Summary

The purpose of this project was to design and build a differential drive robot. To do so, we set two main files. A python code responsible for publishing the robot speed in ROS 2 topic with geometry twist msg and a subscriber code to remote control the robot.

Material list

- Raspberry pi 4(https://www.raspberrypi.com/products/raspberry-pi-4-model-b/)
- Arduino Nano Every Board (https://store-usa.arduino.cc/products/arduino-nano-every)
- Polulu motor driver shield(https://www.pololu.com/product/2518)
- Polulu Gear motor 4754(https://www.pololu.com/product/4754)
- 3s LiPo battery(https://www.amazon.com/dp/B07ZV73D5S)

Design

Figures 1 and 2 illustrated pictures of our robot view from the front and back. The motor drivers are located at the back of the robot. More parts are on the robot because our design and build considered all the parts needed for the projects of the semester. Our robot has a motor driver, 2 wheels, 2 caster wheels, a support in wood, the raspberry pi 4, the encoder, a raspberry pi camera, a robotic arm, a power bank and a lipo battery.

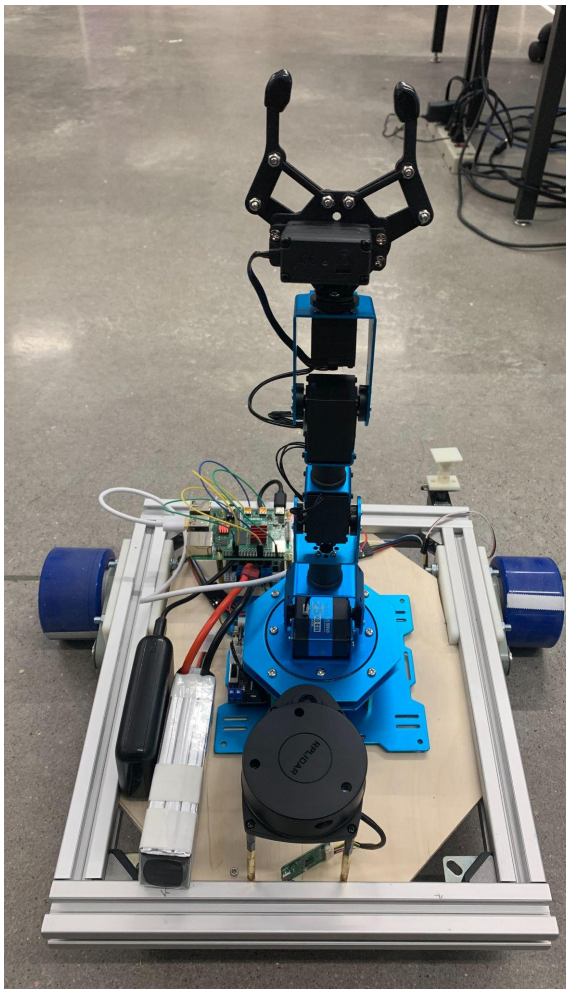


Figure 1. Robot viewed from front

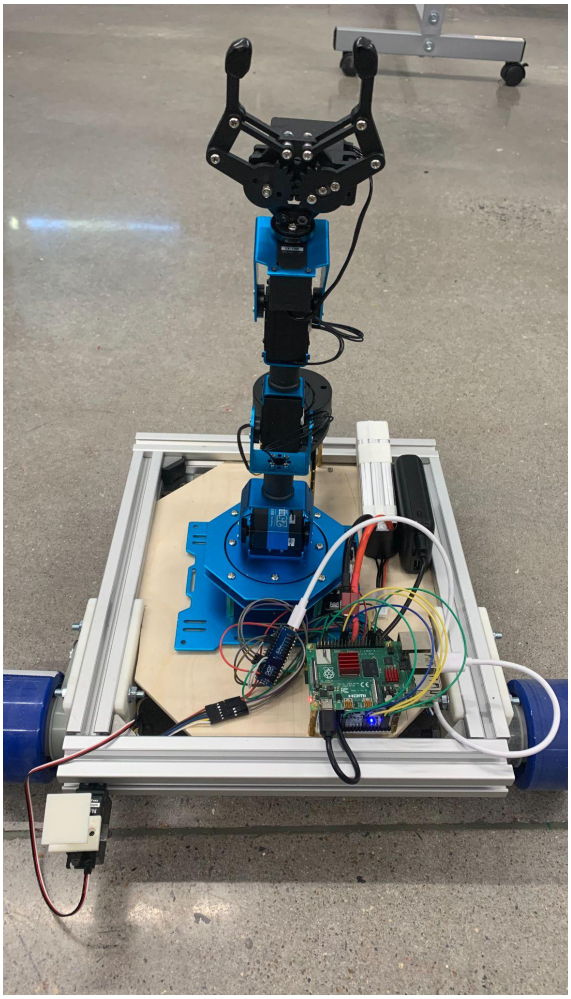


Figure2. Robot viewed from back

Figure 3 illustrates the disposition of the raspberry pi and the encoder.

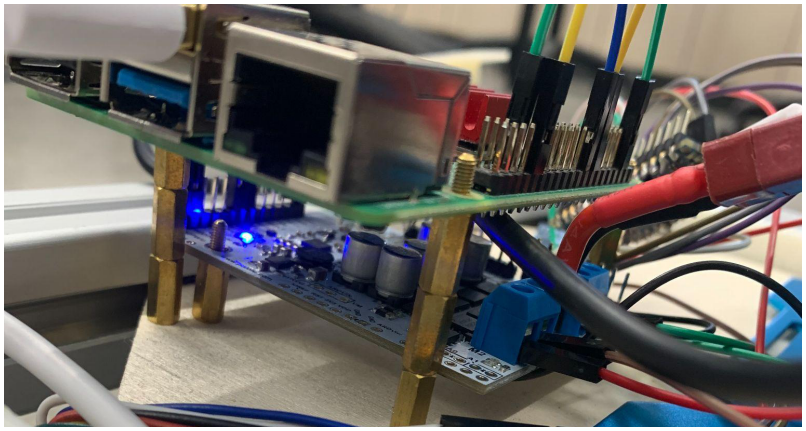


Figure 3. Raspberry pi and motor driver

Figure 4 shows the servo motor mounted to the raspberry pi. The camera would be mounted on the raspberry pi so we can have a 360-degree view of the field.

Figure 4. Servo motor (support of the raspberry pi camera).

Figure 5 shows the setup of our motor drivers and wheel to the chassis. In fact, the main wheel is connected to a gear that decreases the speed of our motor by ½.

Figure 5. Gear assembly

Software list

- Operating system

Ubuntu server 20.04: Ubuntu is a free Linux operating system. Ubuntu can be installed on a raspberry pi by using the pi imager to flash the card and install the server

- (ROS2)

ROS2 is a set of libraries and tools used to make robot applications. The Galactic version is the recommended newest version of ROS. For our projects, we used a simple publisher and subscriber node to communicate over the ROS geometry twist msgs (refer to https://docs.ros.org/en/foxy/Tutorials/Writing-A-Simple-Py-Publisher-And-Subscriber.html).

- Python libraries

GPIO Zero: it is the basic library in python to do physical computing. In our project we used the GPIO zero to initiate the motor pins and access them.
Rclpy: it is a python library used to interact with the ROS2 environment
geometry_msgs.msg: it is a python library used to provide messages about the geometric values. In the project we use the twist/ cmd/vel method to determine the actual angular and linear velocity of the robot.

Theory

We started our assignment by creating a subscriber to the twist geometry msgs so that we can remotely control our robot. We base our code mainly on writing to a subscriber from the ros2 galactic tutorial. Refer to the file named teleop.py in the GitHub repository for details. Figure 6 illustrates an extract of the code setting boundaries for the speed and direction of the robot. In fact, msg.linear.x represents the linear velocity of the robot set between -1 and 1 (depending on the PWM of the motor) and msg.angular.z represents the angular velocity of the robot also set between -1 and 1. Both the linear and angular velocities are input through the teleop keyboard. Figure 7 illustrates the interface of the command on the keyboard.

```
if abs(msg.linear.x)>1:
    msg.linear.x= 1.0
if abs(msg.angular.z)>1:
    msg.angular.z=1.0

if msg.linear.x>0:
    bot.forward(abs(msg.linear.x))
if msg.linear.x<0:
    bot.backward(abs(msg.linear.x))
if msg.angular.z>0:
    bot.right(abs(msg.angular.z))
if msg.angular.z<0:
    bot.left(abs(msg.angular.z))
if msg.angular.z==0 and msg.linear.x==0:
    bot.stop()
```

Figure 6. Extract of the code setting the speed and direction from the Teleop.

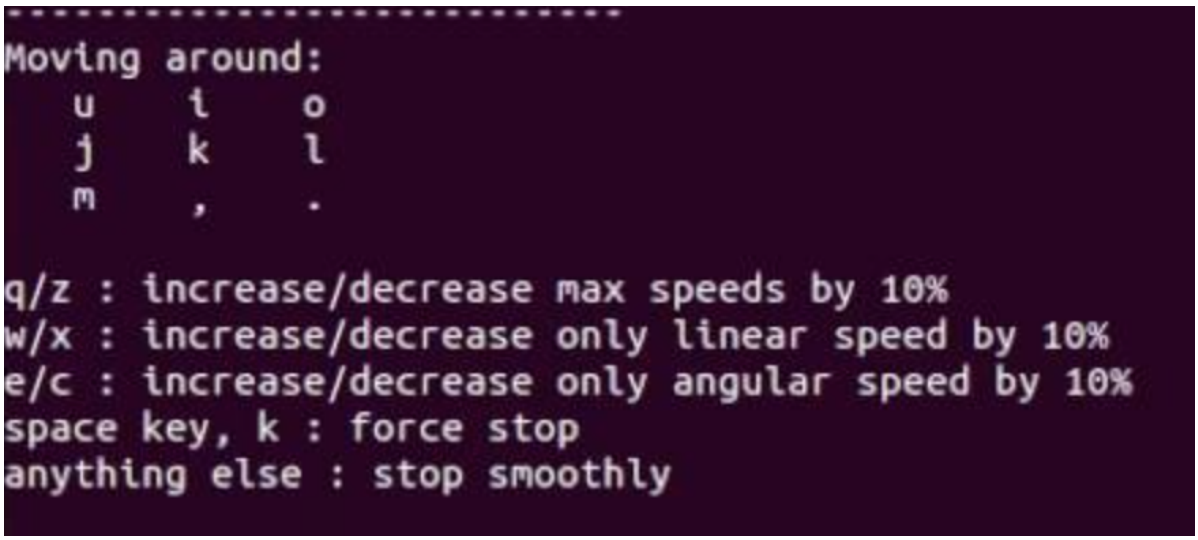


Figure 7. Interface of the teleop keyboard in ROS 2.

After the teleop keyboard control, we created a publisher node responsible for determining the actual odometry of the robot through the encoder. The encoder gets the counts per second from the nano Arduino and sends the information to the raspberry pi. We then use that count per

second to determine the actual speeds of the motor by using equation (1). Figure 8 shows how the publisher node is set up.

$$v = \frac{CPS \cdot 2 \cdot pi \cdot radius}{radius} \quad (1)$$

```
#main loop
class botlistener(Node):
    def __init__(self):
        super().__init__('bot_listen')

print('Hi from bot')
rclpy.init(args=None)
node= rclpy.create_node('bot_listen')
publisher= node.create_publisher(Twist, 'cmd_vel', 1)
msg= Twist()
bot_listen= botlistener()
node= rclpy.create_node('minimal_client')
```

Figure 8. Publisher node

The CPR stands for the count for revolution is determined by multiplying the gear ratio by the clicks per revolution of the motor. In our case, the CPR has to be multiplied by 2 because of the adjacent set of gears we have. However, this decreases the actual motor speed by the factor of 2. Refer to equation (2) for the formula of the CPR.

$$CPR = gear\ ratio \cdot \frac{clicks}{revolution} \cdot 2 \quad (2)$$

```
if ser.in_waiting > 0:
    line = ser.readline()
    if b'\xff' in line or b'\xfe' in line:
        continue
    speeds = line.decode('utf-8').rstrip().split(',')
    right_count = float(speeds[0])
    left_count = float(speeds[1])
    l_speed = left_count*2*np.pi*radius/cpr
    r_speed = right_count*2*np.pi*radius/cpr
```

Figure 9. Code determining the speed of the robot from the encoder count per s

Figure 10 illustrates a graph of the speed vs the time of our robot moving forward at a speed of 0.15m/s during the testing phase. We realize that as soon as the program starts the speed of the motors jumps from 0 to 0.15 m/s. Which means for higher velocities, the robot would behave similarly. In the long run, this is not beneficial to the motor because it may damage it. Also, let's consider the example of a car in a real-life situation. When we are driving, we do not immediately increase the speed of the car from 0 to 70mph or vice versa because it is dangerous, and the car will definitely break at some time. Furthermore, another challenge was getting to have the robot move in a straight line. We notice that as the motor goes forward, it tends to make a circle in the long run and that is because the two motors are not perfect clones of each other.

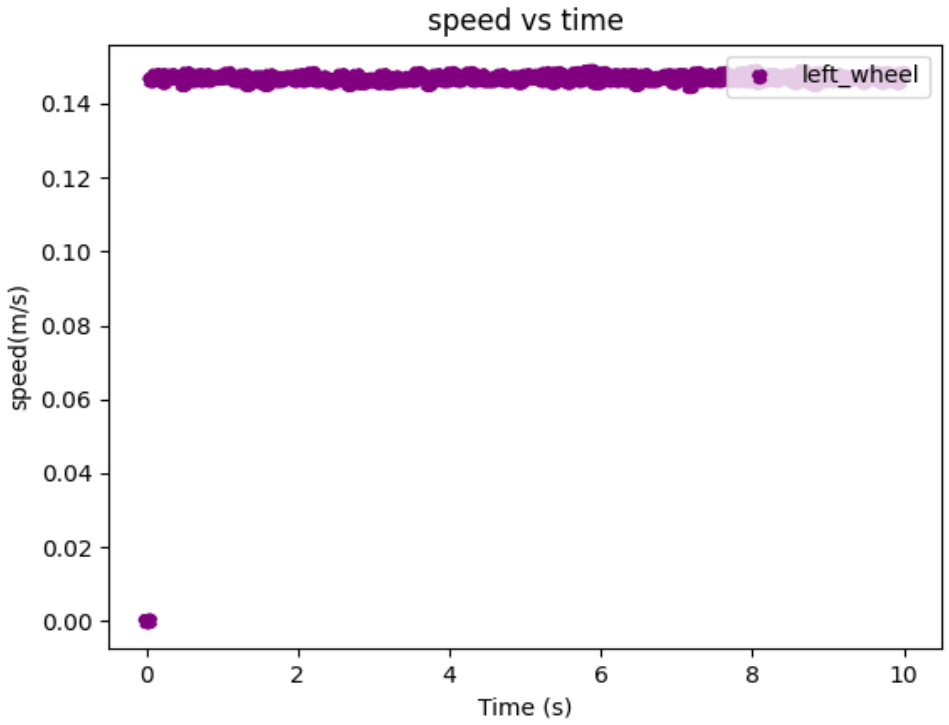


Figure 10. Speed vs time of our robot at a speed of 15cm/s.

Therefore, the best way to manage those challenges is by setting a PID control. PID control stands for proportional integral derivative. The PID control is a loop feedback controller that determines the difference between the desired and actual speed and uses the result to apply correction to the process. In our project we just used proportional control which takes use of a set constant which is proportionally multiplied by the difference in our actual and desired speed. This helps our robot to constantly adjust the speed of the wheels toward our set target speed. Below you can see the snippet of our code that takes care of the proportional control:

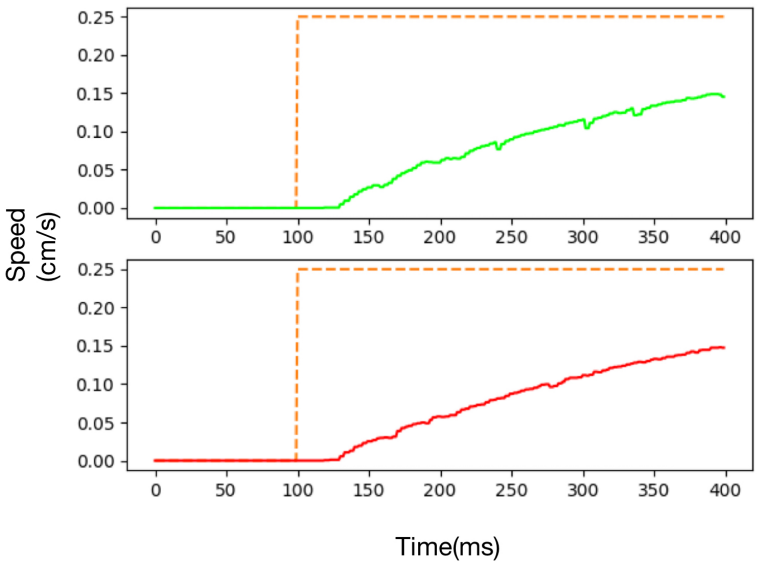
```
# set target velocity
if counter >= 100:  # at 1 sec
    target_speed = 0.25
# compute error
left_error = target_speed - l_speed
right_error = target_speed - r_speed

# compute dutycycle increment
left_dutycycle_inc = K_P * left_error
left_dutycycle += left_dutycycle_inc
if left_dutycycle > 1:
    left_dutycycle = 1
elif left_dutycycle < 0:
    left_dutycycle = 0
right_dutycycle_inc = K_P * right_error
right_dutycycle += right_dutycycle_inc
if right_dutycycle > 1:
    right_dutycycle = 1
elif right_dutycycle < 0:
    right_dutycycle = 0
# drive motors
bot.left_motor.forward(left_dutycycle)
bot.right_motor.forward(right_dutycycle)
sleep(.01)
```

Analyses

After establishing that our speed control worked, it was time to do some actual testing in order to determine an adequate proportional constant. Different constants will greatly impact the robot's driving performance and below you can see graphs of some of the different values we tested:

- K_P= 0.01

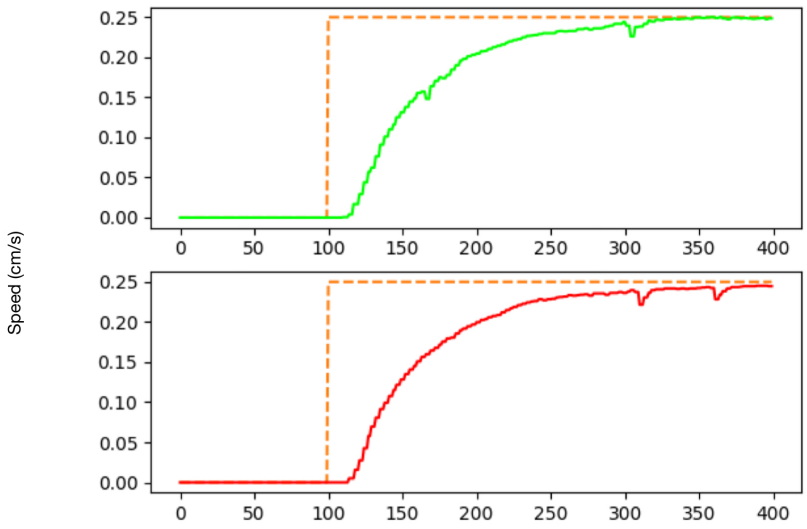


In the graph above we used a constant of 0.01 and set the target speed to 0.25 seconds. We then let the robot accelerate for 3 seconds (300 milliseconds). As you can see from the graph the robot

accelerates slowly and does not reach the target speed within the given time. This means that we will have to increase the set constant.

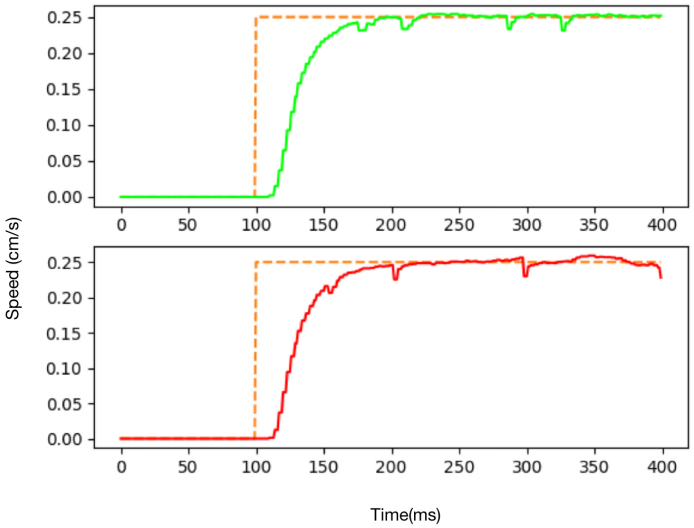
Therefore, we next tried a value of 0.05 and below you can see the results. Obviously, this looks better, and the robot actually reached the target speed within 2 seconds, but we can still do a lot better.

- K_P= 0.05



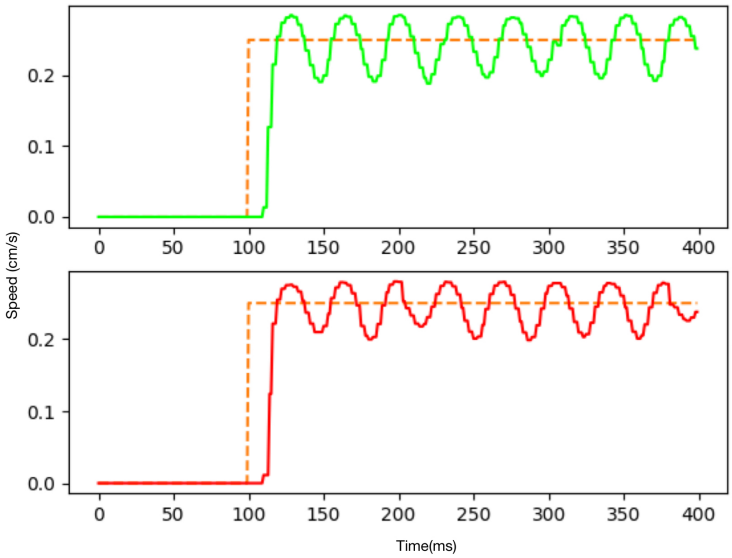
So next we doubled the constant up to 0.1. As you can see in the graph below, the performance increased, and the robot now reached the target speed in about 0.7 seconds while remaining at a stable speed.

- K_P = 0.1



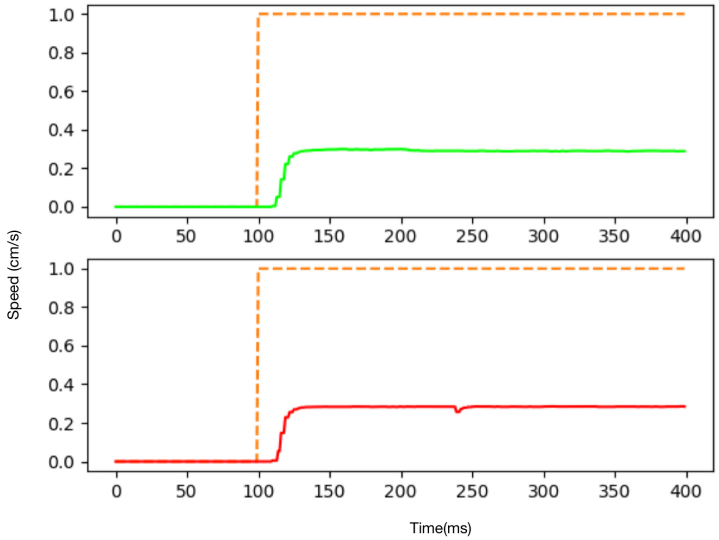
We then continued and increased the constant up to a value of 1. Since the constant is now bigger, it resulted in our robot overshooting and undershooting the target speed, which gave us this oscillation about the target speed. This is not wanted behavior for us because we want the robot to quickly accelerate, but then keep a stable speed. As a result, the optimal constant for us at this stage will lay somewhere between 0.1 and 1.

- K_P= 1



Next, we wanted to determine our maximum speed. To do this we set the target speed to an unrealistic goal of 1 m/s and tracked the progress. The results are plotted below:

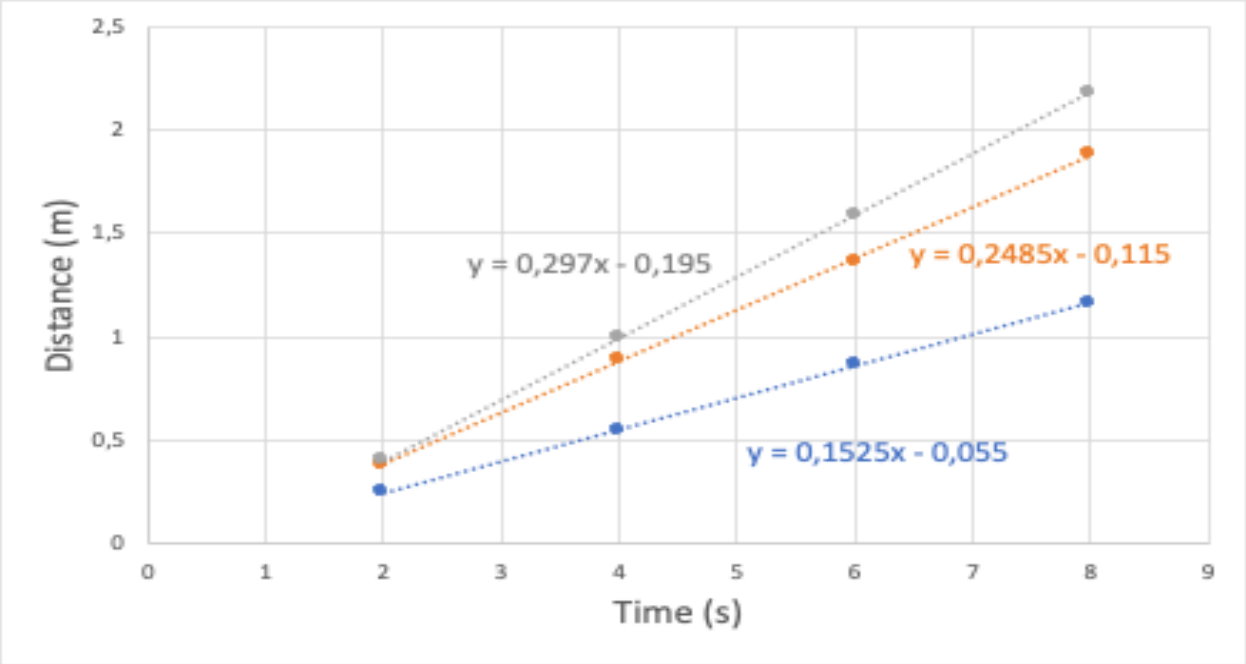
- Max Robot speed



As expected, our maximum speed would come in lower than 1 m/s, and the result showed that the maximum speed is actually under 30 cm/s.

We also wanted to confirm that our graphed data matches up with the real speed our robot is traveling at. To do this we used a stopwatch and measuring tape to get different data points. We then graphed the results below for 3 different speeds. The gray line indicates maximum speed, the orange line represents a target speed of 25 cm/s, and the blue line represents a target speed of 15 cm/s.

- Actual velocity



As you can see, the slope of the line represents the actual speed in meters per second and it matches up great with what we set the target speed as.

Usage

To use the robot, the programmer has to first install the operating systems. First, install Ubuntu on the raspberry pi, then ros2 (https://github.com/linzhangUCA/robotics1-2021/wiki/Install-ROS2-Galactic-on-Raspberry-Pi-3). Also the computer has to be connected to the ubuntu server of the raspberry pi through SSH (see the details on https://github.com/linzhangUCA/robotics1-2021/wiki/Remote-Operation). Also, the GPIO should be configured on Ubuntu so that the robot can run the code(https://github.com/linzhangUCA/robotics1-2021/wiki/GPIO-Configuration-on-Ubuntu-Server-20.04). After completing the preliminary request, the user should be able to copy the repository to the raspberry pi. Make sure to upload the encoder.cpp file to the encoder through Arduino first. Then run simultaneously the teleop.py, the proportionnal_step.py as well as the teleop twist keyboard. The robot could then be controlled with the keyboard with the publisher displaying the actual odometry of the robot.

Conclusions

To conclude we can say that the project has been a success so far. We have obtained our goal of being able to control the robot remotely through the Teleop_Twist_Keyboard, we have achieved a functional proportional controller that results in accurate and stable speed readings. Going forward we still need to incorporate an integral controller in order to improve further on our speed control. This will make our robot able to accelerate faster, without creating the oscillations. In addition to this, we will investigate design changes that will help us obtain a faster maximum speed.

Sources:

- https://docs.ros.org/en/galactic/Tutorials/Writing-A-Simple-Py-Publisher-And-Subscriber.html
- https://mjwhite8119.github.io/Robots/pid-control