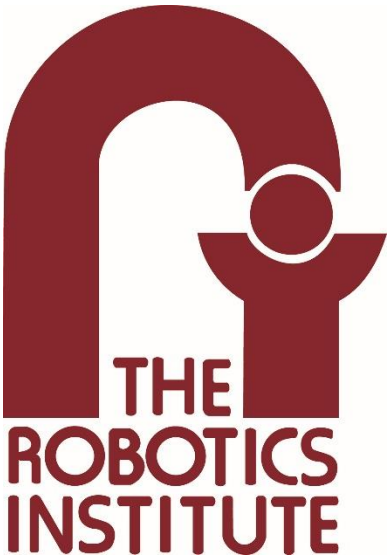

MRSD Project Course

Team I – AIce

Autonomous Zamboni Convoy

Individual Lab Report 01

Author

Nick Carcione

Teammates

Rathin Shah

Yilin Cai

Jiayi Qiu

Kelvin Shen

February 9, 2022

Contents

1. Individual Progress

1.1 Sensors and Motors Lab

My main responsibilities for the Sensors and Motors Lab were to wire the circuits and develop the code for an IR proximity sensor, DC motor, and switch for changing motor states.

I also aided in the integration of the total circuit, especially when it came to debugging and troubleshooting any issues that arose with the DC motor.

1.1.1 IR Sensor

The IR sensor used in the lab was a Sharp GP2Y0A21YK IR proximity sensor, which is an analog sensor that is nominally capable of measuring in a range of 10-80 cm. The datasheet for this sensor provides a plot detailing the transfer function from distance to voltage. This plot is reproduced in Figure 1 below.

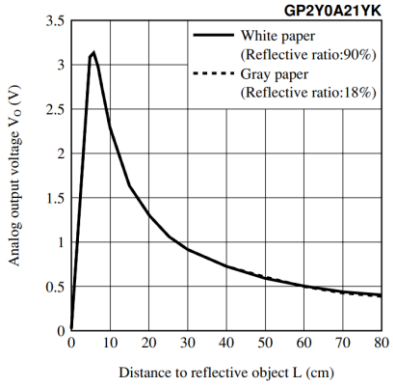

Figure 1: Provided transfer function plot for the Sharp IR proximity sensor[3]

As the plot above makes obvious, the signal output by the sensor is nonlinear with respect to distance. The plot itself is also the reverse of what I needed for reading the sensor, providing distance-to-volts instead of volts-to-distance. A source online[1] informed me of an Arduino library for using this sensor that would automatically convert voltage to distance for me. This site also provided the equation used to do so. However, when I tried implementing this equation, the results I obtained were wildly off. As a result, I had to take my own measurements and develop my own transfer function for converting the voltage signal to distance. While taking initial measurements, I noticed the signal output by the sensor was very noisy, so I implemented a moving average filter to smooth the signal out. After doing so, I continued to take measurements, recording 10 data points at every multiple of 5cm from the reported range of 10cm to 80cm. Doing so showed that the valid range of my specific sensor was actually from 15-80 cm, and that at 10 cm the sensor had yet to reach the peak shown in Figure 1. After collecting all my measurements, I tried fitting multiple curves to the data and selected the curve with the highest R^2 value. The data and transfer function I produced are shown in Figure 2 on the next page.

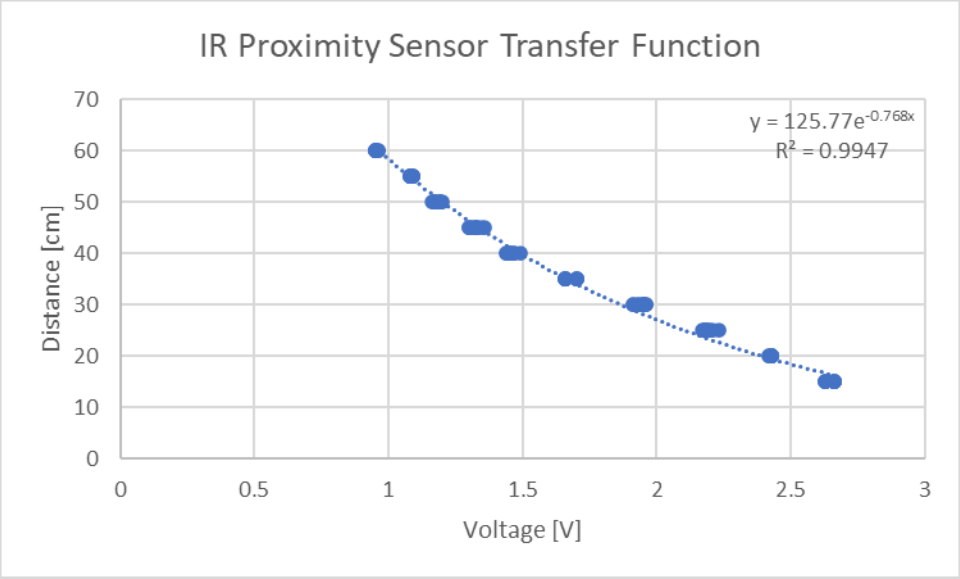

Figure 2: Transfer function converting voltage to distance for my actual sensor

As the plot shows, the range of the sensor was further reduced from 15-80 cm to 15-60 cm, as the sensor accuracy began to drop outside of this range.

1.1.2 DC Motor

The DC motor was a Cytron SPG30-30K motor. Since the motor was a 12V motor, controlling the motor with an Arduino necessitated the use of an L298 motor driver and an external 12V power supply. The two 12V power leads of the DC motor were connected to 2 of the out pins on the motor driver. The 12V power supply was wired into the positive and negative input power terminals on the driver. The Arduino was then connected to the driver by connecting two digital output pins to inputs 1 and 2 of the motor driver and connecting an analog output pin with pulse width modulation (PWM) capability to the enable pin of the driver. Doing so allowed me to control the speed and direction of the motor according to the following logic table presented in Figure 3. By sending a higher PWM signal, the motor would move faster.

ENABLE	L1	L2	Result
L	L	L	OFF
L	L	H	OFF
L	H	L	OFF
L	H	H	OFF
H	L	L	BRAKE
H	L	H	FORWARD
H	H	L	BACKWARD
H	H	H	BRAKE
PWM	L	L	PULSE-BRK
PWM	L	H	FWD-SPD
PWM	H	L	BCK-SPD
PWM	H	H	PULSE-BRK

Figure 3: Logic table for controlling speed and direction of DC motor[4]

The motor came with an encoder included in it. The encoder was given 5V power and ground from the Arduino and the encoder itself was wired to pins 2 and 3 of the Arduino. An

external library for handling encoders was used to read the number of tics sent by the encoder. The library recommended that the leads be connected to pins with an interrupt for the best result; on the Arduino Uno, these are pins 2 and 3. The encoder defined one tic as completion of 1-4 in Figure 4 below. However, the encoder library counted each one of these phases as a single tic. Multiplying the reported number of tics per revolution from the datasheet (90) by 4 resulted in 360 tics per revolution, or 1° per tic; this result was confirmed through visual inspection.

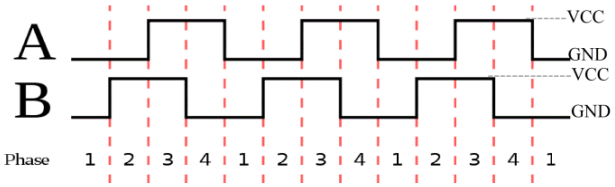

Figure 4: Encoder waveforms[2]

The DC motor was outfitted with PID control so that both the position and speed of the motor could be set to a user specified value. Using the tics to define the angular position of the motor, the error in position (or speed) was calculated and used in a PID controller to generate a PWM signal that would drive the motor to the desired position (or speed). The friction present in the motor restricted the range of PWM signals that could be sent to the motor. Anything below a PWM threshold of ~90-100 would not provide enough power to overcome the static friction forces. As a result, the motor could not reach speeds below 80 rpm. This also resulted in the motor being unable to perfectly reach the desired position since small, slow adjustments were impossible; instead, a positional "dead band" was defined that shut off PID control when the motor got close enough to the desired position.

Finally, the motor was interfaced with the IR proximity sensor. The distance returned by the sensor was mapped to a PWM signal such that as the distance measured decreased, the motor speed would slow down. This was meant to mimic a vehicle stopping as it neared an obstacle or pedestrian.

1.1.3 Button

The push button that I added to the circuit was used to switch the motor between position and velocity control when the motor is being controlled by the GUI. The button was debounced using a simple delay of 40 milliseconds that began when the state first transitioned from low to high.

1.1.4 Circuit and Code

A schematic of the complete sensor-motor-button circuit that I developed is shown in Figure 5a on the next page. The team's fully integrated circuit is shown alongside it in Figure 5b on the next page. The code that I wrote to read the IR sensor, control the DC motor, and handle the button functionality is reproduced in Appendix B. This code was developed without the GUI and therefore uses the Arduino serial monitor for user input and information output.

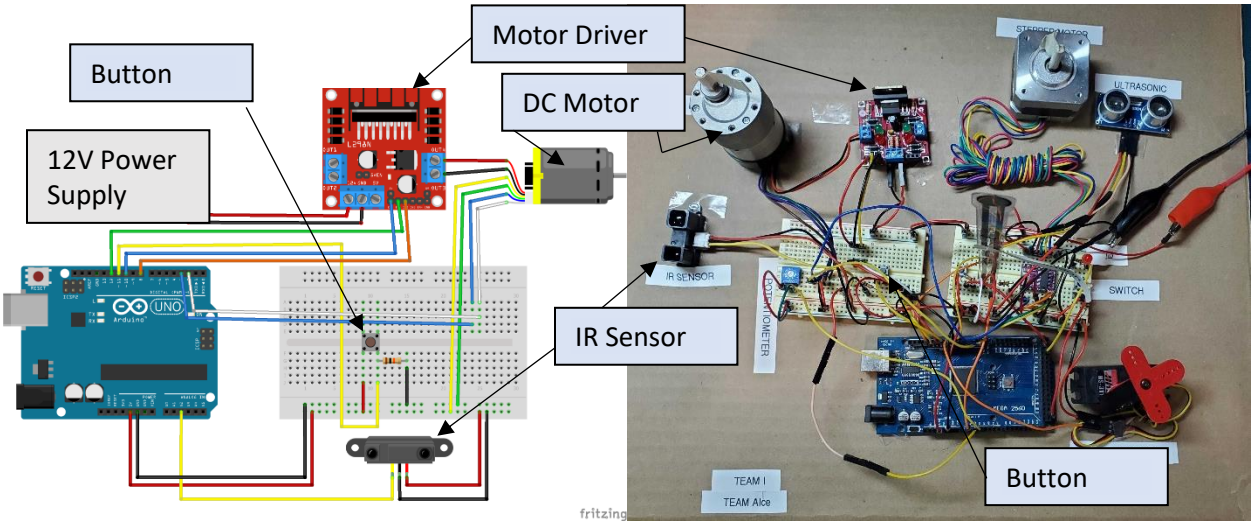

Figure 5: (a) on the left, a schematic of the sensor-motor-button circuit created; (b) on the right, a picture of the fully integrated circuit including (a) [photo courtesy of Rathin]

1.2 MRSD Project

My progress on the project so far has been split between helping to build out the simulation and preparing for future hardware integration. For the simulation, I have researched methods for obtaining accurate estimates of the follower Zamboni's linear and angular velocities. Since the Zamboni will be operating indoors in an arena, it was required that the method I chose did not rely on GPS data for calculating or correcting the estimate. After conducting some research, I identified the "robot_localization" package that has been developed for ROS. This package is capable of fusing data from any arbitrary number of sensors using a Kalman filter. The package supports a wide variety of sensors and data types including IMU data and wheel encoders.

On the hardware side, I have begun identifying potential components that will be useful in converting the Zamboni into a drive-by-wire platform. Currently, some functions of the Zamboni that we wish to control (i.e., steering and braking) are powered hydraulically instead of electrically. I have been able to find some potential solutions to these problems. One of these is an electro-hydraulic steering valve that can attach to the back of the steering wheel and take electrical input to change the pressure in the hydraulic steering lines. I have also begun to look into smaller platforms that the team can use for live demonstrations in the event that we either don't receive the Zamboni or receive the Zamboni too late to integrate our sensors and controllers with it.

2. Challenges

2.1 Sensors and Motors Lab

As described above, converting the output of the IR proximity sensor to an accurate distance proved to be an unexpected challenge due to the inaccuracy of the sensor's datasheet.

An even bigger challenge was combining the individual sensor-motor circuits into one final circuit that interfaced with the GUI. Since the individual circuits were wired separately, many of the GPIO pins on the Arduino Uno were used by multiple components; this required shifting pins around while keeping track of what moved where and what needed pulse-width-modulation capability. The GUI posed an even bigger challenge. ROS was used to interface with the Arduino, and simply initializing the necessary ROS nodes used up 70-80% of the Arduino Uno's memory. This caused the code to not upload to the board at times, increased the amount of lag in the system, and decreased performance of the sensors and motors. Replacing the Uno with an Arduino Mega provided plenty of memory space and solved these issues.

2.2 MRSD Project

The most significant challenge I have faced with respect to the project is a lack of access to and information about the Zamboni that we will be using. Without these, developing accurate models of the odometry sensors, and thus an accurate simulation, is difficult. The robot_localization package requires covariance matrices for these sensors to be tuned, and without real information about their performance, the accuracy of the simulation is not guaranteed. This also makes selecting hardware difficult since the exact sizes and dimensions of parts of the Zamboni are largely unknown. We hope to have a conversation with our sponsors and representatives from Zamboni to get a better understanding of the Zamboni and how we can adapt it to meet our purposes. Another challenge regarding the Zamboni is that it is currently not designed to be a drive-by-wire platform, so additional work must be done to find and integrate hardware that will make it possible to control it electronically.

3. Teamwork

3.1 Sensors and Motors Lab

The work for this lab was divided equally so that four of the five members developed code and circuits for a sensor-motor pair, and the fifth member developed the GUI for interfacing with the complete circuit. The individual contributions of the rest of the team are summarized below.

Rathin Shah

Rathin worked with the potentiometer and the RC servo motor. He also assisted in integrating, debugging, and testing the final circuit.

Yilin Cai

Yilin worked with the ultrasonic rangefinder and the stepper motor. He was also in charge of integrating the circuit components together and cleaning up the wiring

Jiayi Qiu

Jiayi worked with the FlexiForce sensor and the stepper motor. She also integrated the code for the FlexiForce sensor, ultrasonic rangefinder, and stepper motor and developed

debouncing code for a button that would switch the control of the stepper motor between the FlexiForce sensor and ultrasonic rangefinder.

Kelvin Shen

Kelvin developed the GUI that interfaced with all the different sensors in motors in the final circuit. He also took primary responsibility in integrating everyone's code together and testing the final circuit.

3.2 MRSD Project

Rathin Shah

Rathin has started work on developing the controls for the Zamboni in Simulink, including creating a pure pursuit controller that utilized Ackermann steering. Developing the pure pursuit controller in MATLAB and Simulink has also led him to research how to connect Simulink to a ROS node. Finally, Rathin has worked alongside of me on learning and implementing the robot_localization package for velocity estimation.

Yilin Cai

Yilin has overseen the creation and maintenance of the Zamboni convoy simulation in ROS and Gazebo. He has created a model of the Zamboni in Gazebo that is equipped with sensors (camera, lidar, IMU, encoders) and can be teleoperated by keyboard. He has further extended the Gazebo simulation to work with two Zambonis to replicate the intended use case.

Jiayi Qiu

Jiayi has assisted in developing the Zamboni simulation in Gazebo. She has also created an environment in Gazebo which mimics the size, shape, and appearance of a hockey rink.

Kelvin Shen

Kelvin has worked on developing the perception capabilities of the follower Zamboni. This has included learning how to use cv_bridge to communicate between a ROS camera topic and OpenCV. He has also generated and tested a board of ArUco markers in Gazebo so that they can later be used to determine the pose and position of the leader Zamboni.

4. Plans

4.1 Sensors and Motors Lab

The control of the DC motor using the IR proximity sensor serves as a precursor to part of the safety system that will be included on the autonomous follower Zamboni. In a similar fashion to how the DC motor slowed down as it detected objects getting closer, the Zamboni will slow down and, if needed, stop if it detects an obstacle or a person within a certain range. While the exact components will not be the same (e.g., lidar and a vision system instead of an IR proximity sensor), the general concept will be carried over.

4.2 MRSD Project

In the coming weeks, the team as a whole plans on wrapping up development on a basic simulation of the autonomous Zamboni convoy. To help reach this goal, I plan on implementing the robot_localization package in ROS with the goal of obtaining accurate velocities from the IMU and wheel encoders on the follower Zamboni. Further to this end, I will also begin researching methods for estimating the velocity of the leader Zamboni based on the perception sensors that will be on board the follower (RGBD camera, lidar). In preparation for eventual integration with a physical Zamboni, I will continue to identify and specify the hardware necessary to add drive-by-wire functionality to the provided Zamboni.

5. Sources

[1] B. De Bakker, "How to use a SHARP GP2Y0A21YK0F IR Distance Sensor with Arduino," Makerguides, 2021. [Online]. Available: How to use a SHARP GP2Y0A21YK0F IR Distance Sensor with Arduino. [Accessed: 28-Jan-2022].

[2] Cytron Technologies, "DC Geared Motor with Encoder," MO-SPG-30E-XXXK User's Manual V1.1, May 2011.

[3] Sharp, "GP2Y0A21YK/GP2Y0D21YK General Purpose Type Distance Measuring Sensors".

[4] Solarbotics, "The Compact L298 Motor Driver," Revised 5 Jan. 2022.

Appendix A:

Sensors and Motor Control Quiz

1. Reading a datasheet. Refer to the ADXL335 accelerometer datasheet (https://www.sparkfun.com/datasheets/Components/SMD/adxl335.pdf) to answer the below questions.
   - What is the sensor's range?

     $\pm$3.6g (using typical value)
   - What is the sensor's dynamic range?

     7.2g (based on typical value)
   - What is the purpose of the capacitor C_{DC} on the LHS of the functional block diagram on p. 1? How does it achieve this?

     The capacitor C_{DC} is meant to decouple the accelerometer from noise on the power supply. The capacitor achieves this both by absorbing any excess voltage supplied by the power supply and by discharging to power the accelerometer when the supplied voltage drops.
   - Write an equation for the sensor's transfer function.

     $$V = \left(300 \frac{mV}{g}\right) * a + 1.5\,V$$

     $$where\ a\ is\ the\ measured\ acceleration\ in\ [g]$$

     $$and\ V\ is\ the\ output\ voltage\ in\ [V]$$
   - What is the largest expected nonlinearity error in g?

     $\pm$0.3% of F.S.O. = $\pm$0.3% * 7.2g = $\pm$0.0216g
   - What is the sensor's bandwidth for the X- and Y-axes?

     0 Hz to 1600 Hz
   - How much noise do you expect in the X- and Y-axis sensor signals when your measurement bandwidth is 25 Hz?

     $$rms\ noise = Noise\ Density * \sqrt{BW * 1.6} = 150\ \frac{\mu g}{\sqrt{Hz}} * \sqrt{25\ Hz * 1.6} = 949\ \mu g$$

     $$= 0.000949\ g$$
   - If you didn't have the datasheet, how would you determine the RMS noise experimentally? State any assumptions and list the steps you would take.

     To measure the RMS noise experimentally, the accelerometer would be placed on a stationary, level surface with an assumed 0g of acceleration in all three directions. The acceleration measurements would then be recorded at a rate of 25 Hz for some arbitrary amount of time (e.g., 10 seconds). The resulting data points would be individually squared before being summed together. This sum would then be divided by the number of samples and square rooted to yield the RMS noise. This calculation would be done separately for each direction.

2. Signal conditioning
   - Filtering
     - Name at least two problems you might have in using a moving average filter.

       1. A moving filter might introduce lag into a sensor, especially if the averaging window is large.

       2. A moving filter is susceptible to outliers. A single value that is much larger or smaller than the other values in the window can cause the reported average to be dragged up or down as long as the outlier is in the averaging window.

- Name at least two problems you might have in using a median filter.

  1. If the window size of the median filter is not large enough or the data that are being filtered have a high frequency of outliers, the median filter may fail to filter out the outliers as desired.

  2. Using a median filter requires storing previous inputs and generating multiple sorted windows for finding the medians. As a result, median filtering takes up extra space in memory and adds to the run time of the program.

- Opamps
  - In the following questions, you want to calibrate a linear sensor using the circuit in Fig. 1 so that its output range is 0 to 5V. Identify in each case: 1) which of V1 and V2 will be the input voltage and which the reference voltage; 2) the values of the ratio Rf/Ri and the reference voltage. If the calibration can't be done with this circuit, explain why.
    - Your uncalibrated sensor has a range of -1.5 to 1.0V (-1.5V should give a 0V output and 1.0V should give a 5V output).

      Input voltage: V_2
      Reference voltage: V_1
      Rf/Ri = 1
      Reference voltage = -3V
    - Your uncalibrated sensor has a range of -2.5 to 2.5V (-2.5V should give a 0V output and 2.5V should give a 5V output).

      The desired calibration cannot be done with this circuit. Since the dynamic range of the input is already the same as that of the output (-2.5V →2.5V = 5V and 0V →5V = 5V), using V_2 as the input voltage necessitates a resistance ratio of 0. This, however, makes it impossible to solve for the reference voltage or have the needed offset. Furthermore, using V_1 as the input voltage is also impossible, as doing so inverts the input voltage. This causes an input of -2.5V to be mapped towards 5V and an input of 2.5V to be mapped toward 0V, opposite of what is desired. The only way to fix this is with a negative resistance ratio, which is not physically possible.

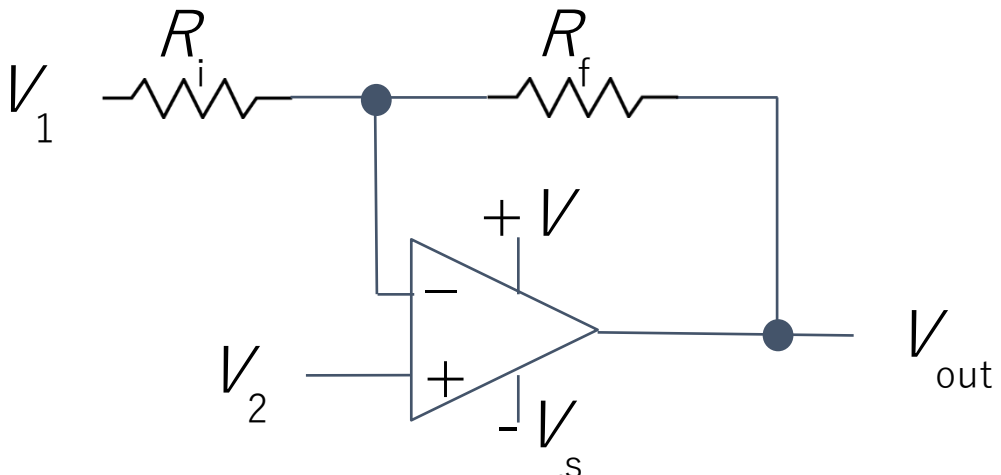

Fig. 1 Opamp gain and offset circuit

3. Control
   - 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 (Proportional, Integral, Derivative) terms.
     Proportional (P): The input for the proportional term would be the error or difference between the desired position of the DC motor and the current position of the motor as reported by a motor encoder.
     Integral (I): The input for the proportional term would be the sum of all the errors between the desired and actual motor positions multiplied by the timestep.
     Derivative: The input for the derivative term would be the difference between the current and previous errors in motor position divided by the timestep.

   - If the system you want to control is sluggish, which PID term(s) will you use and why?
     The proportional (P) term would be increased. Doing so causes the control input to be greater for a given error, making the system more aggressive and decreasing the rise time.

   - After applying the control in the previous question, if the system still has significant steady-state error, which PID term(s) will you use and why?
     The integral (I) term would be used. The integral term of the controller takes into account the integral (sum) of the error. When the system reaches steady-state ad still has a significant error, the integral term will continually grow larger and eventually push the system to the desired steady-state value.

   - After applying the control in the previous question, if the system still has overshoot, which PID term(s) will you apply and why?
     The derivative (D) term would be applied. The derivative term acts as a virtual damper on the system and reduces overshoot in the controlled system. Mathematically, this is because the derivative term incudes the derivative of the error. As the system approaches the desired value and the error decreases, the derivative term applies a control signal opposite that of the proportional term, thereby damping the system.

Appendix B:

Arduino Code for IR Sensor & DC Motor

```
#include <Encoder.h>

//#########################IMPORTANT#########################
//Some general notes about motor speeds/directions
//IN_1 = LOW, IN_2 = HIGH makes encoder tics increase (go +) (CCW)
//IN_1 = HIGH, IN_2 = LOW makes encoder tics decrease (go -) (CW)
//Setting ENABLE_PWM to 0 or LOW causes motor to stop
//-----------------------------------------------------------
//When using program with serial monitor, must set to "No Line Ending" option
//Otherwise, an extra "1" is sent for each command and DC motor does not respond correctly
//#############################################################

//Declare input and output pins
const int IR_SENSOR = A1;            //For reading the IR sensor
const int ENCODER_PIN_1 = 2;         //One of the two DC motor encoder pins
const int ENCODER_PIN_2 = 3;         //The other DC motor encoder pin
const int BUTTON = 11;               //Button for switching between position and velocity control
const int IN_1 = 12;                 //Input 1 of the H bridge
const int IN_2 = 8;                  //Input 2 of the H bridge
const int ENABLE_PWM = 10;           //Enable pin of the H bridge

const int MAX_PWM_PULSE = 255;       //The maximum value PWM can be
const int MIN_PWM_PULSE_POS = 107;   //The minimum PWM value that can be used to move the motor in position control
                                     //Otherwise, the motor cannot overcome friction and does not move
const int MIN_PWM_PULSE_SPEED = 90;  //The minimum PWM value that can be used to move the motor in velocity control
                                     //Otherwise, the motor may not overcome friction and not move
const int MAX_POS_WINDUP = 2000;     //Set limits on the error sum (I term) of the position PID controller to avoid windup
const int MAX_SPEED_WINDUP = 2000;   //Set limits on the error sum (I term) of the velocity PID controller to avoid windup
const int POSITION_DEADBAND = 10;    //How many +/- degrees the motor will get within of the desired position
```

```
int sensor_cont = 1;                //Used for switching between user controlled and sensor control
                                    //Swapped by hand for testing purposes (replaced by dedicated GUI on/off button in final circuit)
bool control_Pos = true;            //Used for deciding if motor in position or velocity control; default to position
bool old_button;                    //Used for switch debouncing; holds previous button state
bool new_button;                    //Used for switch debouncing; holds current button state

//Create an Encoder object for built in encoder handling
Encoder encoder(ENCODER_PIN_1, ENCODER_PIN_2);

//Variables for moving average filter
const int MOV_AVG_WINDOW_SIZE = 5;
int moving_avg_arr[MOV_AVG_WINDOW_SIZE];
int moving_avg_val = 0;

//PID controller variables for position controller
int set_pos;
long input_pos;
double kp_p=0.001, 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;

//Change the motor direction depending on the position error
//In the case of velocity control, change direction depending on sign of desired velocity
void toggleMotorDirection(int error)
```

```
{
  if (error > 0)
  {
    digitalWrite(IN_1, LOW);
    digitalWrite(IN_2, HIGH);
  }
  else if (error < 0)
  {
    digitalWrite(IN_1, HIGH);
    digitalWrite(IN_2, LOW);
  }
}

//PID controller for position control
//Calculates error between desired and current position and converts that
//into a suitable and feasible PWM pulse
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) > POSITION_DEADBAND)
  {
    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)
    {
```

```
    pwm_pulse = MIN_PWM_PULSE_POS;
   }
  }

  if (e_pos_sum > MAX_POS_WINDUP)
  {
   e_pos_sum = MAX_POS_WINDUP;
  }
  else if (e_pos_sum < -MAX_POS_WINDUP)
  {
   e_pos_sum = -MAX_POS_WINDUP;
  }

  return pwm_pulse;
}

//PID controller for velocity control
//Calculates error between desired and current velocity and converts that
//into a suitable and feasible PWM pulse
int calcPIDSpeed(double actual_speed, double des_speed, int prev_pulse)
{
  int e_speed = des_speed - actual_speed;
  int pwm_pulse = prev_pulse;
    if (des_speed > 0)
    {
      pwm_pulse = prev_pulse + kp_s*e_speed + ki_s*e_speed_sum + kd_s*(e_speed - e_speed_last);
    }
    else if (des_speed < 0)
    {
      pwm_pulse = prev_pulse - (kp_s*e_speed + ki_s*e_speed_sum + kd_s*(e_speed - e_speed_last));
    }
    e_speed_last = e_speed;
```

```
    e_speed_sum += e_speed;
    if (pwm_pulse > MAX_PWM_PULSE)
    {
      pwm_pulse = MAX_PWM_PULSE;
    }
    else if (pwm_pulse < MIN_PWM_PULSE_SPEED)
    {
      pwm_pulse = MIN_PWM_PULSE_SPEED;
    }

  if (e_speed_sum > MAX_SPEED_WINDUP)
  {
    e_speed_sum = MAX_SPEED_WINDUP;
  }
  else if (e_speed_sum < -MAX_SPEED_WINDUP)
  {
    e_speed_sum = -MAX_SPEED_WINDUP;
  }
  return pwm_pulse;
}

//Switches between position and velocity control when button is pressed
//Stops the motor when state is switched to avoid side effects
void stateToggle()
{
  if (control_Pos)
  {
    analogWrite(ENABLE_PWM, 0);
    control_Pos = false;
  }
  else
  {
```

```
    analogWrite(ENABLE_PWM, 0);
    control_Pos = true;
    encoder.write(0);            //Resets the current 0 position to where the motor currently is
  }
}

//Calculates the average the moving average array
int getAverage(int arr[])
{
  int avg = 0;
  for (int j=0; j<MOV_AVG_WINDOW_SIZE; ++j)
  {
    avg += arr[j];
  }
  avg /= MOV_AVG_WINDOW_SIZE;
  return avg;
}

//Populates the moving average array with sensor readings when the program first runs
void initMovingAverage()
{
  for (int i=0; i<MOV_AVG_WINDOW_SIZE; ++i)
  {
    moving_avg_arr[i] = analogRead(IR_SENSOR);
  }
  moving_avg_val = getAverage(moving_avg_arr);
}

//Applies a moving average filter to the IR sensor readings and returns the filtered value
//The moving average array is updated for each input received from the sensor
int filterInput(int input)
{
```

```
  moving_avg_val = getAverage(moving_avg_arr) + (1/MOV_AVG_WINDOW_SIZE) * (input -
moving_avg_arr[0]);
  for (int k=MOV_AVG_WINDOW_SIZE-1; k>0; --k)
  {
    moving_avg_arr[k] = moving_avg_arr[k-1];
  }
  moving_avg_arr[0] = input;
  return moving_avg_val;
}

//Reads in the analog value from the IR sensor
//Filters the IR sensor reading
//Converts the filtered sensor reading to volts
//Finally, converts from volts to distance (in cm) and returns the distance
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;
}

void setup() {
  //Begin the Serial monitor (serial monitor used for testing/without GUI)
  Serial.begin(9600);
  //Declare the pin modes as appropriate
  pinMode(IR_SENSOR, INPUT);
  pinMode(ENCODER_PIN_1, INPUT_PULLUP);   //Enable internal pullup resistor on encoder pin for better
readings
  pinMode(ENCODER_PIN_2, INPUT_PULLUP);   //Enable internal pullup resistor on encoder pin for better
readings
  pinMode(IN_1, OUTPUT);
  pinMode(IN_2, OUTPUT);
```

```
  pinMode(ENABLE_PWM, OUTPUT);
  pinMode(BUTTON, INPUT);
  digitalWrite(BUTTON, HIGH);
  old_button = digitalRead(BUTTON);

  //Initialize starting values of controlled variables to 0
  input_pos = 0;
  set_pos = 0;
  input_speed = 0;
  set_speed = 0;

  //Initialize the moving average array
  initMovingAverage();

  //Turn motor off
  analogWrite(ENABLE_PWM, 0);
  digitalWrite(IN_1, LOW);
  digitalWrite(IN_2, LOW);
}

//Set up global variables for calculating speed of motor
long oldPosition = 0;
long newPosition = 0;
unsigned long oldTime = millis();
int last_output = 0;

void loop() {
  //Read IR sensor and print measured distance to serial monitor for validation
  Serial.print("Distance:  ");
  float ir_value = readIR();
  Serial.print(ir_value);
  Serial.println(" cm");
```

```
//Check to see if the button has been pressed using simple delay debouncing
//Switch between position and velocity control if it has been pressed
new_button = digitalRead(BUTTON);
if (new_button != old_button)
{
  if (new_button == HIGH)
  {
    stateToggle();
  }
  delay(40);
  old_button = new_button;
}

if (sensor_cont == 1)
{
  //Use the IR sensor to control the speed of the motor
  //The closer the object gets, the slower the motor spins
  //Meant to mimic vehicle stopping to avoid hitting pedestrian
  //If no obstacle is detected (outside the 60 cm range of sensor), go full speed
  digitalWrite(IN_1, LOW);
  digitalWrite(IN_2, HIGH);
  int output_pwm = map(ir_value, 15, 60, MIN_PWM_PULSE_SPEED, MAX_PWM_PULSE);
  constrain(output_pwm, MIN_PWM_PULSE_SPEED, MAX_PWM_PULSE);
  Serial.println(output_pwm);
  analogWrite(ENABLE_PWM, output_pwm);
}
else
{
  //Use the serial monitor to input desired positions and speeds
  if (control_Pos)
  {
```

```
    //implement PID control on the position of the motor


    //Read in desired position from serial monitor if user has specified one
    if (Serial.available() > 0)
    {
      set_pos = Serial.parseFloat();
    }


    //Read the current position of the motor and use this to find a PID calculated PWM signal
    //to move the motor as appropriate
    //Prints out the current position for validation
    long input_pos = encoder.read();
    int output_pwm = calcPIDPos(input_pos, set_pos);
    analogWrite(ENABLE_PWM, output_pwm);
    Serial.print("Position:  ");
    Serial.println(input_pos);
  }
  else
  {
    //Implement PID control on the speed of the motor.  Motor can rotate at speeds between 80 - 130 rpm in either direction


    //Read in desired speed from serial monitor if user has specified one
    if (Serial.available() > 0)
    {
      set_speed = Serial.parseFloat();
    }


    //Calculate the current speed of the motor using the difference in tics
    //and time between the current and previous iteration of loop(0
    //Change the motor direction depending on desired speed direction (+ or )
    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);

    //Use the current and desired speed and the previous PWM pulse to find
    //the next PWM pulse.  Send that pulse to the motor.
    //Stop the motor if desired speed is 0 rpm
    int output_pwm = calcPIDSpeed(input_speed, set_speed, last_output);
    if (set_speed == 0)
    {
      output_pwm = 0;
    }
    analogWrite(ENABLE_PWM, output_pwm);
    //Update previous times, positions, and PWM pulse values
    oldTime = newTime;
    oldPosition = newPosition;
    last_output = output_pwm;
    //Print speed for validation
    Serial.print("Speed:  ");
    Serial.println(input_speed);
    delay(100);                 //Delay helps performance of speed control (gives more time to elapse,
                                //more tics to count)
   }
}
}
```