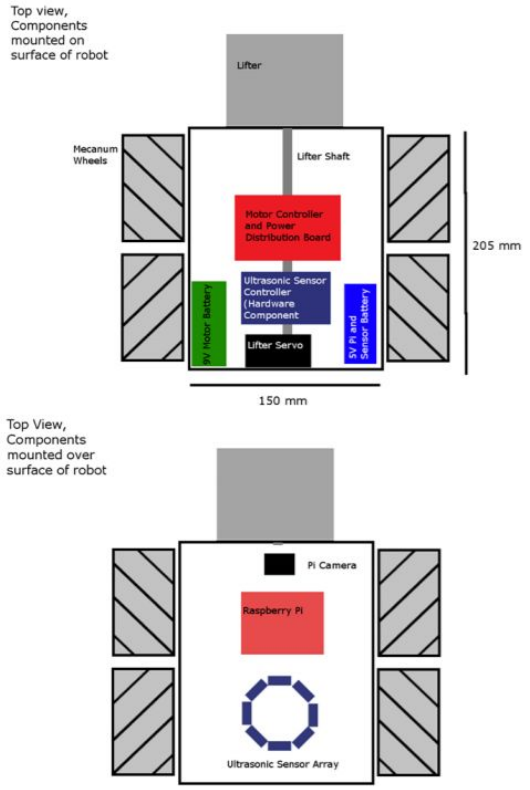

Figure 4. Product Sketch

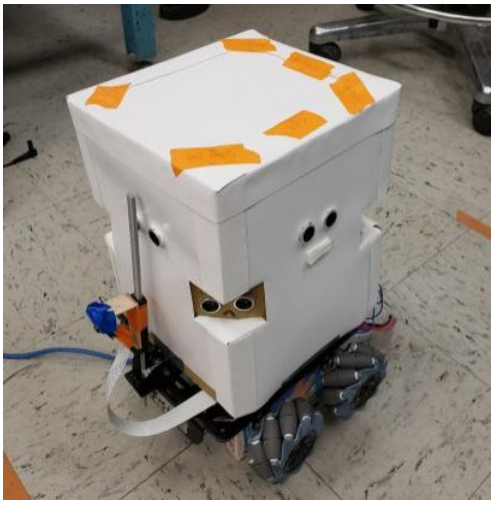

Figure 5. Product Implementation

Figure 6 shows an example of I.G.O.R. in operation, which navigates from the beginning destination to the target

destination while avoiding obstacles, picking up a package, and using AprilTags for global localization. Figure 7 shows our software implementation of our robot system.

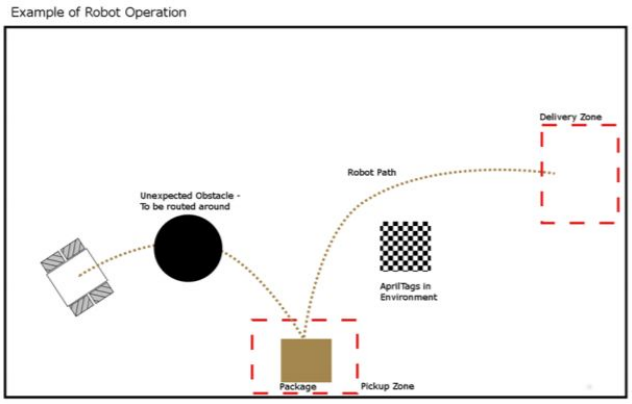

Figure 6. Robot Operation

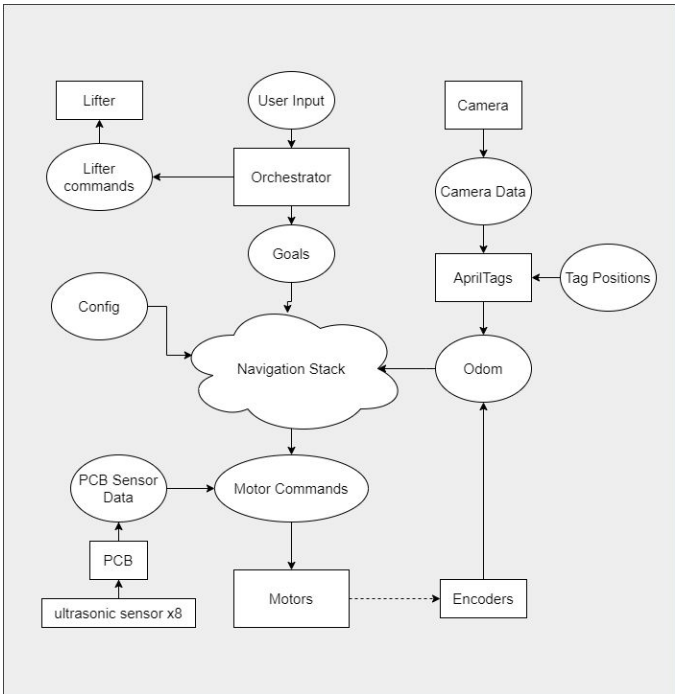

Figure 7. I.G.O.R. Software Architecture.

B. *Electronic Hardware Component*

Before MDR, the PCB was emulated using an Arduino Uno to run four ultrasonic distance sensors. In the transition between Arduino to PCB, a model was established with four ultrasonics on a breadboard. This was then scaled up to include hardware to support eight ultrasonics, and to regulate voltage to the microcontroller. Subsequently, this was created as a schematic using Altium designer and sent to OSH Park for fabrication. The majority of components were surface mounted to the board after being tested. Problems such as shorts between through-hole mounted components were diagnosed with a meter and an oscilloscope.

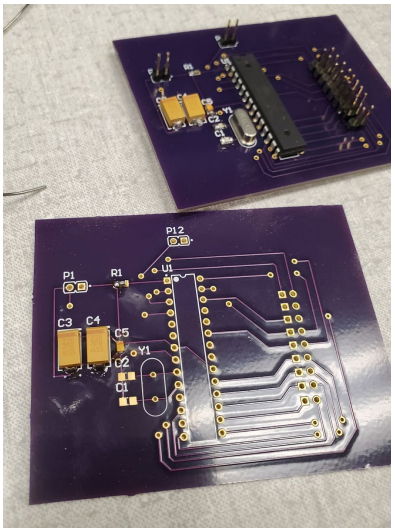

Figure 8. Populated and Unpopulated PCBs

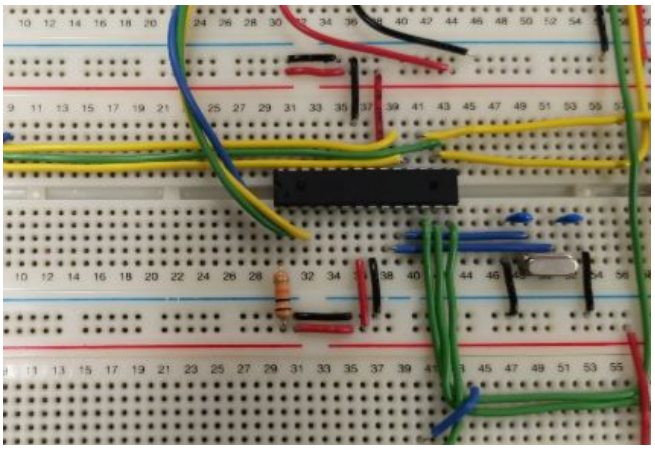

Figure 9. PCB Development Breadboard

C. *Product Functionality*

Table 2 shows a summary of the CDR deliverables we were able to complete. Overall, we did not reach a few major goals, and still had a fair bit of work required to be completed between CDR and FPR.

D. *Product Performance*

In the implementation at the time of CDR, the robot was able to receive command line instructions specifying a start and goal position, and autonomously navigate between the two locations. It should be noted that navigation was not integrated with real time updates from AprilTags so there was no method of correction in absolute position, resulting in disparities of up to 6 inches after traveling 1.5 meters while changing direction three times with a package loaded. Additionally the battery was able to consistently surpass its three delivery requirement with the ability to turn the wheels for over 10 mins. In addition, we initially planned for the robot to pick up a package autonomously, which would have been accomplished via AprilTags through self alignment. However, we realized that this task was a lot more complicated than we had

imagined, and we decided to omit this part of our project in our final design. We also implemented a command line interface that could transmit commands to the robot wirelessly.

Table 2. A table summarizing which CDR goals we were able to complete on time.

Deliverable	Met?
Update global position with April Tags	50%. The robot can determine its position relative to an AprilTag, but it does not update its position relative to the world based on that information.
Path Planning	Yes. The robot can plan and follow a path.
Receive Directives	Yes. The robot receives directives through a CLI interface
Detect and plan around obstacle with Ultrasonic Distance Sensors	50%. The robot detects obstacles and stops, but does not plan a path around the obstacle

IV. Conclusion

A. *Current State of the Project*

Physically, our robot has a mobile base, a prototype lifting mechanism, and a prototype enclosure, as well as all of the sensors and actuators required to sense and move within its environment. A photo of the current state of our robot is shown in Figure 5.

Currently, the robot can receive directives from a CLI interface, and then plan and follow a path from its current position, to the pickup point, and to the dropoff point. However, it performs this using only wheel odometry for position feedback, with no correction for the accumulation of error. Path planning and execution are completed using ROS and the NavStack provided by ROS.

A camera is mounted on the robot, and it can sense the position of an AprilTag relative to the camera. However, this position is not used by the robot to update its position relative to the world.

We have designed a PCB and integrated it into our project. The PCB operates the ultrasonic distance sensors, compiles the data received from them, and relays this data to the Raspberry Pi. A ROS node on the Raspberry Pi then processes this data, and decides whether or not to stop based on the sensor data.

B. *Intended Future Implementation*

Due to the abrupt conditions imposed by the COVID-19 pandemic, our project was unable to see its completion. All of the other sections of this report only deal with what we were able to complete up until CDR, and this section will discuss what we would have focused on between CDR and FPR if this project were to be continued.

Three major tasks remained to be completed between CDR and FPR: (1) implementing visual odometry using AprilTags, (2) testing and tuning the complete navigation system, and (3) making our project clean and presentable for demo day.

At CDR, the robot's navigation was not very accurate. Wheel odometry needs to be experimentally calibrated for new robots, so there is a lot of uncertainty intrinsic in it. As our robot travelled farther distances, the accuracy of its position estimate continuously decreased. Therefore, our first priority would have been to implement visual odometry using AprilTags in order to improve the long-term accuracy of the robot's position estimates.

Next, we would have focused on testing and tuning the navigation. There were many variables in our system that needed to be hand-tuned, such as: the maximum velocity and acceleration of the robot, the amount of uncertainty present in our wheel and visual odometry readings, the rate at which the odometry loops should be run at, and many other parameters related to NavStack. Most of these values need to be determined experimentally, and we would have needed to spend a lot of time on determining the best values for these parameters for our specific application.

Next, we would have worked on cleaning up our project for FPR and demo day, physically as well as in software. We would have remade the lifter to look more presentable, mounted the camera more cleanly to the enclosure, and constructed a more robust enclosure overall. Additionally, we would have converted the CLI to a GUI in order to make the project easier to demonstrate at demo day.

Acknowledgment

We'd like to thank our advisor, Prof. Ciesielski for providing continual guidance and feedback to our team throughout the semester. We'd also like to thank our evaluators, Prof. Stephen Frasier and Prof. Mario Parente, for providing constructive feedback and helping us constrain the scope of our project.

References

[1] "Autonomously Moving Things into the Future," *marble.io* [Online]. Available: https://www.marble.io/ [Accessed December 18, 2019].

[2] "We are a company building a network of robots ready to serve you anytime, anywhere," *starship.xyz* [Online]. Available: https://www.starship.xyz/ [Accessed December 19, 2019].

[3] "Meet Relay Autonomous, Secure Delivery in Dynamic Public Spaces," *savioke.com*, [Online]. Available: https://www.savioke.com/ [Accessed December 19, 2019].

[4] "Buy a Raspberry Pi 3 Model B+ – Raspberry Pi." *Buy a Raspberry Pi 3 Model B+ – Raspberry Pi*, www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/.

[5] "Buy a Camera Module V2 – Raspberry Pi." *Buy a Camera Module V2 – Raspberry Pi*, www.raspberrypi.org/products/camera-module-v2/.
[6] "About ROS." *ROS.org*, www.ros.org/about-ros/.
[7] "Mars Explorer Mecanum Wheel Robotic Kit (Arduino Mega2560)-Lesson1 Assembling the Car." *Mars Explorer Mecanum Wheel Robotic Kit (Arduino Mega2560)-Lesson1 Assembling the Car " Osoyoo.com*, 13 Nov. 2019, osoyoo.com/2019/11/13/mecanum-omni-wheel-robotic-kit-v1-for-arduino-mega2560-lesson-14/.
[8] #600449, Member. "Rover 5 Motor Driver Board." *ROB-11593 - SparkFun Electronics*, www.sparkfun.com/products/retired/11593.
[9] E. Olson, "AprilTag: A robust and flexible visual fiducial system," 2011 IEEE International Conference on Robotics and Automation, 2011.
[10] "A* Search Algorithm." *GeeksforGeeks*, GeeksforGeeks, 7 Sept. 2018, www.geeksforgeeks.org/a-search-algorithm/.

APPENDIX

A. *Design Alternatives*

During the design process of our project, we considered several alternative implementations for our chassis, our path planning algorithm, and our localization algorithm.

We had to decide whether to design and manufacture the chassis ourselves, or to buy a prefabricated robot chassis. We eventually decided to buy a prefabricated chassis. Although we could have designed a more application-specific chassis ourselves, we decided that the chassis was not the core component of our project, so the time that we would have spent designing a chassis would be better spent on other components.

For our path planning algorithm, we had to decide whether to write our own implementation, or to use an existing path planning library. Up until MDR, we had decided to write our own path planning implementation because we thought that it would be easier to make our own simple implementation, rather than to familiarize ourselves with the extensive ROS NavStack ecosystem. However, we eventually decided to use NavStack anyway. Although using NavStack introduced a lot of complexity into our code, it allowed us to worry about the problems specific to our robot, rather than reimplementing several algorithms that had already been implemented by other roboticists.

A final important design decision that we made was how to implement localization on our robot. We considered two initial choices: Simultaneous Localization and Mapping (SLAM) or visual odometry. There are several tradeoffs between these approaches, including their relative material costs, ease of implementation, computational costs, and accuracy.

In terms of sensor cost, SLAM is far more expensive than visual odometry. SLAM requires a LiDAR sensor, the cheapest of which cost around $100, while visual odometry can be implemented with a Raspberry Pi camera, which costs only about $10. Additionally, visual odometry with AprilTags can be run on a Raspberry Pi, while most people opt to run SLAM on a more powerful, more expensive computer due to its increased computational needs. However, SLAM is more commonly implemented by users of ROS, so there may have been more support for SLAM than there was for visual odometry. Additionally, we expect that SLAM would have produced more accurate estimates of our robot's position than we were able to achieve with visual odometry.

In the end, we decided to use visual odometry over SLAM. We weren't comfortable with how much of our budget SLAM would use, and we were less comfortable with its underlying technology, and were not confident that we would be able to successfully implement a robot running SLAM.

B. *Technical Standards*

Our project does not use IEEE hardware standards, but we use software standards for technical reliability and soundness. We use the IEEE 829 standard, which is the standard for software testing. Within our project, we planned to test moving the robot first, then implement path planning from two points of interest, and then incorporate AprilTags for global localization. We first tested the movement of the robot to ensure that the robot moved the correct direction, which was important because our robot uses omni-motor wheels. Next, we moved onto path planning, which incorporated the use of NavStack into our system. However, while NavStack was working properly for our project, we did not finish refining this component of our project. Instead, we moved on to AprilTags to incorporate global information with NavStack. However, before we could finish testing AprilTag incorporation with NavStack, our project was cut short due to COVID-19.

C. *Testing Methods*

We ran several tests in order to verify the performance of several individual components of the system. Notably, we tested the accuracy of our odometry measurements, and the accuracy of the AprilTag detection measurements.

The testing and verification of these two measurements was very important to the overall performance of our robot because these were the two components that allowed the robot to keep track of its current position.

In order to test the accuracy of its odometry, we drove the robot for varying periods of time and in varying directions (forwards, backwards, sideways, and diagonally), measured how far it actually travelled, and compared this to the distance that it calculated it had travelled. From these measurements, we calculated a correction factor that we used to correct its odometry estimates in the future. This is important because odometry calculations are only approximate; the distance that the robot actually moves is dependent on how much the wheels slip, which is dependent on the weight of the robot, the surface that the robot is driving on, and how worn the robot's wheels are.

To test AprilTag detection, we mounted the camera, and then measured the distance to several points in front of the camera where we then placed the AprilTags. From this, we could determine the accuracy of the AprilTag detection, and therefore the long-term accuracy of our localization algorithm.

During this test, we also determined how close the AprilTags needed to be to the camera in order for them to be detected, which helped us decide how close the tags needed to be to each other in the robot's environment.

When performing verification of the PCB, hardware analysis tools, such as a multimeter and an oscilloscope, were used to initially confirm that all contact traces were properly linked. Then, once all components were mounted, we loaded the C code onto the microcontroller to find that it did not function. After testing, we used an oscilloscope to determine that the crystal oscillator on our PCB was not functioning. Once the crystal oscillator was replaced, the board functioned properly, although a few ultrasonic distance sensors were malfunctioning. We used a scope to verify that two of the ultrasonics were broken, and they were replaced.

D. *Team Organization*

On our team, each member made strong individual technical contributions to the project, and several members stepped up to take on a leadership role on the team.

Throughout the course of the project, Alex ensured that the team was on track to meet the formal requirements and deadlines required by SDP. He took charge on scheduling our design reviews with our evaluators, as well as ensuring that the rest of the team was focused on completing the deliverables that we promised to our evaluators.

In addition to providing valuable individual contributions, Victor was also skilled at helping other members of the team make the best use of their lab time. When working with other team members, Victor did a good job of encouraging other team members to organize their thoughts, work through their technical challenges methodically, and therefore improve the overall productivity of the team.

Josh took the lead on two major components of our project: our homemade implementation of A^* path planning, and the design and programming of our PCB. Joshua was an integral part of the technical sides of both of these components, but he also helped to organize the efforts of other team members on these components as well, and he ensured that these components were completed quickly and smoothly.

While Johnathan did not often take on a leadership role within our team, he did frequently take the initiative to lay the groundwork for new components of the project. Johnathan researched and created prototype implementations of the AprilTag detection software, and the orchestrator ROS node that coalesced our entire project.

Despite being the mechanical engineering major on our team, Adam is also a computer science major, and played a fundamental role in our product implementation. When Alex, Victor, and Johnathan were struggling with NavStack and AprilTag incorporation into our robot system, Adam took initiative and propelled the team towards project completion. Overall, this project would have never progressed to its current state without Adam's technical contributions.