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 problem, but also a way to allow 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. I.G.O.R. has been equipped with a forklift-styled lifter to pick up custom designed packages.**

I. INTRODUCTION

A. *Significance*

Package delivery within a building, for humans, requires knowledge of the building topology as well as time to physically deliver the package. The time and knowledge needed to perform these deliveries can be replaced, at least in part, by a robot. 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.

B. *Context and Existing Products*

There are many delivery robots built today, but most of them are designed for alternative use cases. The Marble Robot is able to deliver items from two points of interest in an outdoor setting [1]. Similarly, Starship Technologies' delivery robots [2] exists as a food delivery robot, which is restricted to outdoor navigation. One large advantage of restricting operations to 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 implemented IGOR 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, allowing them to focus on their 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 IGOR may reduce the number of people at sites where delivery of intra-office mail is a time-consuming task. But because IGOR can only deliver packages - not sort or schedule deliveries - human intervention is still very much needed.

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	Graphical user 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
Collision avoidance	Responsiveness	< 180 ms
Portability	Size / weight	< 4cu.ft. / < 20 lbs

E. *Specification breakdown*

First, a user needs to give the robot an order. 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 where they can click to place pickup and drop off points, and then send that data to the robot.

Next, the robot needs to be able to plan a path between goals. 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 the robot goes off course. In order to decrease error, the robot must be able to re-plan the path in under 2 seconds to prevent significant disparity from robot's estimate of its position and its actual position.

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 cause any damage to people or infrastructure while traveling. We decided that the robot should move at about 0.5 mph, in order to increase safety, and decrease error in the robot's movement by decreasing the robot's inertia. 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 often imperative that the package is in the exact correct position; it is only important that it left at about the correct spot. We settled on 3 feet because doorways are about 3 feet across, and we figured it's 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 be able to have an active duty time with sporadic use, we estimate that the duty cycle between delivery and idle will be on average 1 to 3, so the 3+ delivery should be an adequate buffer to allow for recharging between deliveries.

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 broadest. It must be able to be picked up by a human, so our specifications are based around what an average person can easily carry.

II. Design

A. Overview

In our project, we outline the different modules necessary to implement our problem, shown in Figure 1.

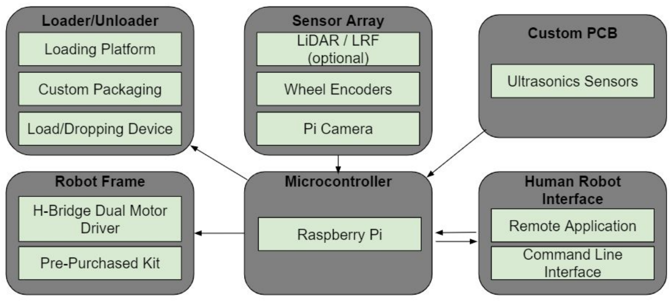


Figure 1. A block diagram describing the components used in our project.

Each module is outlined in grey, which we further modularize with specific components. The overarching component within our system is the Raspberry Pi. The Pi controls every component within our system - the motors, loading and unloading mechanism, Pi Camera, and our custom PCB with integrated ultrasonic distance sensors; our Pi is the driving microcontroller for our system.

Within our Pi, we are using a middleware called Robot Operating System (ROS), which will be discussed more in part D in the Design section. In ROS, we run four nodes that allow for the following: loading in a map of its current environment, planning a path from source and destination points on the map, translating the path positions to velocities, sending velocity commands to motors, and updating its position via odometry. For our MDR, we have demonstrated that our robot can navigate and drive from a source point to a destination point.

To purely drive the motors, the only modules we are using are the Robot Frame module, and the Microcontroller module. We have discussed the purposes of the Microcontroller module and will now discuss the Robot Frame module in tandem with the motor encoders.

For our project, we are using a pre-built mecanum robotics kit. Since this kit consists only of the robot frame, motors, and encoders, we use an H bridge to drive the motors – this allows our motors to drive forwards and backwards, which is necessary for our robot to drive omnidirectionally. Since the motors have built in encoders, we can use encoder information to update the robot's position relative to the map through odometry. This in turn is updated in our path planning function and relative position, which in turn sends new velocity commands to the motors, allowing the robot to drive from source to destination.

The remaining modules that remain unimplemented are the Loading/Unloading module, the Custom PCB module, and the Human-Robot Interface module. Starting with the loading and unloading mechanism, our team has made this optional within our project goals; there are many issues and challenges with package alignment that could be time consuming for our project. As our main project objective is to deliver packages from a source point to a destination point, the loading and unloading mechanism is a project component that we can make a stretch goal. That being said, while we have made this optional within our project, we still anticipate completing this module. Many existing delivery robots require human interaction to load and unload packages. However, autonomous loading and unloading is a feature that will distinguish our work from others. As a result, our team still intend to implement loading an unloading, even though we have made it optional.

The Custom PCB module is strictly used for object detection with our ultrasonic distance sensors. This will be used to detect obstacles along the planned path and react accordingly. Currently, we have tested and written code to read the distance between the sensor and a detected object, but we have not yet integrated the sensors with our system. The integration of these sensors will require another node in ROS that will override commands to run the motors. This is shown by the Custom PCB module pointing to the Microcontroller module in Figure 1.

Lastly, we intend to implement a remote application that will allow users to interact with the robot. This will be done through an initial command line interface with a stretch goal of a graphical user interface. This will directly interact with the Microcontroller module, such that it will give commands to the Pi to make a delivery. This will be touched more upon in section B, which is the State Machine.

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 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.

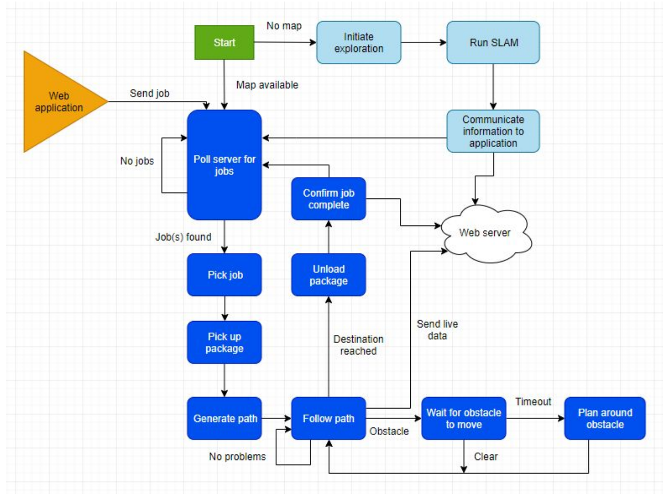


Figure 2: A state machine representing the program flow of our robot, including nodes for our stretch goals.

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 in turn are 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). 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 to its destination.

C. *Robot Operating System*

Figure 3 shows the current nodes that are used within ROS to move our robot from one destination to another. ROS is a framework that allows internal communication and modularization within a system. Each node is able to send and expect specific messages that can be handled by the node accordingly.

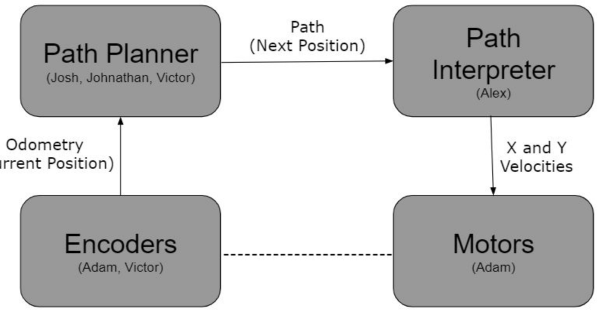
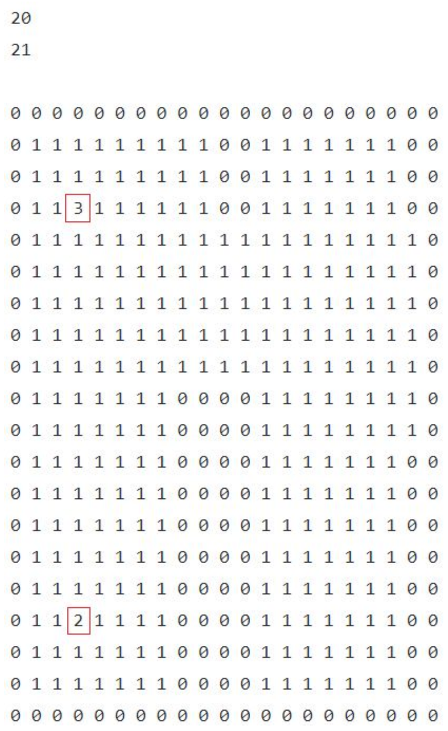


Figure 3. I.G.O.R. ROS Nodes

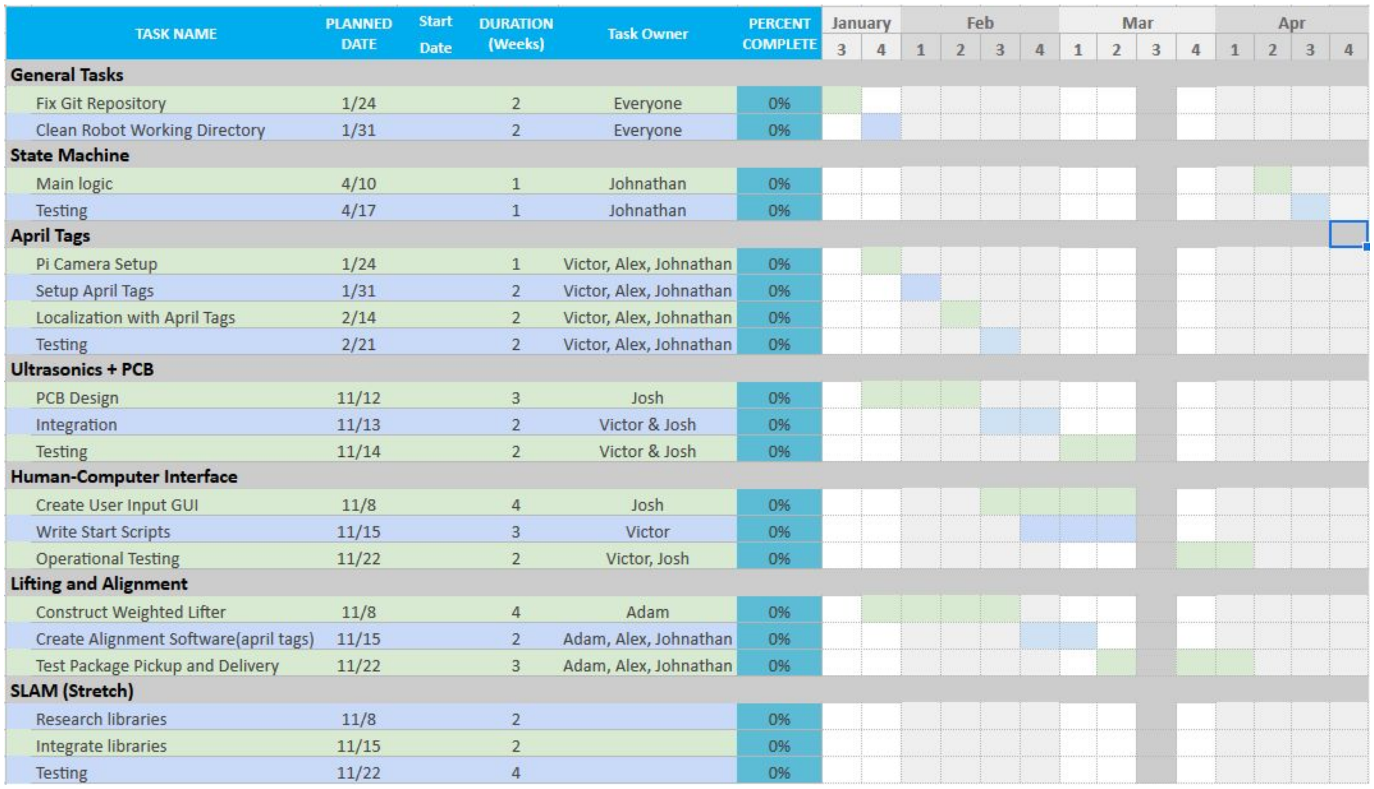


```
20

21

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0
0 1 1 3 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 0
0 1 1 2 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 4. An example of a map that we are currently using to represent the robot's environment and directives. 0's represent walls, 1's represent free space, 2 represents the robot's start position, and 3 represents the robot's goal.

Starting with the Path Planner node, we first import a map as a text file, an example of which is shown in Figure 4. In our map, we define the areas that our robot can traverse as a "1" walls or obstacles as a "0." The digits annotated in red are the starting place, "2," and the destination, "3." This map just tests our robot moving straight since "2" and "3" are in the same column. However, for our MDR demo, we used a larger map that allows our robot to travel in "C" shaped path; this map is too large to include in this report.

After the map is imported, the Path Planner node determines a path from the starting point to end point via the A* search algorithm. This node then sends a message containing an array of positions to the Path Interpreter node. The Path Interpreter node then computes the next move it needs to make by determining the vector from its current position to the next position; this vector is then converted to a linear velocity, which is sent as a message to the Motors node.



TASK NAME	PLANNED DATE	Start Date	DURATION (Weeks)	Task Owner	PERCENT COMPLETE	January		Feb				Mar				Apr			
						3	4	1	2	3	4	1	2	3	4	1	2	3	4
General Tasks																			
Fix Git Repository	1/24		2	Everyone	0%	🟩													
Clean Robot Working Directory	1/31		2	Everyone	0%		🟦												
State Machine																			
Main logic	4/10		1	Johnathan	0%												🟩		
Testing	4/17		1	Johnathan	0%													🟦	
April Tags																			
Pi Camera Setup	1/24		1	Victor, Alex, Johnathan	0%		🟩												
Setup April Tags	1/31		2	Victor, Alex, Johnathan	0%			🟦											
Localization with April Tags	2/14		2	Victor, Alex, Johnathan	0%				🟩										
Testing	2/21		2	Victor, Alex, Johnathan	0%					🟦									
Ultrasonics + PCB																			
PCB Design	11/12		3	Josh	0%		🟩	🟩	🟩										
Integration	11/13		2	Victor & Josh	0%					🟦	🟦								
Testing	11/14		2	Victor & Josh	0%							🟩	🟩						
Human-Computer Interface																			
Create User Input GUI	11/8		4	Josh	0%					🟩	🟩	🟩	🟩						
Write Start Scripts	11/15		3	Victor	0%						🟦	🟦	🟦						
Operational Testing	11/22		2	Victor, Josh	0%										🟩	🟩			
Lifting and Alignment																			
Construct Weighted Lifter	11/8		4	Adam	0%		🟩	🟩	🟩	🟩									
Create Alignment Software(april tags)	11/15		2	Adam, Alex, Johnathan	0%						🟦	🟦							
Test Package Pickup and Delivery	11/22		3	Adam, Alex, Johnathan	0%								🟩		🟩	🟩			
SLAM (Stretch)																			
Research libraries	11/8		2		0%														
Integrate libraries	11/15		2		0%														
Testing	11/22		4		0%														

Figure 5. A Gantt chart detailing the work we need to complete next semester, and a division of this work between our members.

The Motor node then receives those velocities and runs the motors, allow the robot to drive.

Since our motors have built in encoders, we are able to use the encoders for odometry – that is, we are able to update the robot's position relative to the imported map. This is shown as a dotted line in Figure 3 – while the Motor node is not sending messages specifically to the Encoder node, the two nodes are interlinked. As a result, by updating its own position in relation to the map, our robot recalculates its path from its given position to the end point – the cycle of driving the robot towards the end goal then continues as we have described above.

III. Project Management

Table 2 shows a summary of which MDR deliverables we were able to complete. Overall, we were fairly successful with meeting our goals for this semester. However, we could have been more organized overall. Appendix C gives some insight into how we organized ourselves as a team.

A Gantt chart has been included in Figure 5 in order to show how we've planned our project for next semester, and to show all the additional major components we still need to implement.

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

Deliverable	**Met?**
Load Package	Yes. The robot can lift a package.
Path Planning	Yes. The robot can plan and follow a path.
Receive Directives	Almost. The robot can receive directives through hardcoded values, but not through a CLI.
Ultrasonic detect collision	Yes. Ultrasonics detect when the robot is close to a wall or obstacle.

IV. Conclusion

A. *Current State of the Project*

Currently, our robot can plan a path between two points, with odometry as the only feedback it receives regarding its position. Additionally, we have a prototype lifting mechanism that can raise and lower packages. A photo of the current state of our robot is shown in Figure 6.

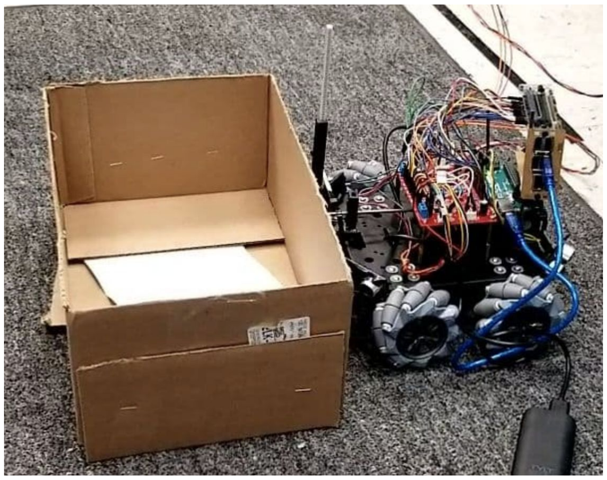


Figure 6. The current state of our robot.

Our robot currently receives its goal and start positions through hard-coded coordinates, and updates its current position only through odometry. The path planning and execution are currently performed using a simple implementation that we wrote ourselves.

We have ultrasonic range sensors connected to our robot that let us know whether an obstacle is near the robot, but we don't use that information to plan paths around unexpected obstacles. Data from these sensors are currently read and processed by an Arduino microcontroller.

Our current lifting mechanism is flexible, causing it to have a hard time lifting packages high enough to clear the ground.

B. *Next Steps*

Figure 5 shows a Gantt chart, which provides an overview of the work we must complete between now and demo day in order to have a complete product at that point.

Next semester, we need to develop a GUI interface that users can use to specify package pickup and dropoff locations.

We also need to implement localization using AprilTags [4]. AprilTags are small grids of pixels, shown in Figure 7. Given the characteristics of a camera, a software library can be used to calculate the camera's position and orientation in space relative to an AprilTag. As the robot moves, odometry inevitably accumulates error. So, the farther the robot has moved from its starting position, the more inaccurate its estimate of its current position will be. To fix this, we will put AprilTags in the environment with known positions, and then we will calculate the robot's position relative to those tags, and therefore the robot's position in a building.

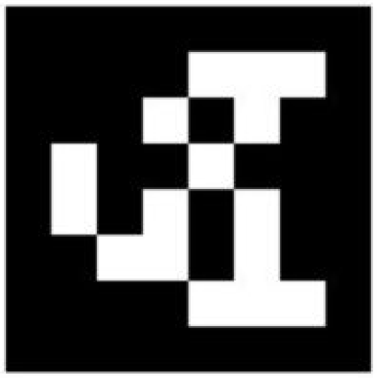


Figure 7. An image of an AprilTag.

We also need to design and print the hardware component of our project. Currently, we are using an Arduino to obtain and process data from the ultrasonic sensors related to local obstacles. However, Arduinos are not allowed in the final iteration of our project, so we will need to replace it with a custom designed PCB before FPR.

Finally, we need to improve the stiffness of our lifting mechanism. Right now it is too flexible, so it cannot reliably keep a package off the ground. We need to revise its design and implementation so it does not flex as much under load, keeping the package clear of the ground.

After these minimum requirements are completed, we will begin to focus on our stretch goals. First, we will focus on loading packages without human intervention. If time permits, we also hope to implement Simultaneous Localization and Mapping (SLAM) so our robot does not need to be loaded with a handmade map of its environment to be able to function. SLAM is an algorithm that allows a robot to build up a representation of its environment on its own, and determine its position within that environment at the same time [5].

Acknowledgment

We'd like to thank our advisor, Prof. Ciesielski for providing continual guidance and feedback to our team throughout the semester, and for helping us grow as engineers. We'd also like to thank our evaluators, Prof. Stephen Frasier and Prof. Mario Parente for providing constructive feedback and helping us to 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] E. Olson, "AprilTag: A robust and flexible visual fiducial system," 2011 IEEE International Conference on Robotics and Automation, 2011.
[5] H. Durrant-Whyte and T. Bailey, "Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms," p. 9.

Appendix

A. *Design Alternatives*

This semester, there were two major decisions that we made regarding our design.

One major decision we made was in software – we decided to write our own path-planning software rather than to use ROS's navigation software. ROS provides a series of nodes and libraries, collectively known as the "navigation stack," that robots can use to plan and follow a path between given positions. The navigation stack accepts the following as input:

- A map of the robot's environment
- The geometry of the robot, including which sensors it has available
- A goal position that the robot needs to plan a path to, which can be periodically updated
- The robot's position, which will be continually updated

Given these inputs, the navigation stack can send velocity commands to the robot's motor interface. The navigation stack is very feature-rich; it has the ability to path around local obstacles not included in the map, recover from faults, and it easily integrates new sensor information into its path planning.

However, we decided to not use the navigation stack. Among us, only one of us had experience with ROS at all, and none of us had ever used the navigation stack before. So, integrating it into our project was a daunting task, and we were worried that it would take a long time to get it working. On the other hand, we had all written A* before, and therefore thought that writing all of this ourselves wouldn't be a difficult task.

This turned out to be a mistake – we spent a lot of time writing and debugging our path planning code, and ended up with a product that is both not very feature rich and difficult to extend. Since MDR, we have briefly attempted to implement path planning using the navigation stack. Because we all now have far more experience working with ROS, this process went relatively smoothly. Next semester, we plan to completely switch to the navigation stack for path planning.

We also decided to use mecanum wheels over traditional wheels for our robot. Robots with mecanum wheels can move in any direction - even directly sideways - unlike traditional wheels. However, mecanum wheels require more traction than traditional wheels, so they are more limited in the types of terrain that they can traverse.

This decision was partially influenced by our decision to write our own path planning code. To control a tank drive robot, we would need to take into consideration the constraints on which directions the robot can move in. However, there are no constraints on how a mecanum drive robot can move in a 2D plane, so it was much easier for us to plan paths for a mecanum drive robot.

Additionally, if we eventually implement autonomous package pickup, it will be easier for the robot to adjust laterally to realign itself with the package.

B. *Testing Methods*

This semester, our testing methods were rather primitive. To test whether path planning worked, we simply placed the robot on a map and saw if it ran over any walls during its path execution. To test our lifting mechanism, we simply placed a package on the lifter's fork, and saw if it could successfully raise the package.

While these experiments are primitive, they are still useful. After testing our path planning, we noticed that we weren't sufficiently taking the robot's geometry into account when avoiding walls. So, while the center of our robot would clear the walls, the edges of our robot would not. After testing our lifter, we realized that, if the package was too heavy, the package would bend the lifter until it was on the ground, and the gear driving the lifter would skip. This experiment is currently guiding our redesigns of the lifter, so that we ensure that in the next revision, the gear is positioned precisely and the lifter's fork is stiffer.

C. *Team Organization*

Team coordination at the beginning of the semester was lacking – roles were not particularly well defined , but eaus was assigned a part. Before our project started picking up pace, each group member had a part in designing or debugging a particular module within our system. Adam by far had the most contribution towards the project. He set up all the wiring for our robot frame and also wrote programs to use our motors and encoders within ROS. He started off strong on the before we came together as a team mid-November. However, when mid-November came around, the rest of the team members filled in their roles. Josh wrote the path planning code, and Victor and Johnathan played a significant role to make the path planning ROS node work. Alex translated the path planning directions to real-time motor velocities. With each ROS nodes working individually, we were able to successfully run and test all nodes together without many issues. Our team was able to collaborate well to finish off our project before MDR.

For our team collaboration during Spring semester, we aim to set well defined roles for each team member to ensure the steady progress in our work. This will be done in tandem with the Gantt chart as a general reference for our project milestones.

D. *Beyond the Classroom*

Adam Rivelli:

So far, I've needed to learn a lot about how ROS works and about the ROS ecosystem in order to be a contributor this semester. Initially, ROS was very overwhelming to me; it was hard to figure out how to do anything with it, and while it seemed to be very useful for any project involving a robot, I found it difficult to see exactly how we should implement it in our project. To learn about ROS, I found that the basic ROS tutorials were the best place to start. Overall, I thought that the ROS documentation was either lacking or too terse for me to understand, so I generally think that the tutorials for any given part of ROS are far more useful than the documentation for that part. Additionally, the ROS community is very helpful, and I found their Q&A forums to be very useful.

Additionally, I had to learn a bit more about 3D printing, especially about how to set tolerances on parts. The lifter uses three parts which are 3D printed. One part needed to make a press fit with another part, the second part needed to be a press fit with a gear rack, and the third part needed to both be a press fit with another part, and needed to have a slip fit with the gear rack. I didn't get the tolerancing for these parts perfect - they all needed heavy post-processing in order to work - but I have learned a lot about the significance of part tolerancing.

Victor Wong:

On top of learning how ROS works from scratch, I spent a significant amount of time integrating the individual contributions of others to form a working system. My prior knowledge in Python and object-oriented programming helped in integrating and debugging the various ROS nodes written in Python and C++. Additionally, I learned about ultrasonic sensors and Arduino from writing the collision detection node.

Joshua Savard

Throughout the course of the semester I have learned many things from this years senior design project, the first of which is the project development lifecycle. While at first it may not be something that you often think about, it's importance should not be underestimated. At at least 3 different interviews throughout this semester, I have been asked about the product development lifecycle, and a situation where I was able to apply it. Looking back and reflecting upon how many times I have been able to apply this information I now realise how important it is when actually applying it to the industry. On a more technical note, I also got a significant amount of experience in understanding ROS. From drone swarms to vacuum robots, ROS is being used to create new innovative drone and robot ideas, and to be able to get hands on experience working with these current technologies is something I will likely be able to apply later on in my career. I also got hands on experience working in C++ as well as some minor experience in python. I am glad to be able to say that this is relatively insignificant because I have gained enough experience working with so many languages that I can now pick up two more with little effort. Another large takeaway that I would like to mention is the experience I gained integrating other people's software into my own. Utilising other people's software is not something I have often done in the past, but that is the way that most software development is done in practice, so it is good to gain that experience. One last thing that I took away from my time working in senior design project is to make sure that when you have downtime you are preparing for and planning out future schedules so that way you can avoid long sprints when things go wrong and project deadlines come near.

Johnathan Shentu

As a result of poor planning early in the semester, I was unable to contribute as much as I would have liked to for this project thus far. However, I believe I was able to grasp fairly well the scope of the project, and was still able to contribute to debugging and (the current) navigation code. Like the other team members, I spent a significant percentage of time working on this project learning ROS.

Alex Yen

I personally feel like I did not contribute as much as I had hoped to. This semester was particularly challenging for with along with other commitments. That being said, I had the most experience ROS in our team, so I did not have a big learning curve while implementing ROS nodes. I was able to strengthen my familiarity with ROS, such that I feel more confident in my own ability to use ROS in future projects.