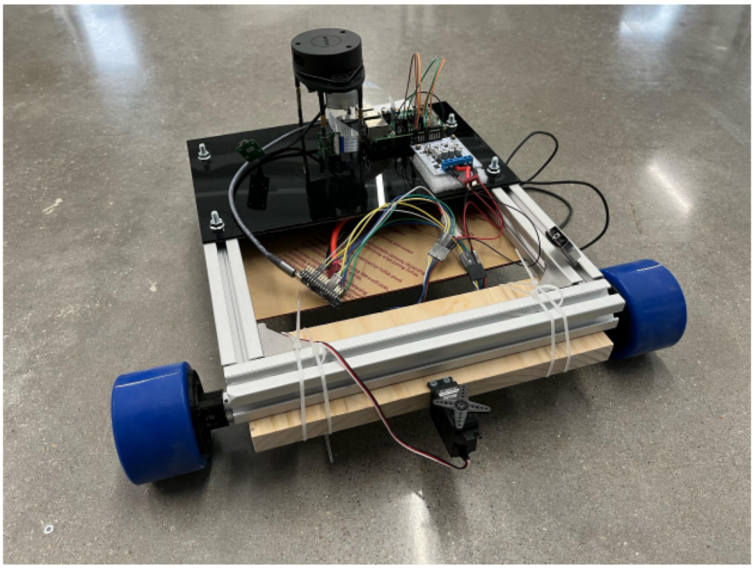
Robotics 2 Project 1: Bot-Bros Begin

Brandon Donaldson, Austin Miller, and Will Ward

UCA Department of Physics & Astronomy

Date: March 17, 2022



Executive Summary:

An autonomous ground robot was designed and built from scratch and programmed on a Raspberry Pi. The robot uses ROS2 nodes to respond to user keyboard commands and motor encoders to regulate its own speed.

Parts List and Specifications:

CanaKit Raspberry Pi 4

Arduino Nano Every (ATMega4809 Processor)

Pololu 70:1 Metal Gearmotor #4754 (motor specifications listed in Table 1)

Pololu Dual G2 High-Power Motor Driver Shield #2518 (see Figure 1 and Table 2)

BEYST 84 mm Electric Skateboard Wheels

GOLDBAT LiPo 3s 11.1V 5200 mAH battery

Gigastone 5V 10000mAh USB-C Power Bank

Slamtec RPLIDAR A1 - 360 Laser Range Scanner

Arducam 5MP Camera for Raspberry Π

xArm 1S Intelligent Bus Servo Robotic Arm (Hiwonder)

Table 1: Robot General Specifications	
Robot Length	15 in
Robot Width	18.5 in
Height of Frame off Ground	4.125 in
Wheel Radius	4.19 cm
Wheel Separation	39.37 cm
Caster Wheel Radius	2.475 cm
Robot Max Speed	0.55 m/s
Pololu 70:1 Metal Gearmotor (#4754)	
Gear Ratio	70:1
Counts Per Revolution (CPR)	64

Motor Driver Board:

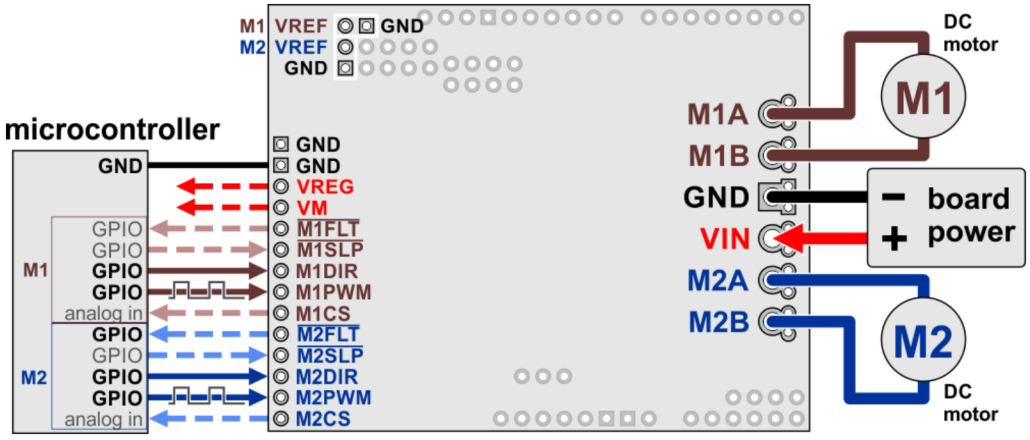


Figure 1: Motor Driver Board Connections [2]

We used a Pololu Dual G2 High-Power Motor Driver to control our motors with the Raspberry Pi. The driver board itself is powered by the LiPo 11.1V 5200 mAH battery. Figure 1

shows the schematic of the motor driver board provided by Pololu [2], and Table # lists the pin connections we used between the Raspberry Pi and the driver.

Table 2: Motor Driver to Raspberry Pi Pin Connections	
Motor Driver	**Raspberry Pi**
GND	GND
M1DIR	GPIO 25
M2DIR	GPIO 24
M1PWM	GPIO 13
M2PWM	GPIO 12

Design Process:

We began our project by disassembling an old robot chassis built on an 80/20 T-slot aluminum frame. The initial design was a rectangle frame built from the 80/20 and a wood board was to be fastened to the top with T-bolts. Since the motor output shaft is D shaped we decided to 3D print an insert (see Appendix A) that fits inside of the skateboard wheel hub and on the outside of the motor shaft, we then used a set screw to lock our D shaped shaft inside of the part. The wheels were able to be separated easily from the mounts so we took some electrical tape and wrapped the prongs to make them thicker and stay inside of the wheel better. Once this was done we made an order for a 5V 3A battery, motor mounts, and locking nuts. We fastened the caster wheels directly to the T-slot frame, but in order to keep the robot level we needed to add some height to the rear wheel mounts. We did this by mounting a ½” board using T-bolts and then mounting the motors to the wood (Figure 2).

The initial testing of the robot on the ground with code did not go as planned, the robot struggled to move and the wheels did not sit flush with the ground. Our first assumption was that the robot was too heavy for the motor mounts, so we replaced the ½” board on top with plexiglass that only covers the front half of the robot. This did not fix the bowing of the wheels,

so we swapped the motor mounts out with a stronger set, still no difference was made. We noticed that the mounts were technically upside down from the normal installation, and this created a pivot point. To add extra support, we used zip ties to the frame to keep the motor from bending the mounts, and this allowed for our wheels to sit flush with the ground (Figure 3).

The 3D printed skateboard wheel mount was short-lived. As we ran tests, the mounts slid off of the motor drive shaft. Since the part was plastic, after several uses the set screw hole was stripped and would no longer hold the shaft in place. To fix this problem, we found a metal 6 mm universal mounting hub that used two set screws on the shaft and 6 screws to mount the hub to the printed wheel mount, shown in Figure 4.

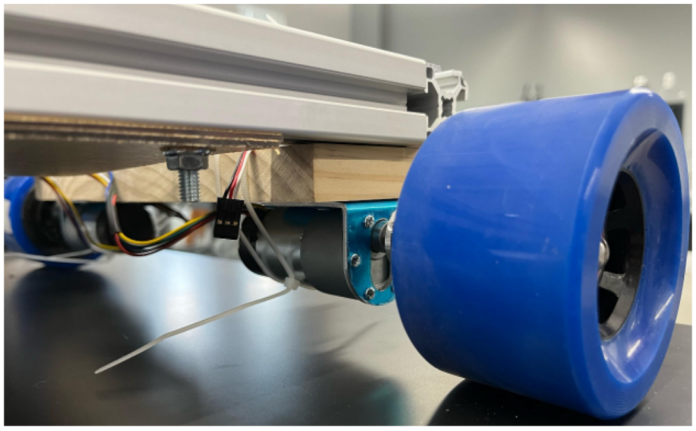


Figure 2: Motors mounted on 1/2" wood for levelness.

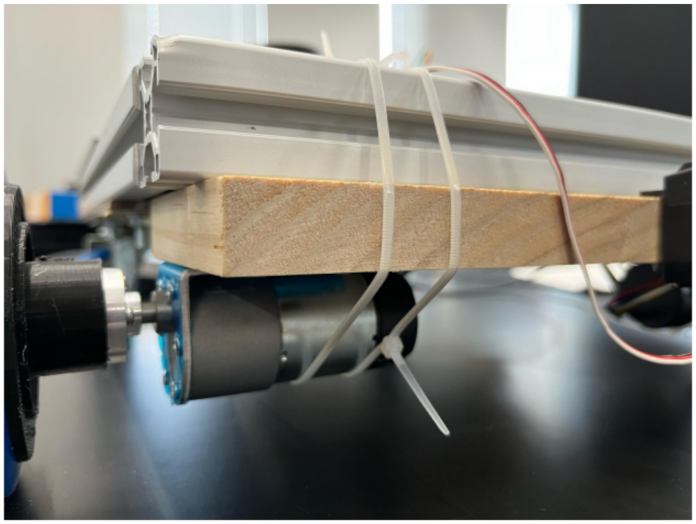


Figure 3: Motors zip-tied to frame to prevent sagging under the weight of the robot.

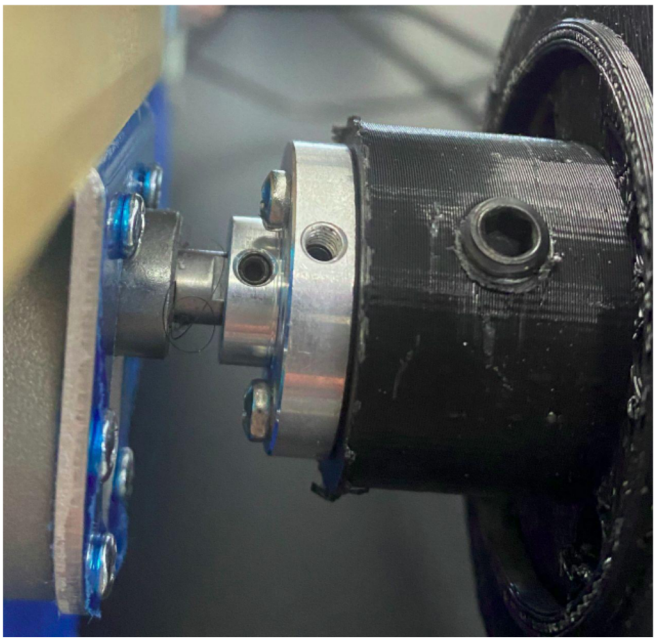


Figure 4: 6mm universal mounting hub mounted to both the axle and the 3D printed wheel hub.

After fixing that issue, the mounts began to pop out of the wheel. To fix this, we took a 2” long bolt with a large washer through the center of the wheel and screwed into the existing center hole for the drive shaft. The washer is there to give the bolt head a surface on the wheel to pull towards the 3D printed mount (Figure 5).

Figure 5: Wheel bolted into the printed mount.

Placing our components on the robot required us to think about elements that will be added later, like lidar and the robotic arm. In order for the lidar to gather good data, we needed to ensure the sensors would not be obstructed by any objects on the robot. Since the final project design includes a robotic arm, we decided to plan for this and mount the arm below the lidar. In

order to do that, we needed to raise the lidar sensor and find a place to tuck the arm out of the way that would still allow us to drive around. We decided to mount the arm on the same board the motor mounts are attached to, this lowers the initial height of the arm and decreases the amount of height we added to the lidar. Figure 6 displays how the final design stores the arm out of the way.

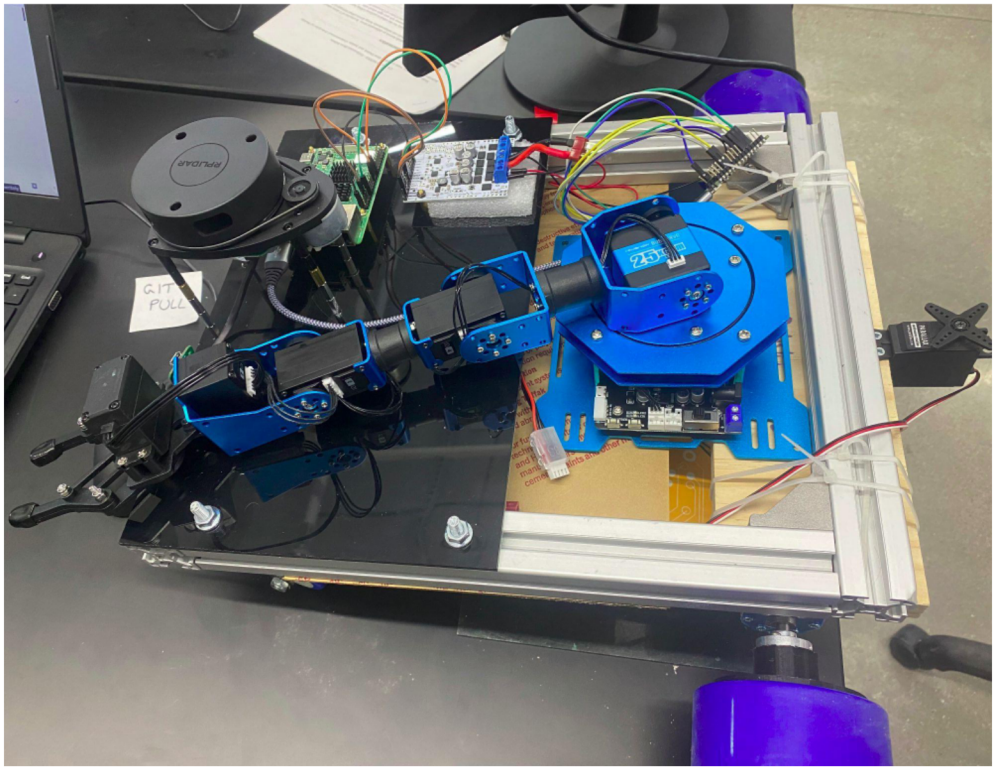


Figure 6: Final design iteration with robotic arm tucked out of the way.

Software:

After the robot was constructed and the motor driver board was working properly, we developed two methods of driving the robot: keyboard control through the "teleop_twist_keyboard" package and autonomous speed control using the motor's encoders and proportional gain control.

Keyboard Control:

Teleop twist keyboard is a ROS2 package that allows users to drive their robot using keyboard input. Running "ros2 run teleop_twist_keyboard teleop_twist_keyboard" in the

terminal launches a teleop_twist_keyboard node that publishes geometry messages to a Twist topic called /cmd_vel. The keys ‘u’, ‘i’, ‘o’, ‘j’, ‘k’, ‘l’, ‘m’, ‘,’, and ‘.’ keys all correspond to a different Twist message when pressed. Twist messages can contain linear and angular components, as shown by Figure 7. The teleop_twist_keyboard node only publishes linear x and angular z (rotation about a vertical z-axis) messages, allowing the robot to rotate in place, move in a straight line, or move in an arc.

```
robotics-b@roboticsb-Latitude-3590:~$ ros2 topic echo /cmd_vel
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

Figure 7: Twist message example for topic /cmd_vel

Our program subscriber_member_function.py creates a subscriber node that listens to the /cmd_vel topic and moves the robot depending on what message it hears. While the teleop_twist_keyboard node has the ability to change wheel speed, we did not incorporate this into our subscriber. We focused solely on interpreting what action was being published: forward, forward left, forward right, stop, backward, backward right, backward left, left, or right. Table 3 matches the /cmd_vel messages heard with the action our robot performs. When the robot is rotating on the spot or moving forward or backward, the speed is set to its maximum. When the robot is moving in an arc, one of the wheels is rotating at half speed while the other rotates at full speed.

Table 3: Teleop Twist Keyboard Robot Controls

Keystroke	msg.linear.x	msg.angular.z	Robot Action
I	0.5	0.0	forward()
U	0.5	1.0	forward_right()
O	0.5	-1.0	forward_left()
K	0.0	0.0	stop()

J	0.0	1.0	right()
L	0.0	-1.0	left()
,	-0.5	0.0	backward()
M	-0.5	-1.0	backward_left()
.	-0.5	1.0	backward_right()

Odometry:

The motors' encoder pins are connected to an Arduino Nano Every to read the encoder's counts and translate them into linear wheel speed. Table 4 shows how the motor's pins are connected to the Arduino.

Table 4: Encoder Pin Connections		
Color	**Function**	**Arduino Pin**
Green	Ground	GND
Blue	Encoder Vcc (3.5-20V)	3.5V or 5V rail
Yellow	Encoder A input	2 (left), 8 (right)
White	Encoder B input	3 (left), 9 (right)

The Arduino program NewEncoderCounter.cpp receives the number of counts in a time interval from the encoder and translates that into the linear speed of each wheel using the following equation:

$$linear\ speed = cps * 2 * \pi * R / (CPR * gearRatio)$$

where *cps* is the counts per second of the motor, R is the radius of the wheel, and *CPR* is the motor's constant counts per revolution. See Table # for a table of these constants.

The total linear speed of the robot is the average of the left and right speeds:

$$total\ linear\ speed = (linearLeft + linearRight) / 2$$

The angular speed of the robot is calculated by

$$angular\ speed = (linearRight - linearLeft) / wheel\ separation$$

where the wheel separation is given by Table 1.

The Arduino is programmed to print the counts per second of each motor, the linear speed of each wheel, and the total linear and angular speed to the serial monitor. The program odom_listener.py reads serial data from the Arduino when it is connected to the Pi through the USB port. odom_listener.py creates a publisher node that publishes the linear left and linear right speeds in a Twist message on the topic "linear_stuff". Left wheel linear speed is published to the linear.x component of the message, and right wheel linear speed is published to the linear.y component.

Proportional Gain Control:

Reading encoder messages and publishing motor speeds is useful because it allows the implementation of proportional gain control. In proportional gain control, the actual linear speeds of each wheel are compared to some target speed and incrementally adjusted to try to reach this speed. Our program bot_testing.py demonstrates proportional gain control over the two motors of our robot. First, the program creates a subscriber node to listen to left and right linear motor speeds published by odom_listener.py over the "linear_stuff" topic. Next the error between the actual speed and the desired speed is calculated:

$$error = desired\,speed - actual\,speed$$

Next, the left and right motor speeds are adjusted by increasing or decreasing their pwm proportional to the size of the error:

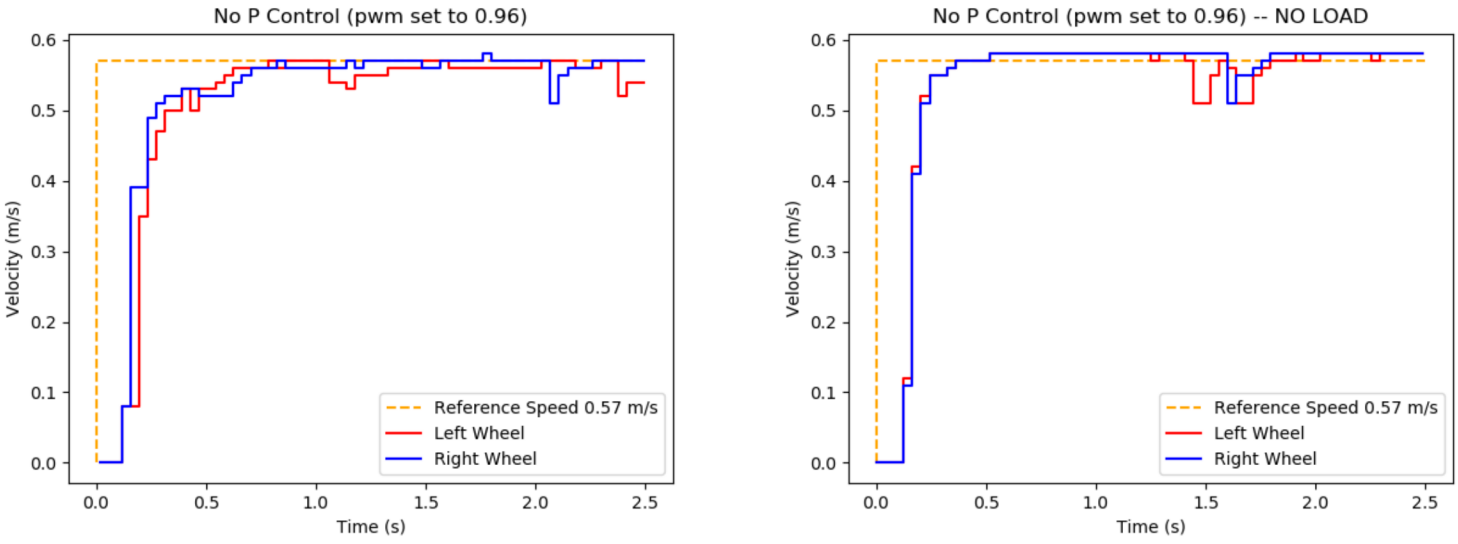
$$pwm = current\,pwm + p * error$$

where p is a fixed constant. We used p = 0.1 for both motors. This method is effective because if the error is large, the change in pwm is large. As the error decreases, the rate of change of the pwm also decreases so that the wheel speed carefully approaches the desired speed.

Driver Tests:

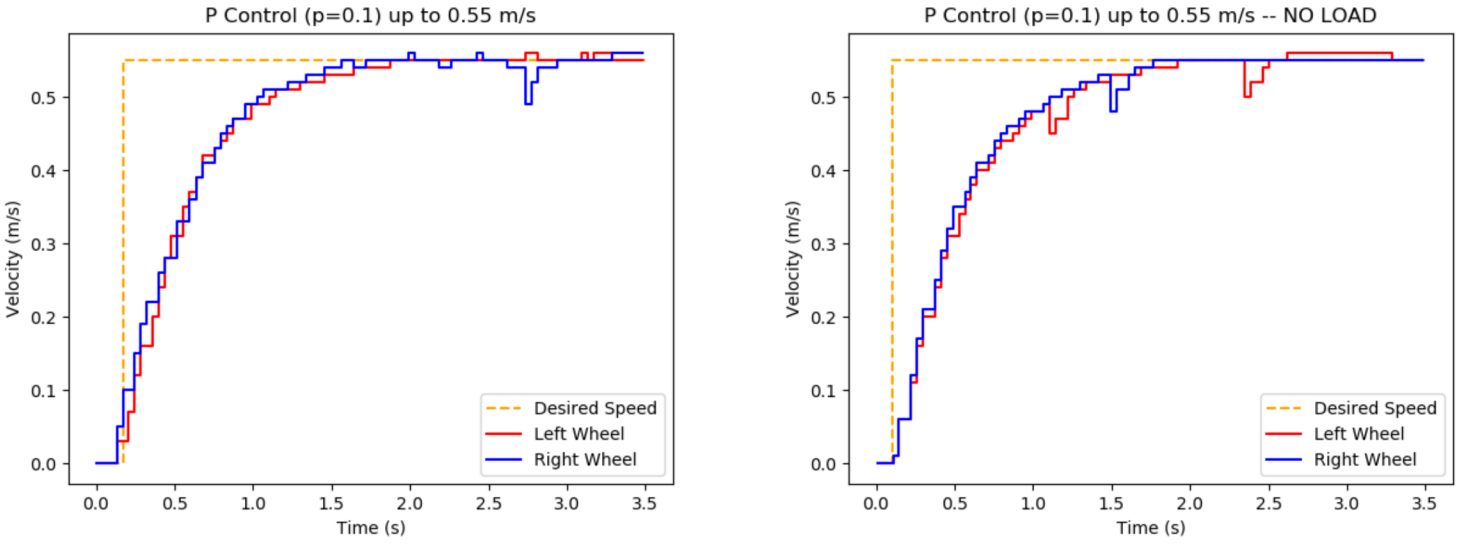
We performed a series of simple tests to evaluate the motors' response when the robot was fully assembled. First, we tested how fast the motors increase from zero to near their maximum speed. The pwm of the motors can be set on a scale from 0 to 0.999, but for this experiment we set the pwm to 0.96 due to the left and right motors not having the same maximum velocity. Figure 8a shows the results when the robot was driving on the ground, and Figure 8b shows the results when the robot was propped up off the ground (no load on the

motors except from the weight of the wheels). At time t = 0 seconds, the motor's pwms were changed from 0 to 0.96. When the robot was on the ground, both motors took about 0.75 seconds to ramp up to the speed corresponding to 0.96 pwm (0.57 m/s). When the robot was propped up on the table, the motors only took about 0.3 to 0.4 seconds to reach maximum speed.



(a) (b)

Figure 8: Left (red) and right (blue) motor response to sudden change in pwm with (a) and without (b) load.



(a) (b)

Figure 9: Proportional gain control p=0.1 with (a) and without (b) loading.

Second, we tested our proportional gain control program. The desired speed for the robot to reach was 0.55 m/s, and proportional gain constant p was set to 0.1. With this value of p, when the robot is driving on the ground, it takes about 1.5 seconds for the robot to reach the desired p, as shown in Figure 9a. Figure 9b is a plot of the same proportional gain test but with the robot's wheels propped up (no load from the robot body). There is not a significant difference in the time it takes for the robot to get to the max speed between these two set ups (load and no load). Figure 9 shows that the left and right motor speeds are much closer to the same value when proportional gain control is used versus without it (Figure 8). Visually, the robot is observed to drive in nearly a straight line after the motors both reach the maximum speed with P control. One flaw we noticed, however, is that the two motors do not stop at the same rate. This causes the robot to turn to the left when the motors are stopped abruptly.

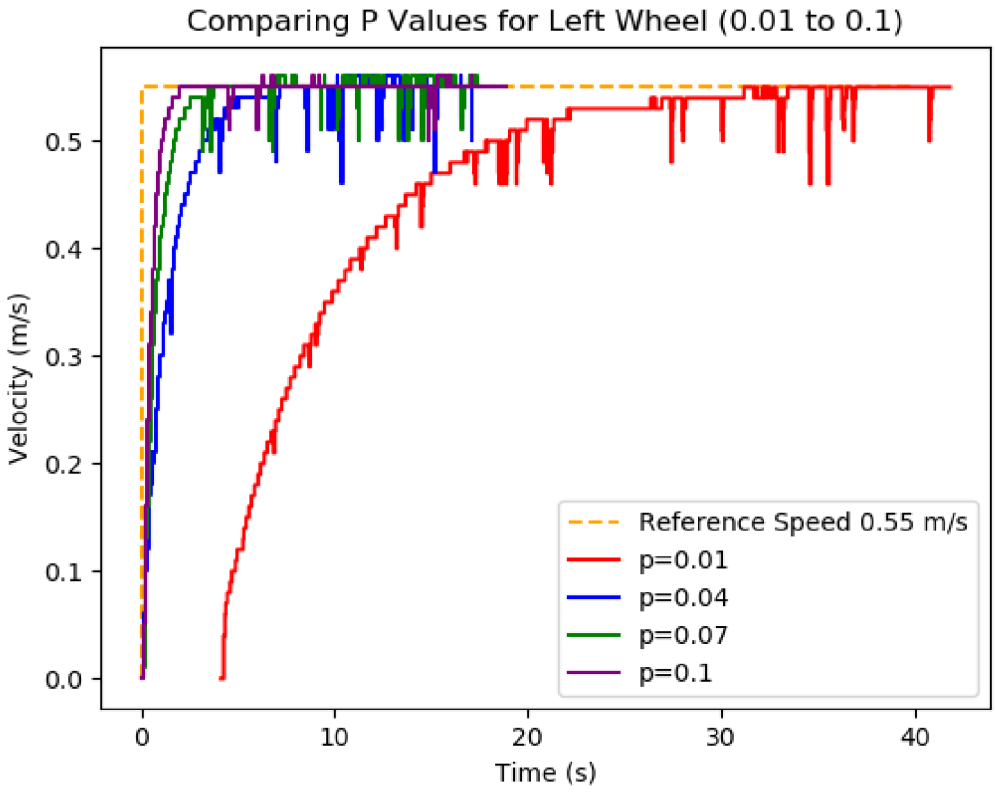


Figure 10: Comparing proportional gain control for different values of p.

Next, we observed the effect of changing the proportional gain constant between values of 0.01 and 0.25. Figure 10 shows the ramp up time of the left wheel for p values of 0.01, 0.04, 0.07, and 0.1. There is a significant difference between 0.01 and 0.04. When p=0.01, it takes the

motor 40 seconds to reach the desired speed, but when p=0.04, it only takes about 10 seconds. The difference between the other values is not as significant, but for p=0.1, it only takes about 2 seconds. Figure 11 compares the ramp up time for p values of 0.1, 0.16, and 0.25. The wheel ramps up faster for p=0.16 and p=0.25, but it overshoots the goal. A value for p between 0.1 and 0.16 may be the ideal constant that ramps speed up fast without overshooting. Figure 12 is an extension of the plot in Figure 11 to show the behavior over time. For all p values, the speed of the motor oscillates away from the desired speed and has to readjust. The motor speed oscillates furthest from the set speed for the higher values of p=0.16 and p=0.25, and during readjustment, both of these values cause the motor to overshoot the desired speed before settling.

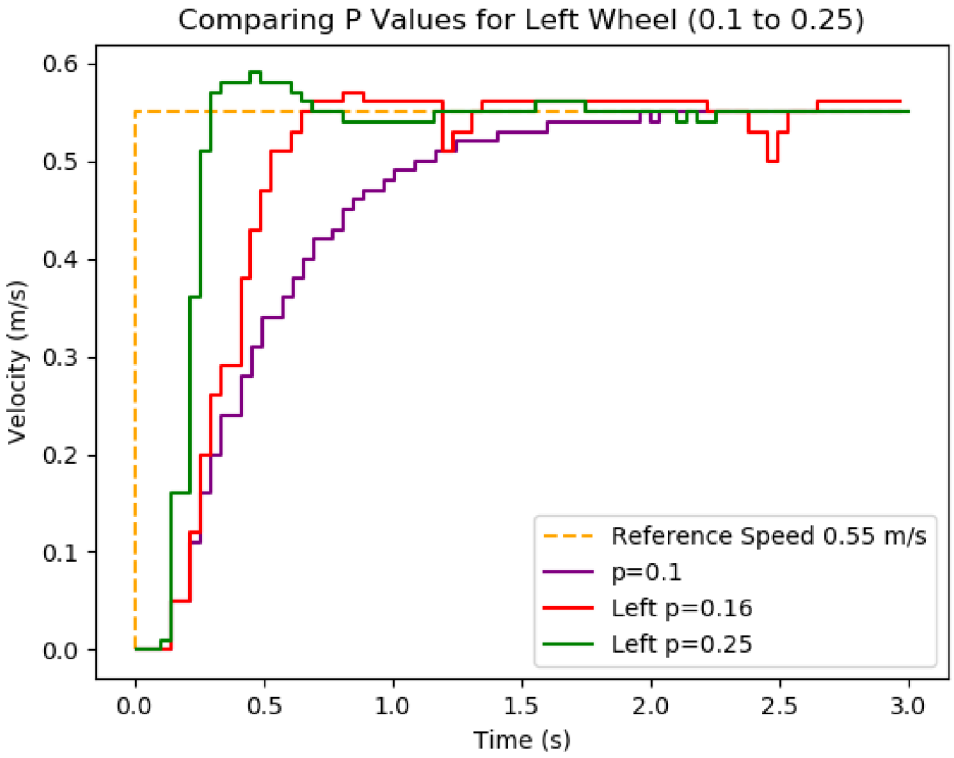


Figure 11: Comparing proportional gain for values of p between 0.1 and 0.25.

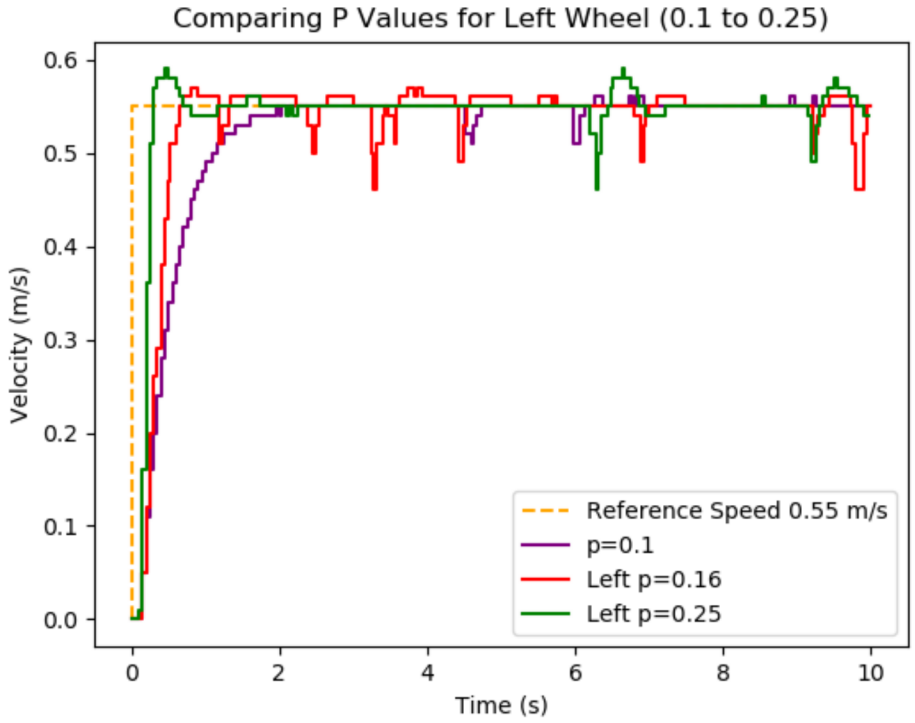


Figure 12: Extension of Figure 11.

Conclusion:

We overcame challenges throughout the prototyping process such as creating a functional design for wheel adaptors, gaining a better understanding of PID motor controls, and understanding how the computer controls communicate with the Raspberry Pi wirelessly. We designed two functional driver systems: one with keyboard control and the other with proportional gain control. We optimized our proportional gain constant to p =0.1, but we could increase this value in the future if we incorporate integral gain control. Another goal is implementing proportional gain control in conjunction with our keyboard control. It is unnecessary to read serial data in odom_pubisher.py and publish it for a different node (bot_testing.py) to read. If serial data is read in the same program that subscribes to keyboard control (subscriber_member_function.py), we can implement PID control and keyboard control together. In the future, we will also incorporate LiDAR detection and mapping and combine all of our programs into a single ROS2 package.

Appendix A:

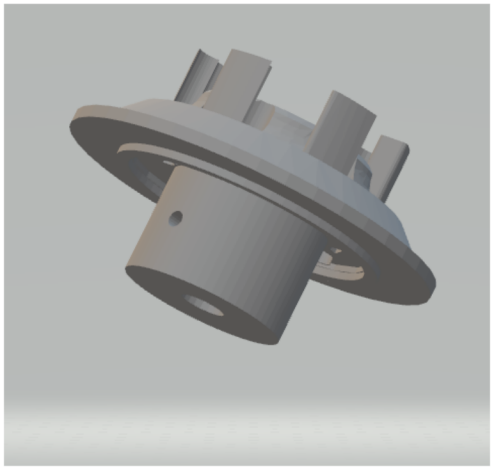


Figure A1: Image of the STL file used to 3D print the skateboard wheel mount.

References:

[1] Pololu Robotics and Electronics, " 12V 70:1 Metal Gearmotor," 70:1 Metal Gearmotor 37Dx70L mm 12V datasheet. https://www.pololu.com/product/4754/pictures.

[2] Pololu Robotics and Electronics, " Dual G2 High-Power Motor Driver," Dual G2 High-Power Motor Driver 24v18 Shield for Arduino datasheet. https://www.pololu.com/product/2518/pictures.

[3] Slamtec, "A1M8 Datasheet," RPLidar instructions and datasheet. https://www.digikey.jp/htmldatasheets/production/3265529/0/0/1/a1m8.html.

[4] Arduino, "Arduino Nano," Nano datasheet. https://docs.arduino.cc/hardware/nano.