Checkpoint # Report

[EECN30169] Mobile Robot 2024

Student ID: 313605019 **Name:** 方敏 **Date:** **2024/10/18**

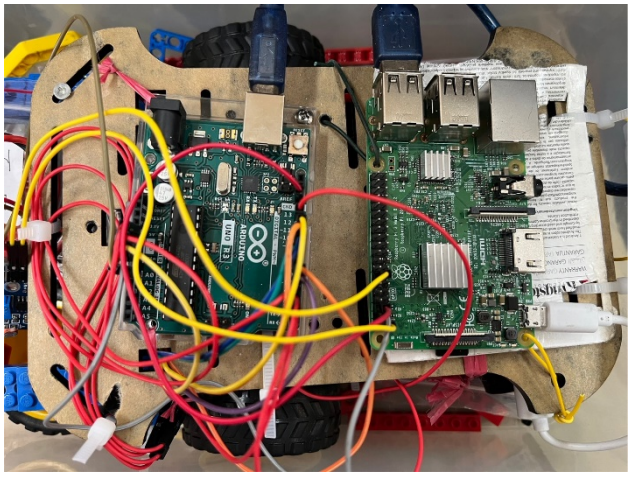
1. Purpose:

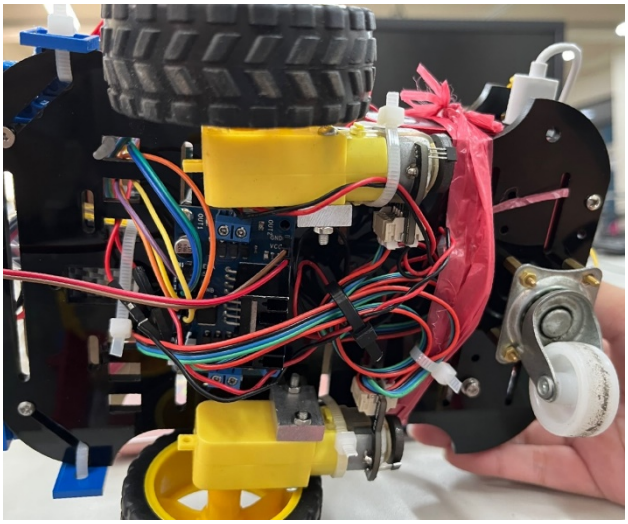
The purpose of this checkpoint is to make sure we can control the motion of DC motors by using PWM with Raspberry Pi and Arduino.

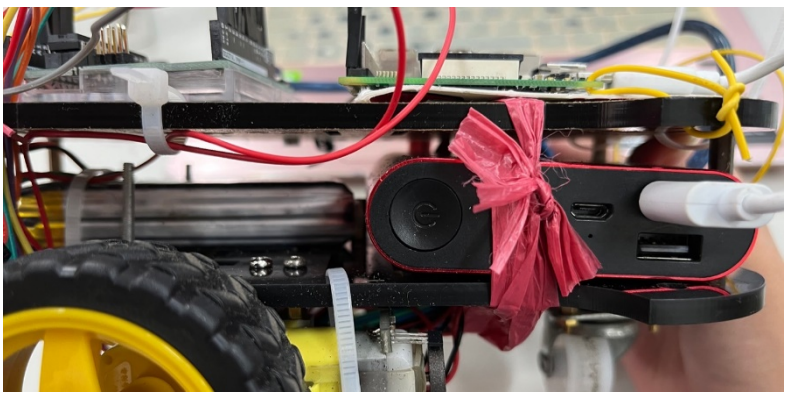
2. Description of Design:

Assembling:

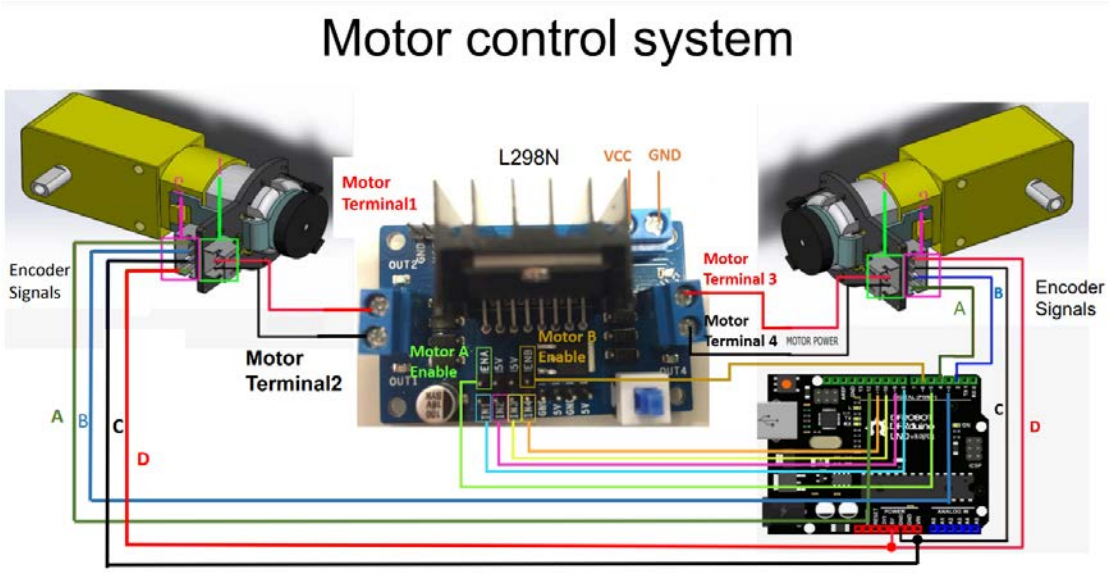
First, our task is to assemble the robot. Our design concept is to place the Arduino board and Raspberry Pi on the top layer, position the battery and power bank between two wooden boards, and install the motors and the L298N motor driver on the bottom layer.







Wiring:



Next, we connected the two motors with the L298N motor driver and the Arduino according to the above wiring diagram. A key difference is that, due to insufficient 5V pins on the Arduino, we used the 5V from the Raspberry Pi. Additionally, remember to ensure a common ground connection between the Raspberry Pi, Arduino, L298N, and the battery. The Raspberry Pi also powers the Arduino by connecting their 5V pins. Finally, we connected the L298N and the battery.

Coding:

Raspberry Pi:

We set up a node called 'cp2' to publish the 'control' topic. This topic carries two messages: the speed of the left wheel and the speed of the right wheel, both of which can be freely input by the user.

Arduino:

We first create a subscriber to monitor the 'control' topic. Upon receiving a value, it triggers a callback function that stores the value into 'Setleft' and 'Setright' for use as PID settings. Additionally, the function configures the motor's direction based on the received input.

```
// control subscriber
void controlCallback(const std_msgs::Int16MultiArray& msg) {
  // Set left control
  Setleft = abs(msg.data[0]);
  // Set right control
  Setright = abs(msg.data[1]);
```

```
  // Set left motor direction
  if (msg.data[0] >= 0) {
    digitalWrite(motorA1, LOW);
    digitalWrite(motorA2, HIGH);
  } else {
    digitalWrite(motorA1, HIGH);
    digitalWrite(motorA2, LOW);
  }
  // Set right motor direction
  if (msg.data[1] >= 0) {
    digitalWrite(motorB1, LOW);
    digitalWrite(motorB2, HIGH);
  } else {
    digitalWrite(motorB1, HIGH);
    digitalWrite(motorB2, LOW);
  }

  // If same speed need straight line correction
  if (msg.data[0] == msg.data[1] && msg.data[0] != 0) {
    need_correction = true;
  } else {
    need_correction = false;
  }

  // Reset cumulative distance
  encoderA_accumulation =0;
  encoderB_accumulation =0;
}
```

We use PID to control the rotation of the two wheels, through the PID_v1.h library. Note: PID_v1.h should put into the folder called “libraries” which is in Arduino.

```
/*-----Define PID parameters-----*/
double Kpleft=24.0, Kileft=0.05, Kdleft=0.2;
double Kpright=38.0, Kiright=0.40, Kdright=0.80;
double Setleft=0.0, Setright=0.0, Inputleft=0.0, Inputright=0.0, Outputleft=0.0, Outputright=0.0;
PID PIDleft(&Inputleft, &Outputleft, &Setleft, Kpleft, Kileft, Kdleft, DIRECT);
PID PIDright(&Inputright, &Outputright, &Setright, Kpright, Kiright, Kdright, DIRECT);
```

The setup function will carry out several tasks:

1. **Configuring hardware interrupts:** Pins 2 and 3 on the Arduino are connected to the B phase of the left and right wheel encoders, respectively. Each time the encoder detects a change, the corresponding interrupt function is triggered to compute the rotations.

2. **Defining output limits:** This ensures that the output values stay within the range of 0 to 255, complying with PWM output standards.

```
// Set ISR
attachInterrupt(1, encoderAInt, CHANGE);
attachInterrupt(0, encoderBInt, CHANGE);

// Set PID mode
PIDleft.SetMode(AUTOMATIC);
PIDright.SetMode(AUTOMATIC);

// Set PID limit
PIDleft.SetOutputLimits(0, 255);
PIDright.SetOutputLimits(0, 255);
```

```
void encoderAInt()
{
  encoderAPos++;
}

void encoderBInt()
{
  encoderBPos++;
}
```

In the main loop, the encoder readings are typically used as the input for the PID controller, which then calculates the corresponding output. This output is subsequently applied as the motor's PWM signal. However, this approach alone is insufficient to ensure consistent straight-line movement.

```
void loop() {
  // Read from encoder
  Inputleft = encoderAPos;
  encoderAPos = 0;
  Inputright = encoderBPos;
  encoderBPos = 0;

  // Calculate moving distance
  encoderA_accumulation += Inputleft;
  encoderB_accumulation += Inputright;
```

```
// Motor PWM control
analogWrite(motorAPWM, round(Outputleft));
analogWrite(motorBPWM, round(Outputright));
```

To ensure straight-line movement, both wheels need to rotate at the same speed and cover equal distances as closely as possible. To achieve this, I implemented code that adjusts the wheel speeds dynamically: the wheel on the shorter path speeds up, while the wheel on the longer path slows down. This approach uses a control factor KcK_cKc based on the difference between the two paths, with the correction accumulating over time. If the path difference continues to grow, the adjustments become progressively more aggressive, ensuring better alignment.

```
// Calculate moving distance
encoderA_accumulation += Inputleft;
encoderB_accumulation += Inputright;

// Straight line correction
if (need_correction) {
  distanceDifference = abs(encoderA_accumulation - encoderB_accumulation);
  speed_correction = distanceDifference * kc;
  if (encoderA_accumulation > encoderB_accumulation) {
    if (distanceDifference > last_distanceDifference) {
      if (speed_correction > Setleft) speed_correction = Setleft;
      Setleft -= speed_correction;
      Setright += speed_correction;
    }
  } else if (encoderA_accumulation < encoderB_accumulation) {
    if (distanceDifference > last_distanceDifference) {
      if (speed_correction > Setright) speed_correction = Setright;
      Setleft += speed_correction;
      Setright -= speed_correction;
    }
  }
  last_distanceDifference = distanceDifference;
}
```

In the program, we incorporated a publisher with the topic 'output' specifically for debugging. This allows us to monitor the encoder values of the left and right wheels, the PWM output, and the distances traveled by both wheel tracks, ensuring accurate performance tracking and troubleshooting.

```
#if EN_DEBUG
  output_msg.data_length = 6;
  output_msg.data = new int16_t[6];
  nh.advertise(output_pub);
#endif
}
```

```
#if EN_DEBUG
  if((Setleft != 0) && (Setright != 0)) {
    output_msg.data[0] = Inputleft;
    output_msg.data[1] = Inputright;
    output_msg.data[2] = (Outputleft + 0.5);
    output_msg.data[3] = (Outputright + 0.5);
    output_msg.data[4] = encoderA_accumulation;
    output_msg.data[5] = encoderB_accumulation;
    output_pub.publish(&output_msg);
  }
#endif
```

3. Result

• Move forward.

If the input speed of left wheel = right wheel and both value > 0, the robot will move forward.

• Move backward.

If the input speed of left wheel = right wheel and both value < 0, the robot will move backward.

• Turn right.

If the input speed of left wheel > right wheel and both value > 0, the robot will turn right.

• Turn left.

If the input speed of left wheel < right wheel and both value > 0, the robot will turn left.

• How straight the robot can move when moving forward

We adjusted the parameters Kp, Ki, Kd, and Kc to ensure that the robot moved steadily forward. At last, during the demo, the car's path deviated by only 6 centimeters.

Discussion

Problem 1:

Why are the Kp, Ki, and Kd parameters for the left and right wheels so different?

```
/*-----Define PID parameters-----*/
double Kpleft=24.0, Kileft=0.05, Kdleft=0.2;
double Kpright=38.0, Kiright=0.40, Kdright=0.80;
double Setleft=0.0, Setright=0.0, Inputleft=0.0, Inputright=0.0, Outputleft=0.0, Outputright=0.0;
PID PIDleft(&Inputleft, &Outputleft, &Setleft, Kpleft, Kileft, Kdleft, DIRECT);
PID PIDright(&Inputright, &Outputright, &Setright, Kpright, Kiright, Kdright, DIRECT);
```

Explanation 1 :

This is because our right wheel is quite loose and particularly unstable. The robot can travel well in the first meter but will start to veer to the right in the next meter. Therefore, we set the robot to veer slightly to the left in the first meter to balance out the rightward deviation caused by the instability of the right wheel.

Problem 2:

Why do we need to stop running “rosrun rosserial_python serial_node.py /dev/ttyACM0” and “rosrun lab2_pkg cp2_node” every time we modify the PID?

Explanation 2:

Because we wrote the publisher on the Raspberry Pi and the subscriber on the Arduino. So every time we modify the PID, we need to stop these processes since they have established a connection between the Arduino and the ROS nodes and are using the hardware ports for communication. When we modify the PID or the Arduino program, we need to recompile and upload the new code to the Arduino, which requires disconnecting the existing connections to release the port. And requires ROS nodes be restarted as long as there’s new changes.

Problem 3:

When we finished connecting the wires and writing the code, we found that when we input a positive speed value, the wheels moved backward, and when we input a negative speed value, the wheels moved forward. How did we resolve it?

Explanation 3:

There’s two ways that we can resolve this problem, first is to switch IN1 and IN2 connections in L298N and switch IN3 and IN4 connections in L298N. Second is to rewrite the code to:

```
// Set left motor direction
if (msg.data[0] >= 0) {
  digitalWrite(motorA1, HIGH);
  digitalWrite(motorA2, LOW);
} else {
  digitalWrite(motorA1, LOW);
  digitalWrite(motorA2, HIGH);
}
// Set right motor direction
if (msg.data[1] >= 0) {
  digitalWrite(motorB1, HIGH;
  digitalWrite(motorB2, LOW);
} else {
  digitalWrite(motorB1, LOW);
  digitalWrite(motorB2, HIGH);
}
```

We choosed the first way.

In this checkpoint, I learned a lot about hardware control and software integration for robots. I am grateful for the guidance of the professor and teaching assistants. I hope that next checkpoint will go even more smoothly.