MRSD Project Course

Team I – AIce

Autonomous Zamboni Convoy

Individual Lab Report 1

Team

Rathin Shah

Nick Carcione

Yilin Cai

Jiayi Qiu

Kelvin Shen

Author

Kelvin Shen

Feb 6, 2022

Table of Contents

Individual Progress

Sensor and Motors Lab

In this lab, we are required to demonstrate the use of a GUI and sensor input to an Arduino board to control motors. My responsibility in this lab includes setting up a screen-based GUI to interface between Arduino and ROS, as well as integrating every team member's motor and sensor control code into a single Arduino program. Our final deliverable is shown in Figure 1.

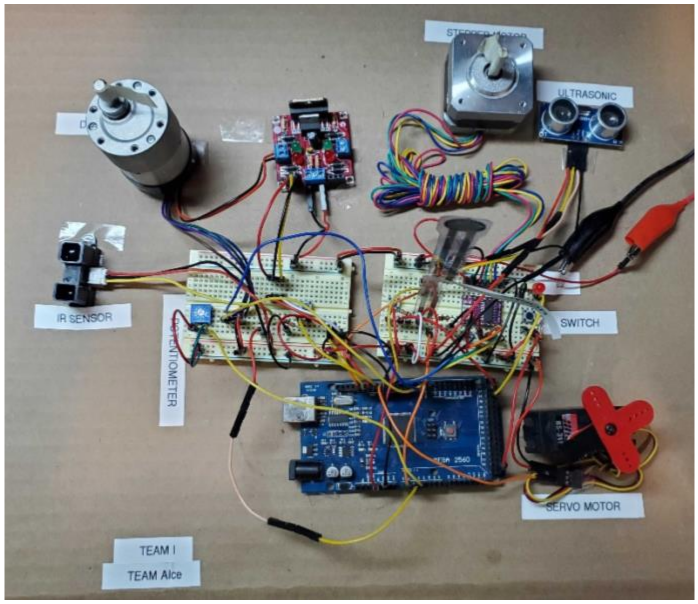

Figure 1. Final Product

GUI

There are a few options to implement a screen-based GUI for this lab. I chose ROS because of the following reasons:
(1) Every member in our team is getting familiar with ROS over the past winter break and now we are all in the stage when a lot of practice is needed to solidify what we've learned.
(2) ROS takes up a significant part in the upcoming Programming Familiarization assignment, this is a great opportunity to get hands-on experience with ROS topics.
(3) The other alternatives won't be useful for our capstone project as we will definitely use ROS to be the major platform.
(4) ROS comes with a handful of centralized GUI tools such as RViz.

Therefore, I implemented our GUI based on RViz to visualize the three motors, as well as RQT plots to visualize the readings from the sensors. First of all, to establish the communication between ROS and Arduino, I used the "rosserial_arduino" package, which allows me to connect to an Arduino to the ROS runtime graph, all through a NodeHandle initialized inside the Arduino code. Then we can publish to a topic or subscribe to a topic just as how we do in roscpp.

For our lab, I created a publisher that publishes the readings from our sensors (ultrasonic sensor, Flex force sensor, IR sensor, and potentiometer) to ROS so that we can visualize those readings in RQT plots. There are existing messages that can satisfy our needs but the naming of the data types in those messages is confusing. Therefore, we decided to create our own message, named "sensors.msg" with the structure shown in Figure 2. In our Arduino program, we update each variable of the message whenever we analogRead the reading from a sensor, and we publish the message through the publisher during each loop.

```
uint16 button
float64 pot
float64 flex
float64 ir
float64 ultra
```

Figure 2. Contents of sensors.msg

To manipulate the state of the motors via the GUI, I override the sensor feedback loop and inserted control values to motors directly from ROS. To do this, I created a URDF file for three motors, each being consisted of a base link, a shaft link, and a revolute joint between two links. We can now control the angles of three motors by using the sliders inside Joint State Publisher GUI from ROS, while visualizing the angles by opening the URDF file inside RViz. As shown in Figure 3, I have drawn three simplified models to represent our three motors, servo motor in blue, stepper motor in black, and the DC motor in white. In the Joint State Publisher GUI, servo_angle specifies the angle of the servo motor, stepper_angle specifies the angle of the stepper motor, dc_angle specifies the position of the DC motor (indicating position control in the motor control code), and dc_velocity specifies the velocity of the DC motor (indicating velocity control in the motor control code). Given these four values from the GUI, I created the subscriber in our Arduino program, which listens to the topic "joint_states" and invokes callback to actually send control values or commands to the motors. Inside the callback function, angles retrieved from sensor_msgs:JointState are converted to motors' inputs respectively. For example, servo_angle is converted to degrees as input to the servo motor, stepper_angle is proportionally converted to steps as input to the stepper motor, and dc_angle and dc_velocity is converted to PWM as input to the DC motor via the corresponding position or velocity control.

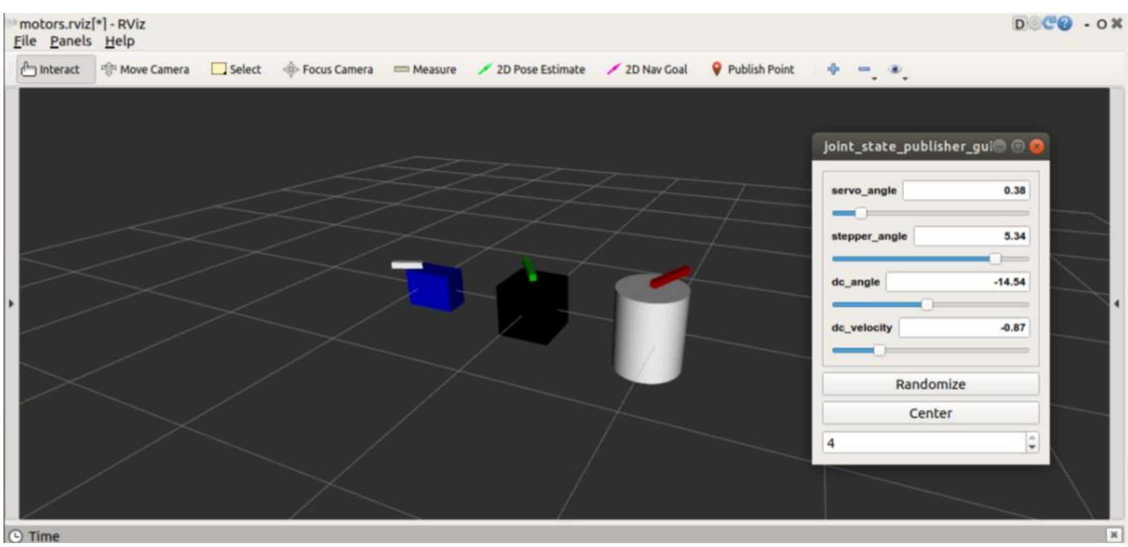

Figure 3. Joint State Publisher GUI that specifies angle or velocity of each motor

To plot the sensor readings, I used the Plot plugin in RQT. By echoing the topic where the custom sensors message is published, I plotted the button state along with four sensors' readings in RQT. The final RQT window (with Rviz embedded) is shown in Figure 4. Screen-based GUI for Sensors and Motors Lab. The five plots around the Rviz window correspond to the four sensors' readings plus the state of the button that decides motor control by either the joint state publisher or the sensor output.

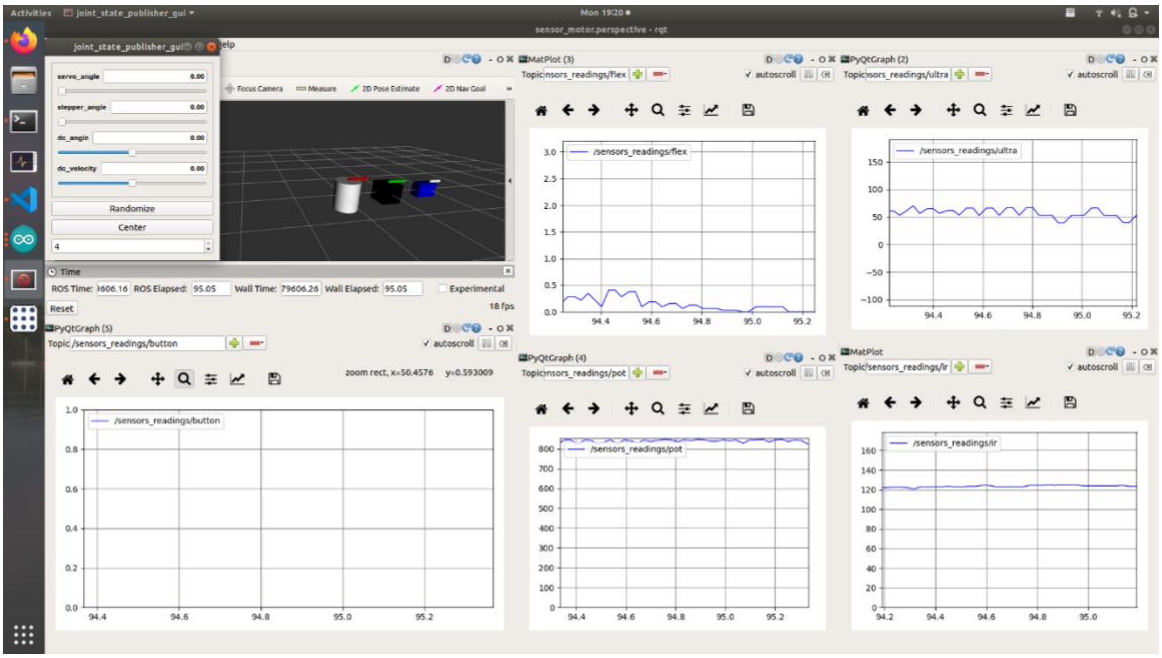

Figure 4. Screen-based GUI for Sensors and Motors Lab

Integration

I integrated three Arduino programs (each controlling a motor with one or two sensors) and my own GUI program into a single one. I managed to understand the logic behind each program so that there was not any collision of variables or functions. During integration, I added a program for one motor to GUI and tested it thoroughly before adding another one, which was actually the time when we found most errors and spent dozens of hours debugging the hardware. That is, each motor control program would work without issues if run alone, but when integrated with other programs, problems frequently occurred due to memory constraint on Arduino UNO, port collisions (Pin 9 and 10), and malfunction of the serial monitor (if TX and RX pins are plugged in when uploading). Further details are explained in next section.

Autonomous Zamboni Convoy Project

As the perception lead for our capstone project, I've been focusing on building the perception stack on ROS since the beginning of the winter break. Over the break, I got myself familiar with ROS by taking several online courses, including ROS basics in Python, URDF for Robot Modeling, TF ROS, ROS Navigation, ROS Perception, OpenCV for ROS, ROS Control, and

ROS Autonomous Vehicles. Some of them are very helpful in that the example packages can be directly used in our project. Starting from this semester, I worked with Yilin to set up the environment to simulate the perception stack. After Yilin managed to run a Zamboni vehicle model in Gazebo, I generated an ArUco board (a board of ArUco markers) with OpenCV, and used Blender to create a wall with the ArUco board printed on it, which was then inserted inside Gazebo, right in front of the Zamboni model. In parallel, I also finished the Programming Familiarization assignment beforehand because Part 3 is about detection of AprilTags, which would be very helpful to our project since I will implement detection algorithm to estimate the pose of the leader Zamboni based on the board of ArUco markers behind it.

Challenges

Sensor and Motors Lab

The greatest challenge to me during the completion of this lab must be understanding the mechanism behind the communication between ROS and Arduino. The package rosserial_arduino is prone to glitches, such as not being able to generate the header file for my custom message even though I followed the correct tutorial. There was not too much support from ROS on this package and hence the tutorials are outdated. Plus, the function definition of the callback for my subscriber must be located before the initialization of the subscriber, which took me a lot of time debugging as it was against the coding rules in Arduino. The other major challenge that's related to my GUI implementation must be the memory issue with Arduino UNO. The three lines of code that initialize a NodeHandle, a publisher, and a subscriber, take 70% of the memory on UNO, which directly causes problems such as Arduino not responding when uploading code, or even the IDE crashing. However, these initializations are necessary for the GUI and are not possible for any further optimization. As a result, our team had to request an Arduino MEGA board to avoid such memory problems.

Autonomous Zamboni Convoy Project

Spawning a wall of ArUco markers (described in the previous section) with an appropriate dimension into Gazebo was the largest challenge regarding my progress on our project so far. I took advantage of the packages from the ROS courses I've learned during the break, which provides launch scripts that spawn different models into an existing world in Gazebo. However, the markers on the board were not large enough relative to the size of a Zamboni vehicle. Therefore, I had to recreate an URDF model that takes in a mesh file of a board of ArUco markers with the correct dimensions (large enough to be seen by the following Zamboni). In addition, I was having a difficult time setting up the correct working environment on my Ubuntu. Because my Ubuntu was Bionic version, I installed Melodic as my ROS distribution but Melodic only worked with Python 2.7, which caused a lot of environment issues when I tried to install the correct version of OpenCV for my rospy scripts. I also re-installed Melodic, only to find out that even my previous packages did not work correctly. After several hours searching solutions to each error message I encountered, I finally set up the environment prepared to work with the perception stack for our project.

Teamwork

Sensors and Motors Lab

We divided the work according to the instructions on Canvas, i.e. each person working with one of the sensor to control a motor and the remaining person working on GUI. Therefore, the four following members on our team worked on sensors and motors control while I worked on GUI and integration.

- Nick worked on using the IR sensor to control the DC motor with PID control, including controlling the motor via either position input or velocity input.
- Rathin worked on using the potentiometer to control the servo motor.
- Jiayi worked on the flex force sensor and the stepper motor.
- Yilin worked on the ultrasonic sensor and the stepper motor. He also helped build the entire circuit that integrated all members' motors and sensors.

Autonomous Zamboni Convoy Project

In our project, we worked on different topics separately according to the schedule shown in our CoDRR.

- Nick worked on methods and packages to fuse wheel encoder and IMU data so as to obtain accurate velocity estimation of the follower.
- Rathin worked on the steering and velocity controller in MATLAB Simulink, specifically for the Ackermann geometry, as well as the interface between Simulink and ROS.
- Jiayi worked on building the simulation environment for our Zamboni Convoy inside Gazebo, which includes a synthetic ice rink for the ice hockey game.
- Yilin worked on setting up the URDF for the Zamboni vehicle based on an Ackermann model, which can be smoothly controlled through keyboard in simulation.

Plans

I would keep on working on the perception stack in ROS. First I shall get the correct pose estimate from the wall of a board of ArUco markers using the camera on our Zamboni model in Gazebo. Then I would attach that ArUco board to the rear of another Zamboni and test the pose estimation algorithm when both the leader and the follower are moving. In parallel, I will work on the Intel RealSense camera and integrate it into our ROS environment.

Quiz

1. ADXL335 Datasheet
    a. What is the sensor's range?

    $$\pm 3.6\ g\ \ (\pm 3g \text{ at minimum})$$

    b. What is the sensor's dynamic range?

    $$6\ g \text{ at minimum}$$

    c. What is the purpose of the capacitor CDC on the LHS of the functional block diagram on p. 1? How does it achieve this?
    The capacitor is there to decouple noise from the power supply.

    d. Write an equation for the sensor's transfer function.

    $$V_{out} = 0.3\left(\frac{V}{g}\right)a + 1.5V$$

    e. What is the largest expected nonlinearity error in g?

    $$0.3\% \times 6g = 0.018g$$

    f. What is the sensor's bandwidth for the X- and Y-axes?
    1600 Hz if there's no external filter.

    g. How much noise do you expect in the X- and Y-axis sensor signals when your measurement bandwidth is 25 Hz?

    $$150 \times \sqrt{25 \times 1.6} = 948.7\ \mu g$$

    h. If you didn't have the datasheet, how would you determine the RMS noise experimentally?
    Assuming no noise from the power supply, we can determine the RMS noise experimentally by placing the accelerometer on a static surface and recording its readings over a very long duration. Then we can approximate the RMS noise with the classic equation for a root mean square, $\sqrt{\frac{1}{n}\Sigma_i z_i^2}$ where z_i is each reading.

2. Signal Conditioning
    a. Filtering
        i. If a moving average filter uses a large window size, the average value will not be representative enough for the latest value because of the delay.
        if a moving average filter uses a small window size, the information will be more relevant but in turn will allow noise to be read.
        ii. Median filter is computationally expensive since a selection or sorting algorithm is required to find the median.
        Median filter works poorly with continuous noise or multiple outliers if the window size is not large enough.
    b. Opamps
        i. Your uncalibrated sensor has a range of -1.5 to 1.0V
        V_1 is reference voltage and V_2 is input voltage.

$$V_{out} = V_{in}\left(1 + \frac{R_f}{R_i}\right) - V_{ref}\left(\frac{R_f}{R_i}\right)$$
$$5 = 1\left(1 + \frac{R_f}{R_i}\right) - \frac{V_1 R_f}{R_i}$$
$$0 = -1.5\left(1 + \frac{R_f}{R_i}\right) - \frac{V_1 R_f}{R_i}$$
$$\therefore \frac{R_f}{R_i} = 1, V_1 = -3V = V_{ref}$$

ii. Your uncalibrated sensor has a range of -2.5 to 2.5V
In this case we can get $\frac{R_f}{R_i} = 0$ using the same approach as above (no matter if it's V_1 as reference voltage and V_2 as input voltage or the other way around). Therefore, it's not possible to calibrate the sensor.

3. Control
   a. If you want to control a DC motor to go to a desired position, describe how to form a digital input for each of the PID terms.
      i. Proportional: we can read the positional output from a motor encoder which is then subtracted from the desired position to get the position error as the input into the proportional controller.
      ii. Integral: we sum the position errors (described above) over each timestep as the input into the integral controller.
      iii. Derivative: we divide the position error obtained as above by the time difference between every two timesteps to get the speed of change in position error, which is then fed into the derivative controller.
   b. If the system you want to control is sluggish, which PID term will you use and why?
      I would increase the proportional gain to reduce the rising time so that the system can be more responsive.
   c. After applying the control in the previous question, if the system still has significant steady-state error, which PID term will you use and why?
      I would use or increase the integral gain, which will record the sum of all the steady state errors over time so that it can reach the desired position more quickly.
   d. After applying the control in the previous question, if the system still has overshoot, which PID term will you apply and why?
      I would increase the derivative term because it would calculate and predict if the system will respond too fast to the error so that it can slow down the rate of error reduction and hence increase damping.

Arduino Program

Note: the definitions of helper functions that are used for sensor interfacing (hence unrelated to the mechanism of the GUI) are omitted. These can be found in other members' ILRs.

```
#include <Servo.h>
#include <Encoder.h>

#include <Arduino.h>
#include <ros.h>
#include <sensor_motor_gui/sensors.h>
#include <sensor_msgs/JointState.h>

// Declare your pins and variables here
#define pi 3.14159265359
int guiState = 1;
int guiButton;
int guiButton_last = 0;
int guiState_count = -1;

const int LED_pin = 24;

//=================== Sensors ===================//
#define window_size 5
const int gui_switch = 53;

// Potentiometer
int pot = A2;
int servoVal;

// IR Sensor
const int IR_SENSOR = A1;

// Flex Sensor
int flexiForcePin = A0;

// Ultrasonic Sensor
const int TrigPin = 7;
const int EchoPin = 13;
float distanceCm;
int duration;

// Button to switch btw ultrasonic or flex to control stepper
const int button = 22;
int buttonState;
int buttonState_last = 0;
int mode_count = -1;
int mode=-1;
unsigned long lastDebounceTime = 0;
unsigned long debounceDelay = 50;
```

```
int moving_avg_val = 0;

//PID controller variables for position controller
int set_Pos;
long input_Pos;
double kp_p=0.01, ki_p=0.001, kd_p=5;
int e_pos_sum = 0;
int e_pos_last = 0;

//PID controller variables for speed controller
double input_Speed, set_Speed;
double kp_s=0.35, ki_s=0.001, kd_s=0.15;
int e_speed_sum = 0;
int e_speed_last = 0;

// DC Motor Helpers
void toggleMotorDirection(int error)
{
  if (error > 0)
  {
    digitalWrite(L1, LOW);
    digitalWrite(L2, HIGH);
  }
  else if (error < 0)
  {
    digitalWrite(L1, HIGH);
    digitalWrite(L2, LOW);
  }
}

int calcPIDPos(long actual_pos, int des_pos)
{
  int e_pos = des_pos - actual_pos;
  int pwm_pulse = 0;
  toggleMotorDirection(e_pos);
  if (abs(e_pos) > 10)
  {
    pwm_pulse = abs(kp_p*e_pos + ki_p*e_pos_sum + kd_p*(e_pos - e_pos_last));
    e_pos_last = e_pos;
    e_pos_sum += e_pos;
    if (pwm_pulse > MAX_PWM_PULSE)
    {
      pwm_pulse = MAX_PWM_PULSE;
    }
    else if (pwm_pulse < MIN_PWM_PULSE_POS)
    {
```

```
    moving_avg_arr[k] = moving_avg_arr[k-1];
  }
  moving_avg_arr[0] = input;
  return moving_avg_val;
}


float readIR(){
  int ir_reading = analogRead(IR_SENSOR);
  int in = filterInput(ir_reading);
  float volt = in * (5.0 / 1023.0);
  float dist = 125.77 * exp(-0.768 * volt);
  return dist;
}

//======================= ROS =====================//
void callback(const sensor_msgs::JointState& msg) {

    if (guiState == 0) return;

//    set_Speed = msg.position[3] * 255 / 1.57;  // velocity
    // DC Motor (writing to motor happens in void loop)
    set_Speed = msg.position[3];
    set_Pos = msg.position[2] * 360 / 3.14;  // angle

    // buttonState == 1 means controlling motors with GUI
    // Servo
    servo.write(msg.position[0] * 180 / pi);

    // Stepper
    // Code that drives a stepper, given the radian to set, msg.position[1]
    int desiredStep = msg.position[1] * stepsPerRevolution / (2 * pi);
    stepper(desiredStep);

}

ros::NodeHandle nh;
ros::Subscriber<sensor_msgs::JointState> sub("joint_states", callback);
sensor_motor_gui::sensors sensors_msg;
ros::Publisher pub("sensors_readings", &sensors_msg);

//====================== Setup ====================//
void setup() {
    Serial.begin(57600);

```

```
    pinMode(LED_pin, OUTPUT);

    // Ultrasonic Sensor
    pinMode(TrigPin, OUTPUT);
    pinMode(EchoPin, INPUT);

    // IR
    pinMode(IR_SENSOR, INPUT);

    // Servo
    servo.attach(servoPin);

    // Stepper
    pinMode(stepperEnable,OUTPUT); // Enable
    pinMode(stepperStep,OUTPUT); // Step
    pinMode(stepperDir,OUTPUT); // Dir
    digitalWrite(stepperEnable,LOW); // Set Enable low
    pinMode(button, INPUT);

    // DC Motor
    pinMode(ENCODER_PIN_1, INPUT_PULLUP);
    pinMode(ENCODER_PIN_2, INPUT_PULLUP);
    pinMode(L1, OUTPUT);
    pinMode(L2, OUTPUT);
    pinMode(ENABLE_PWM, OUTPUT);
    pinMode(dc_BUTTON, INPUT);
    digitalWrite(dc_BUTTON, HIGH);
    old_button = digitalRead(dc_BUTTON);

    input_Pos = 0;
    set_Pos = 0;

    initMovingAverage();

    //Turn dc motor off
    analogWrite(ENABLE_PWM, 0);
    digitalWrite(L1, LOW);
    digitalWrite(L2, LOW);

    // ROS
    nh.initNode();
    nh.subscribe(sub);
    nh.advertise(pub);
}
```

```
void loop() {
    // Potentiometer
    float potValue;

    // Flex
    float force;

    // // IR
    float irValue;

    // Ultrasonic
    float distanceCm;

    guiButton = digitalRead(gui_switch);
    if ((guiButton ==  HIGH) && (guiButton_last ==  LOW)){
      guiState_count ++;
      guiState = guiState_count % 2;
    }

    // debouncing
    if (guiButton != guiButton_last){
      delay(50);
    }

    guiButton_last = guiButton;

    Serial.print("Button: ");
    Serial.println(mode);

    // Controlling motors using sensors
    if (guiState == 0) {
        // Servo Control Part
        potValue = analogRead(pot);
        servoVal = map(potValue, 0, 1023, 0, 180);
        servo.write(servoVal);
```

```
        // Stepper Control Part
        buttonState = digitalRead(button);
        if ((buttonState ==  HIGH) && (buttonState_last ==  LOW)){
          mode_count ++;
          mode = mode_count % 2;
          // mode = mode_count % 4;
          if (mode ==0){
            // nh.loginfo("Change to Ultrasonic Sensor");
//            Serial.print("ULTRASONIC");
          }
          if (mode ==1){
            // nh.loginfo("Change to Flexiforce Sensor");
//            Serial.print("FLEX");
          }
          sum = 0;
          index = 0;
          memset(sensorReadings, 0, sizeof(sensorReadings));

          flex_sum = 0;
          flex_index = 0;
          memset(flex_sensorReadings, 0, sizeof(flex_sensorReadings));
        }

        // debouncing
        if (buttonState != buttonState_last){
          delay(10);
        }
        // update the last state of Button 0
        buttonState_last = buttonState;

        force = Flexiforce();
        distanceCm = Ultrasonic();

        if (mode==0){
//          distanceCm = Ultrasonic();
          sensors_msg.ultra = distanceCm;
          int desiredStep = round(map(distanceCm, 0, 100, 0, stepsPerRevolution));
          stepper(desiredStep);
        //   nh.loginfo("CurrentStep: %f", currentStep);
        //   nh.loginfo("desiredStep: %f", desiredStep);
          Serial.print("Ultrasonic: ");
          Serial.println(distanceCm);
        }
        if (mode==1) {
//          force = Flexiforce();
//          sensors_msg.flex = force;
```

```
        int desiredStep = round(map(force, 0, 4.4, 0, 200));
        stepper(desiredStep);
      //   nh.loginfo("CurrentStep: %f", currentStep);
      //   nh.loginfo("desiredStep: %f", desiredStep);
        Serial.print("Flex Force: ");
        Serial.println(force);
      }

      delay(10);

      // DC Motor Control Part
      irValue = readIR();
      digitalWrite(L1, LOW);
      digitalWrite(L2, HIGH);
      int output_pwm = map(irValue, 15, 60, MIN_PWM_PULSE_SPEED, MAX_PWM_PULSE);
      constrain(output_pwm, MIN_PWM_PULSE_SPEED, MAX_PWM_PULSE);
      analogWrite(ENABLE_PWM, output_pwm);
  }
  else {
    new_button = digitalRead(dc_BUTTON);
    if (new_button != old_button)
    {
      if (new_button == HIGH)
      {
        stateToggle();
      }
      delay(40);
      old_button = new_button;
    }

    if (control_Pos)
    {
      long input_Pos = encoder.read();
      float output_pwm = calcPIDPos(input_Pos, set_Pos);
      analogWrite(ENABLE_PWM, output_pwm);
    }
    else
    {
      unsigned long newTime = millis();
      float timeElapsed = (newTime - oldTime) / 1000.0;
      long newPosition = encoder.read();
      float input_Speed = (newPosition - oldPosition) / timeElapsed;
      input_Speed = input_Speed * 60.0 / 360.0;
      toggleMotorDirection(set_Speed);
```

```
        int output_pwm = calcPIDSpeed(input_Speed, set_Speed, last_output);
        if (set_Speed == 0)
        {
          output_pwm = 0;
        }
        analogWrite(ENABLE_PWM, output_pwm);
        last_output = output_pwm;
        oldPosition = newPosition;
        oldTime = newTime;
      }
    }

    // Publish message
    sensors_msg.button = guiState;
    sensors_msg.flex = force;
    sensors_msg.ultra = distanceCm;
    sensors_msg.ir = irValue;
    sensors_msg.pot = potValue;
    pub.publish(&sensors_msg);

    nh.spinOnce();
}
```

Reference

[1] Rosserial Arduino Tutorial, http://wiki.ros.org/rosserial_arduino/Tutorials

[2] Coborg Arduino Project, https://github.com/CoborgCMU/Arduino-project, 2021