Spring 2019 ME 102B Project OLLY Final Report

Jason Anderson, Ben Chang, Ryan Cosner, Samuel Kruger, Rachel Lim, Rohan Sinha

***Abstract*— The Oversized Load Lifting and Yielding (OLLY) robots carry long extended loads between them. The front robot is controlled by user input, while the rear robot maps the surroundings using a SLAM algorithm from LIDAR data and autonomously follows the front robot at a fixed distance. We implemented two model-predictive controllers to maintain robot distance, stay within road constraints, and minimize actuation. A set of three omnidirectional wheels allows the robots to handle any velocity command whether from user input or from controller output. In tests, the follower robot was able to hold the loading distance within ± 1 inch while (1) turning a corner and (2) staying on a fixed line. A video demo can be found here[7].**

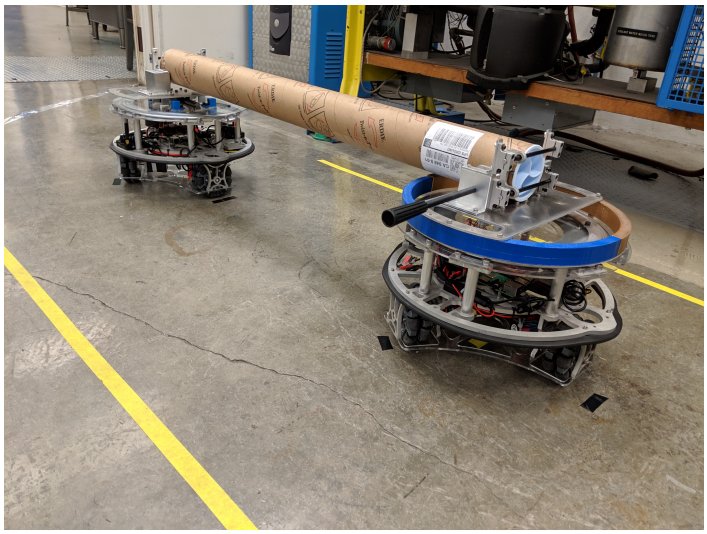

Fig. 1. Finished picture of the OLLY robots, with Folly on the left and Molly on the right

I. Introduction

In order to transport large objects over land which cannot fit into one truck ("loads"), modern transportation companies utilize a complex sequence of operators, machinery, and vehicles. Most commonly, one master truck will drive the front of the load while a follower truck will carry the back of the load. Each driver communicates and moves slowly in order to navigate difficult turns and terrain. In addition to these drivers, other safety officials might be standing by in order to ensure that the two drivers have all the information they need to drive safely.

Project Olly aims to bring a novel robotic solution to this transportation issue using an autonomous application. By equipping the follower vehicle with sensors to map the surroundings, the follower robot will autonomously track the motion of the master robot, thereby removing the need to have more than one operator. To demonstrate this solution, two squat Oversized Load Lifting and Yielding robots (termed "Olly") were created to model the motion of these vehicles. The Master Olly (termed "Molly") receives user input in order to move in any direction while the Follower Olly (termed "Folly")[1] manages to keep the load at a fixed distance.

The Olly robot is broken down into four components. First, the chassis and drivetrain use omnidirectional wheels with stepper motors to accurately move in any directional input. Second, the gripper attaches the oversized load between Molly and Olly. Third, a LIDAR sensor works with a Raspberry Pi and Arduino to process on-board sensing and motor commands. Lastly, a model predictive controller determines Olly's ideal absolute velocity based on Molly's motion. The Folly robot was able to track the Molly robot in a variety of situations to within a few inches of the desired place, which did not strain the load due to compliance on the attachment point.

[1]In some places, the follower Olly is just termed "Olly." For the sake of this report, the convention of "Molly" and "Folly" will be used.

II. Mechanical Hardware

A. Chassis

Olly's chassis features a lightweight, waterjet aluminum baseplate that is identical across both Molly and Folly robots. The chassis is designed to be a rigid, load bearing platform that fundamentally serves two purposes. First, the chassis integrates the mechanical subsystems. The baseplate undercarriage mounts to the drivetrain, doubly supporting each drive axle and ensuring a precise 14cm wheelbase radius to create a robust platform for control. Vertical standoffs atop the baseplate affix the gripper mechanism. Second, the chassis achieves electrical hardware and sensor integration. Careful hardware placement ensures proper cable management, accessibility, precision 15cm LIDAR mounting radius, and 360 degree LIDAR field of view.

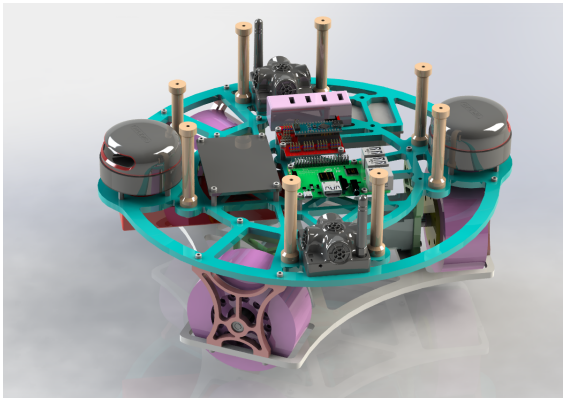

Fig. 2. CAD Model of Chassis and Drivetrain

B. Drivetrain and Dynamics

Each Olly is actuated using 3 omni-directional wheels to create a holonomic drive system. These wheels are actuated using NEMA 17 stepper motors, which were chosen for simple open loop control. While inefficient, these motors still have a sufficient holding torque of 0.49 Nm and operate well for our slow target velocity. A custom driveshaft was designed for each wheel. A holonomic drive was chosen for the robot to allow it to move in any direction without having to reorient its heading. The 3-wheeled variant of the omniwheel system was chosen because it produces a nonsingular transformation matrix relating wheel motion to cartesian motion. On the other hand, a 4-wheeled omniwheel system is nonsingular and multiple actuation methods can produce the same motion.

In order to develop the system kinematic model, it is assumed that the wheels will never experience forces that exceed the limits of static friction. Also, it is assumed that the wheels slide perfectly in their perpendicular directions. These assumptions

produce the following transformation for the conventions as shown below:

$$\begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} sin(\frac{\pi}{3}) & cos(\frac{\pi}{3}) & -r_{robot} \\ -sin(\frac{\pi}{3}) & cos(\frac{\pi}{3}) & -r_{robot} \\ 0 & -1 & -r_{robot} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix}$$

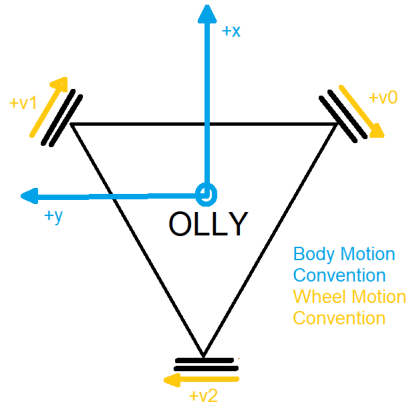

Fig. 3. Drivetrain kinematic model conventions

C. Gripper

In order to attach loads to both Molly and Folly, a mechanism is needed that can attach to a variety of objects without putting undue stress on the object being carried. The gripper accomplishes this through three components. First, the actual gripping is accomplished via a custom vise with rubber padding to ensure minimal damage to the load. This vise is bolted to a top mounting plate, which has many threaded holes to facilitate any other method of load attachment. Next, on Folly, a linear rail provides translational compliance for this top mounting plate, since the vise can then freely slide back and forth along the load's mounting axis. This allows for Folly to be $\pm$5cm of the proper distance away from Molly without pulling on the other robot. Lastly, both robots have a large thrust bearing beneath the vise in order to introduce rotational compliance on the direction of the attached load.

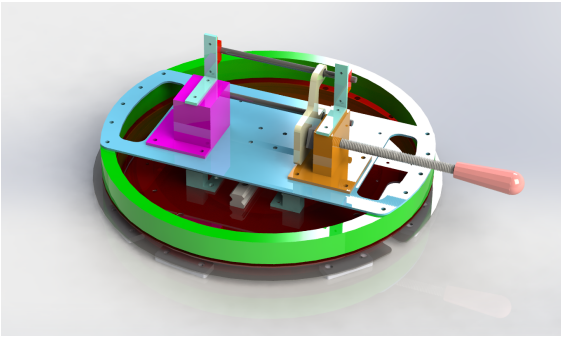

Fig. 4. CAD Model of the Gripper on Folly

This compliance is important to ensure that the omnidirectional drive can rotate freely beneath the load without being concerned about one specific orientation. These three features make up the gripper, thereby isolating the process of the load attachment from the motion of the wheels, which importantly allows for each robot to be treated as a particle for the following control algorithms.

III. Electronic Hardware

A. System Architecture

Figure 5 shows a high level overview of our system architecture. Both Molly and Folly are equipped with computers that

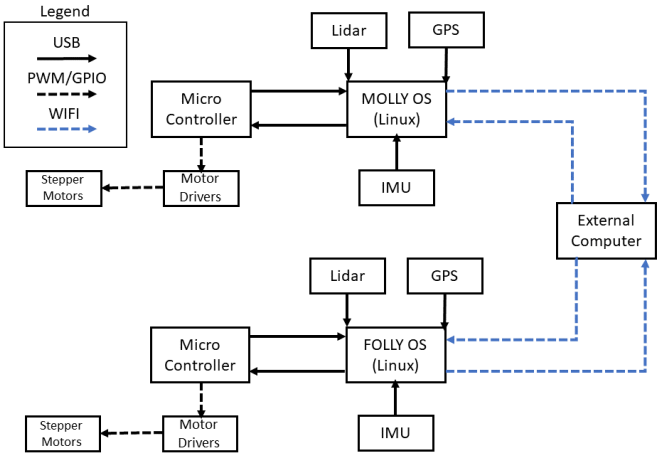

Fig. 5. System Architecture Diagram

run a Linux operating system (Ubuntu 16.04 LTS) and a microcontroller (in this case Arduino Nano's). The Linux computers connect through USB to the sensors to receive data, as well as to their respective microcontroller to send and receive actuation commands and feedback. Although this work only fully utilizes the sensor capability of the LIDARs, all our vehicles equip LIDARs, Indoor GPS beacons, and IMUs. The microcontrollers interact with the stepper motors through direct PWM interface as mentioned in the following section.

Additionally, the Molly and Folly operating systems are connected to an external Linux computer through a local Wifi Network managed by Folly. The Wifi network allows Molly and Folly to exchange information freely with each other using TCP over SSH connections. Moreover, a connection with an external computer allows Molly and Folly to offload computational load from the planning, estimation, and control algorithms by sharing sensor data and running these algorithms remotely. In addition, a human operator can give commands to, or control the system by sending inputs through the external computer, for example by using an Xbox controller. All of the code written for this project is accesible online, and access can be requested by following the link in [6].

B. Arduino and Motor Drivers, and Embedded Software

The Raspberry Pi sends velocity commands to an Arduino Nano which used the kinematic model described above to convert the commanded velocity into PWM signals. The Arduino Nano sends 5V digital commands to actuate steps, which are received by DVR8825 motor drivers. Each robot has three motor drivers, one associated with each stepper motor. Each driver received discrete "step" signals from the Arduino as well as a 12V power directly from the system's LiPo battery.

The stepper motor drivers inherently control for position, so the embedded software on the Arduino had to adjust the timing of the step commands so that the system could be driven with velocity instead of position-based commands.

C. Computer

The external Linux computer, running Ubuntu 16.04 LTS with ROS Kinetic, connects to the onboard computers on Molly and Folly through Folly's local Wifi network. In addition to running the ROS master, it handled the majority of the computational load, connected to the Xbox controller, and ran the Rviz display.

D. LIDAR

Both Molly and Folly are equipped with RPLIDAR A2 [1] Light Detection and Ranging (LIDAR) sensors. These sensors

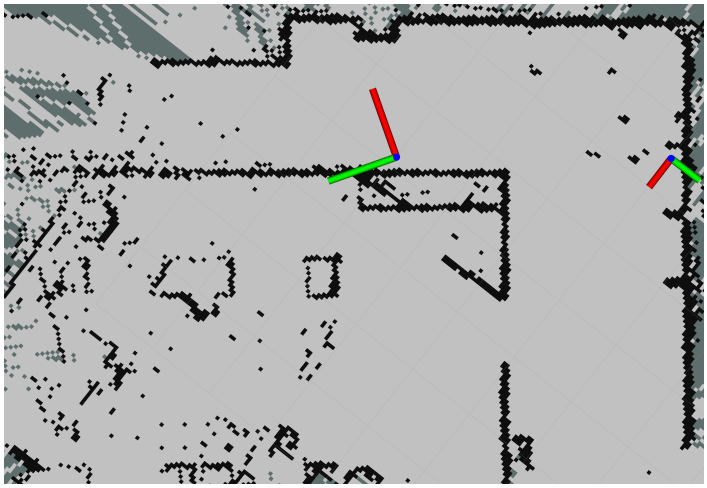

Fig. 6. Map Generated From the SLAM Algorithm

rotate at high velocity and fire laser beams into their surroundings. The LIDARs detect the light that bounces back, which gives them spatial information of their environment. This data is accumulated over time, and is used by the robots to build a map of their environment and localize themselves inside it, as described in a later section.

IV. Software System

A. Robotics Operating System

We implemented our code using Robotics Operating System (ROS), a framework that has become the industry standard used for complex robotics software development. In ROS, separate algorithms or sections of a system run as parallel processes referred to as 'nodes'. Each node publishes/subscribes to topics of specific message data types, allowing for highly modular development and easy implementation of open source packages.

Molly and Folly each have a Raspberry Pi running Ubuntu 16.04 with ROS Kinetic. We leveraged the ROS's Multiple Machines framework to setup the communication networks between the robots discussed in the previous section.

Our high level algorithms are wrapped in nodes implemented in Python 2.7. The low level node that controls motor PWM signals to track robot state velocity references was written in C++ and runs on the Arduino. The LIDAR driver node runs on the Raspberry Pi. The rest of the nodes run on the external computer, such as the path planning, joystick input conversion, and SLAM nodes.

B. State Estimation through LIDAR based SLAM

As mentioned above, a LIDAR sensor was mounted on each Olly. We used the point-cloud data from these sensors to build environment maps. To do this, we used a Simultaneous Localization and Mapping (SLAM) algorithm called Hector Slam [8]. Figure 6 is a map that we built of the basement of Hesse Hall at UC Berkeley. In the image, the darker a line is, the more confident the robot is that that region is a solid wall.

SLAM algorithms are probabilistic algorithms that hinge around non-linear Kalman Filters (EKF in this case). First off, the maps and LIDAR scans that are generated are represented by Occupancy Grids, discrete grids representing 2 dimensional space. The value at an x-y coordinate (1 or 0 for the LIDAR scan, or a probability value) represents the certainty that an object exists at that location. First the SLAM algorithm tries to find the current position of the robot by matching an incoming LIDAR scan to the map it previously built by solving an optimization problem of the form:

$$\hat{x}^*_{init} = arg\min_{\hat{x}_{init}} \sum_{i=0}^{n} [1 - M(S_i(\hat{x}_{init}))]^2 \tag{1}$$

Where $M(\cdot)$ represent the value of the map at a global coordinate position and $S_i(\cdot)$ represents the transformation of the i'th lidar scan coordinate into the global map frame by $\hat{x}_{init}$.The $\hat{x}^*_{init}$ resulting from this scan matching step is then used by an Extended Kalman Filter and any (optional) other sensors to generate a state estimate. A similar procedure is then run in reverse to update the map.

In our application, we have two robots that move around independently, which makes it much harder to build a single map using both robots' lidar scans and state estimates. Localizing one robot in a map built by the other gives ambiguity as to which robot should be building the map, since our problem setup results in both robots seeing vastly different sections of the world that may or may not have been mapped yet. Instead we ended up relying on the Hector Slam implementation available to us, with both robots SLAM-ing their environments independently from each other. We operate from the assumption that the robots are initially at a given relative orientation, which we then use to transform the state estimate from Folly into the global frame of Molly. We found this to be sufficiently accurate for our application since lidar based SLAM is not subject to sensor drift. Moreover, the probabilistic nature of these algorithms makes them robust enough to ignore the other Olly vehicle. Although both lidar sensors are scanning the same spatial plane (since both Molly and Folly have identical design), we found interference from the sensors to be negligibly small.

C. Joystick control

Molly was controlled by an Xbox controller connected to the external computer. For the joystick driver, we used the Joy package which published the controller state [2]. We then converted those inputs from inertia to body frame using the orientation of Molly as determined from SLAM.

D. Path Planning and Control

Project Olly uses receding horizon optimal control strategies commonly referred to as Model-Predictive Controllers (MPC) to achieve autonomous operation of the Folly. The MPC uses an estimate of Folly's current position, the current position and velocity states of Molly, a simple dynamic model, and preset constraints on states and inputs to achieve an objective over time. From a high level, these objectives are (1) to maintain a preset distance from Molly, (2) keep Folly within a specific map area, and (3) minimize actuation. For Project Olly, we created two different MPC algorithms, described in Sections IV-D.2 and IV-D.3 below, each of which achieves the aforementioned in a different manner.

1) The MPC Dynamic Model: The MPC assumes the following dynamic model of Olly, where k designates a specific discrete time, and Δt is the time between time k and $k+1$.

$$\begin{bmatrix} x \\ y \\ \theta_z \end{bmatrix}^{k+1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta_z \end{bmatrix}^{k} + \begin{bmatrix} \Delta t & 0 & 0 \\ 0 & \Delta t & 0 \\ 0 & 0 & \Delta t \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega_z \end{bmatrix}^{k} \tag{2}$$

To simplify and incorporate standard MPC notation, we make

the following substitutions.

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ \theta_z \end{bmatrix}$$
$$A = I_3$$
$$\mathbf{u} = \begin{bmatrix} v_x \\ v_y \\ \omega_z \end{bmatrix}$$
$$B = \Delta t \cdot I_3$$

Therefore, Equation (2) becomes Equation (3).

$$\mathbf{x}^{k+1} = A\mathbf{x}^k + B\mathbf{u}^k \tag{3}$$

2) Analytical MPC Position Control: Folly is constrained to move along a line in the inertial frame. Folly uses the current position of Molly and the known lifted-object length l to deduce the circle of points that maintain a distance l. The intersection of the line and circle yields two possible optimal positions for Folly, so Folly selects the optimal position closest to its current position estimate. This selected optimal position becomes the setpoint (x_m, y_m) inertial position for Folly for the following MPC.

At every discritization time, the optimization problem of Equation (4) is solved, where n is the horizon, the number of discrete times in the future considered, $\mathbf{x}_0$ is the current state estimate of Folly, and $\mathbf{u}_{\max}$ is the maximum allowed actuation commands. To prevent Folly from arbitrarily rotating yaw, and incorporating the desired optimal position from above, the selected optimal state is defined as $\mathbf{x}_m = [x_m, y_m, 0]^\top$ and incorporated into the MPC. Using Molly's broadcasted velocity state, Folly predicts Molly's future states by assuming Molly's velocity won't change, meaning that the circle and line problem is solved for each time step. The solved $\mathbf{u}_t$ of the final MPC of Equation (4) will allow Folly to track the optimal setpoint position.

$$\begin{aligned} \min_{\forall \mathbf{x}_t \ \forall \mathbf{u}_t} \ & \sum_{t=1}^{n} 100 \cdot \|\mathbf{x}_t - \mathbf{x}_m\|^2 + \|\mathbf{u}_t\|^2 \\ s.t. \ & \mathbf{x}_t = A\mathbf{x}_{t-1} + B\mathbf{u}_t \ \forall t \in [1 \dots n] \\ & \mathbf{x}_0 = \mathbf{x}_0 \\ & \mathbf{u}_t \leq \mathbf{u}_{\max} \ \forall t \in [1 \dots n] \end{aligned} \tag{4}$$

For Folly, this MPC was implemented in Python with convex solver CVXPY[5]. We selected a discritization step time 0.3s, horizon 10, and an element-wise $u_{\max}$ of 0.1 m/s. Every discritization step time, $\mathbf{u_1}$ is sent to the the arduino for actuation, and the problem is then updated with the current state and resolved.

3) Nonconvex MPC: Folly is constrained to move within a specific convex region of inertial space X that is preset in agreement with the map road. Incorporating the previous notation, except that now $\mathbf{x}_m$ is Molly's current position estimate, at each discretization time, the optimization problem of Equation (6) is solved. Using Molly's broadcasted velocity state, Folly predicts Molly's future states by assuming Molly's velocity won't change, meaning that that Folly anticipates Molly's future position in the problem.

$$\begin{aligned} \min_{\forall \mathbf{x}_t \ \forall \mathbf{u}_t} \ & \sum_{t=1}^{n} 10 \cdot \left\| \|\mathbf{x}_t - \mathbf{x}_m\|^2 - l^2 \right\|^2 + \|\mathbf{u}_t\|^2 \\ s.t. \ & \mathbf{x}_t = A\mathbf{x}_{t-1} + B\mathbf{u}_t \ \forall t \in [1 \dots n] \\ & \mathbf{x}_0 = \mathbf{x}_0 \\ & \mathbf{x}_t \in X \ \forall t \in [1 \dots n] \\ & \mathbf{u}_t \leq \mathbf{u}_{\max} \ \forall t \in [1 \dots n] \end{aligned} \tag{5}$$

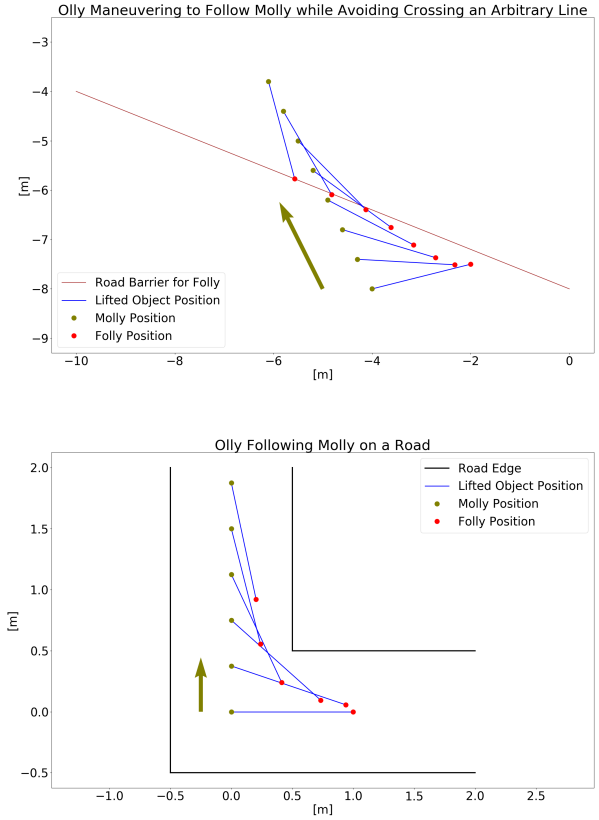

Fig. 7. The first figure shows Folly following Molly according to the MPC of Section IV-D.3 for an arbitrary road constraint. The second figure shows Folly following Molly according to the ME102B demo. The second figure shows the road only while there are three active constraints that form three sides of a box at any given time not shown in the second figure. As Folly moves, the three active constraints reflect the current section of the road.

The simplified Equation (5) encourages Folly to stay on the radius l circle around Molly and minimize actuation while within a road-like constraint. While the object function is not convex, all local minima are global mininma, due to the symmetry of the cost function.

This MPC was implemented in python with solver GEKKO [4]. We selected a discretization step time 0.3s, horizon 10, and an element-wise $u_{\max}$ of 0.1 m/s. Every discretization step time, $\mathbf{u_1}$ is sent to the the arduino for actuation, and the problem is then updated with the current state and resolved. Figure provides an example path of Folly following Molly using the MPC.

In the normal movement of Folly, the road map will create overlapping road constraints that cannot be satisfied simultaneously (e.g., an L shaped road cannot be described as a single convex shape). To account for this, Folly caches multiple Equation (6) MPC problems, each covering a section of the map. Each MPC problem is identical except for X. The current position of Folly determines which cached MPC is solved. To account for sensor noise, we added additional slack variables, denoted by $\lambda = [\lambda_1, \dots, \lambda_n]^T$, to soften the polytopical state constraints. These soft constraints maintain feasibility of the problem when sensor noise pushes Folly out of the desired feasible region. This results in the following final MPC problem:

$$\min_{\forall \mathbf{x}_t \ \forall \mathbf{u}_t} \sum_{t=1}^{n} 10 \cdot \left\| \|\mathbf{x}_t - \mathbf{x}_m\|^2 - l^2 \right\|^2 + \|\mathbf{u}_t\|^2 + C \sum_{t=1}^{n} \lambda_t \tag{6}$$

$$\begin{aligned} s.t.\ \mathbf{x}_t &= A\mathbf{x}_{t-1} + B\mathbf{u}_t \ \forall t \in [1 \dots n] \\ \mathbf{x}_0 &= \mathbf{x}_0 \\ G\mathbf{x}_t - h &\leq \lambda_t \ \forall t \in [1 \dots n] \\ \lambda_i &\geq 0 \forall t \in [1 \dots n] \\ \mathbf{u}_t &\leq \mathbf{u}_{\max} \ \forall t \in [1 \dots n] \end{aligned}$$

In our implementation, $C = 1$.

4) Comparison of MPCs: The MPC of Section IV-D.2 restricts the motion of Folly to a line; however, the optimization problem is convex and easy to solve. The MPC of Section IV-D.3 restricts the motion of Folly to a convex set; however, the optimization problem is nonconvex which complicates the robustness of the solution. We prepared both planners anticipating that the solver of Section IV-D.3 might fail in the demo; however, this did not happen. Since the MPC of Section IV-D.3 better achives the Project Olly objective and is able to afford better road mobility, it is the better MPC.

E. Rviz

To assist in debugging as well as explaining concepts during the demo, we used Rviz [3], a visualization software commonly used in ROS. We used it to show the map built from Folly's LIDAR data, the direct LIDAR laser scan, and the positions and orientations of both Molly and Folly.

V. Results

All of the results of the demonstrations can be seen in the corresponding video. [7]

The first two demonstrations of Project Olly serve to show how well Folly can keep a given distance given differing external boundaries. In the first test, Folly is given six constraints within the L-shaped track, while in the second test, Folly is given only the constraint of staying on a straight line. In both tests, Folly is able to meet the obstacle constraint as well as the distance from Molly constraint within ± 1 inch, which is determined based on the fact that the gripper translational compliance only has a range of that distance. It is worth mentioning that the controller for Folly runs into issues when the control of Molly pushes Folly towards a wall. In this way, the controller has to decide between violating either a wall constraint or a fixed distance constraint. In some cases, the controller is smart enough to avoid this issue, but when Folly is belligerently sent into the corner the controller can fail. Besides this one place of concern, Folly keeps up rather well with Molly. The gripper isolates the motion of the robot from the load itself and the omnidirectional wheels serve well to allow for complex motion.

In the third demo, one of the robots is told to go to multiple waypoints on a square using only the LIDAR as the sensor. Visually, the robot accomplishes this motion very well even with the lack of sensors. This result shows the success of the SLAM algorithm in positioning the robots in space.

VI. Conclusion and Future Work

Through these tests, Project Olly was able to demonstrate the success of autonomous control of one robot based on the controlled input of another. Though Folly's positioning was inherently not exact, the compliance within the gripper ensured that the SLAM map's positioning would be sufficient to localize each robot with respect to one another. Additionally, while the robots moved relatively slowly due to less powerful motors, each robot moved correctly, overcame heavier loads, and navigated through the designed geometry well.

For future iterations, the robots certainly could be made more robust in two manners. First, the hardware could be improved by upgrading the motors and adding additional sensors. A more sophisticated gripper could also be used to attach a wider variety of loads. Second, if more time had been allowed, more complicated tracks would result in more complex road constraints, so improving the way Folly handles the surroundings would help the autonomous robot be more robust. Other functionalities which would be worthwhile to explore would be object detection and avoidance. A robot with four-wheel drive also could be useful in representing how this control could be implemented in cars.

Regardless of these improvements, Project Olly still succeeded at showing how multiple robots can work together to simplify transportation challenges. By understanding the design methodology used here as well as the successes showcased in the project, one can understand how robotic solutions can be reduced to simpler terms in order to solve non-trivial real-world problems.

References

[1] https://www.slamtec.com/en/lidar/a2.

[2] http://wiki.ros.org/joy.

[3] http://wiki.ros.org/rviz/.

[4] Logan Beal, Daniel Hill, R Martin, and John Hedengren. Gekko optimization suite. *Processes*, 6(8):106, 2018.

[5] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[6] R. Cosner S. Kruger R. Lim R. Sinha J. Anderson, B. Chang. https://github.com/ollyme102b.

[7] R. Cosner S. Kruger R. Lim R. Sinha J. Anderson, B. Chang. https://youtu.be/VnuGRmiWuGc.

[8] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. A flexible and scalable slam system with full 3d motion estimation. *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, November 2011.

Appendix

In this paper, x, y, z, correspond to the Cartesian position state where x and y are the lateral coordinates with x generally meaning forward and z meaning upward, and x, y, and z forming a right-hand coordinate system. The velocities in the corresponding coordinate axes are denoted v_x, v_y, v_z; the Euler angle positions, θ_x, θ_y, θ_z ; and the Euler angle velocities, ω_x, ω_y, ω_z. For any states with respect to the body frame of the drone (centered at the drone center of mass with the aforementioned directional definitions) have the super script B, whereas, with no superscript, the states are global. A concatenates state vector is in bold, for example, $\mathbf{x}$ and $\mathbf{u}$. The length of time in between iterations is Δt, either in the simulator or in the controller. The iteration number of a specific state is denoted with a subscript, with x_0 being the x position state at present.

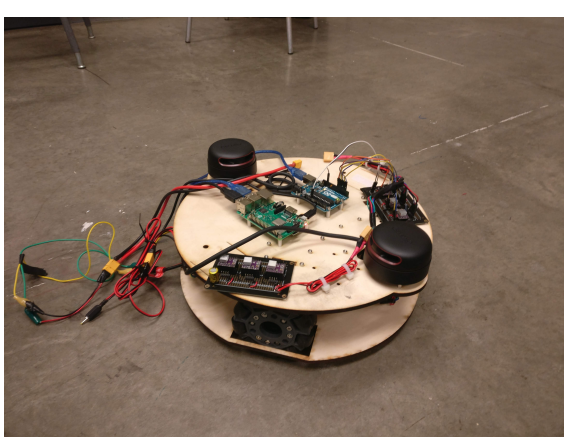

Fig. 8. Prototyping Test Platform