EE-335 Lab Project Book

EE-335: Advanced Microcontrollers

Allen Douglas

OIT – Spring 2020

Authors

Zachary Whitlock

Introduction

The objective of this document is to outline the design and learning process of creating a multi-function robot. The term project for the class is to construct a robot using per-specified components by a series of smaller labs attaching, studying, and using the individual components. Ultimately, the robot will be capable of autonomous operate as well as remote control. At the end of the term, the robot is tested in an obstacle course and has to overcome challenges designed to test the design and implementation.

Labs consist of software and hardware development, focusing on using a microcontroller as the heart of the project. This term, the more powerful “Arduino Mega” microcontroller is used instead of a device like the Arduino Uno or Nano.

Table of Contents

Lab 1 – Bluetooth Communication......6
Objective......6
Part 1 – Connecting (with wires)......6
Part 2 – Connecting wirelessly......7
Part 3 – Custom Software......8
Conclusion......9
Appendix......9
Recieve/Send Program......9
Deciphering Program......10
Lab 2......12
Objective......12
Part 1 – Using the Encoders......12
The Encoder Class......13
Upon Rollover......13
Upon Zero RPM......13
Timer Prescaler......13
The Circuit......14
Serial Monitor Output......14
Part 2 – Using the Motors......15
Code and Function Explanation......15
Physical Setup......15
Conclusion......16
Appendix......16
Lab2-Part1.ino......16
encoder.cpp......18
encoder.h......21
motors.cpp......24
motors.h......26
Lab 3......27
Introduction......27
Part 1 – Gripper & Servo......27
The Gripper Class......27
Functions/Methods in the Class......27
The Circuit......28
Figures and Results......28
Part 2 – Line Sensors......29
Overview......29
The LineSensor Class......29
Functions/Methods......29
Figures and Results......30
Part 3 – Ultrasonic Rangefinder......31
Overview......31
The RangeFinder Class......31
Figures and Results......33
Conclusion......34
Appendix......35

Lab3.ino....35
gripper.h....36
gripper.cpp....37
rangeFinder.h....37
rangeFinder.cpp....38
lineSensor.h....39
lineSensor.cpp....40
Lab 4....41
Introduction....41
Part 1 – PI(D) Control....41
Overview....41
Code Explanation....41
Tuning....42
Figures and Results....42
Part 2 – Line Following....44
Overview....44
Code Explanation....44
Figures and Results....46
Tuning....47
Conclusion....47
Appendix....48
Lab4.ino....48
lineFollow.h....50
lineFollow.cpp....51
PIDSpeedControl.h....54
PIDSpeedControl.cpp....57
Lab 5....61
Objective....61
Part 1 – Testing PID Speed Control....61
Overview....61
Approach....61
Outcome....61
Part 2 – Testing Line Following....63
Overview....63
Approach....63
Outcome....63
Part 3 – Testing Remote Control & Gripper....64
Overview....64
Approach....64
Outcome....65
Part 4 – Testing Maze Navigation....66
Overview....66
Approach....66
Outcome....66
BOM....67
Prints / Parts....68
Conclusion....69
Appendix....69
Lab5.ino....69

PIDSpeedControl.cpp....76
PIDSpeedControl.h....79
lineFollow.cpp....82
lineFollow.h....85
encoder.cpp....87
encoder.h....89
motors.cpp....91
motors.h....92
rangeFinder.cpp....93
rangeFinder.h....94
gripper.cpp....95
gripper.h....96
lineSensor.cpp....96
lineSensor.h....97

Lab 1 – Bluetooth Communication

Objective

The objective of this lab is to wirelessly communicate with an Arduino Mega via Bluetooth. The Arduino Mega (an Atmega2560) is hooked up to a SparkFun RN-42 “BlueSMiRF” Bluetooth module. An app on my cell phone is used to send text to the PC via the Bluetooth module and through the Mega.

Part 1 – Connecting (with wires)

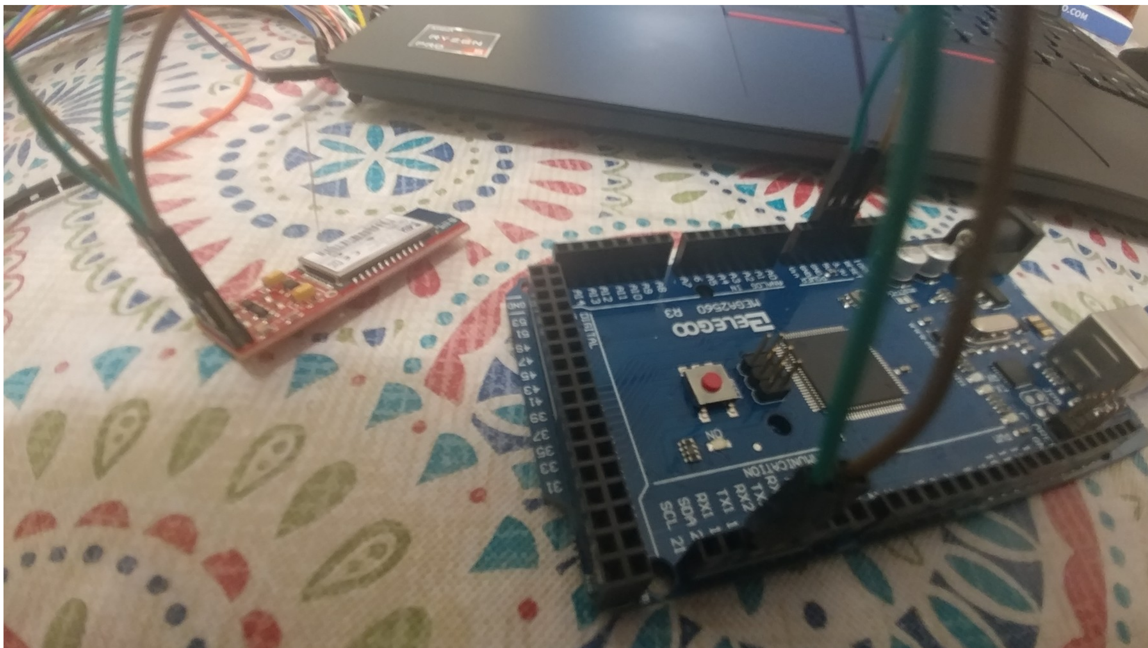

Figure 1: Wiring setup (And my wife's tablecloth...)

I opted to use one of the Mega’s actual serial ports instead of a software serial connection. I used “Serial1”, connecting the BlueSMiRF’s TX pin to RX1 and it’s RX pin to TX1 on the Mega. The Bluetooth module is also powered by the Mega’s GND and 5V pins, making a total of 4 wires. Later on, the oscilloscope was also attached to the MEGA’s ground and it’s probe pin was held on the SMiRF’s transmit pin by hand. Figure 1 shows the setup.

Part 2 – Connecting wirelessly

In this part of the lab, the software is set up and data is sent between the devices. The "Bluetooth_Testing_HC05.ino" example code was downloaded from the class webpage and was modified slightly. Namely, the baud rate for the Bluetooth module was found to be 115200 and I chose to use the hardware serial ports.

The app "Bluetooth Serial Controller" from "NEXT PROTOTYPES" was installed on my android phone. I also went ahead and paired the Bluetooth module while I was add it, after powering on the Arduino Mega.

Finally, I configured the app to send text when I hit a button. Namely, as per class instructions, the buttons would send "b1?", "b2?", and so on until "b9?". Figure 2 is a decoded capture from my oscilloscope while recording the TX line of the Bluetooth module.

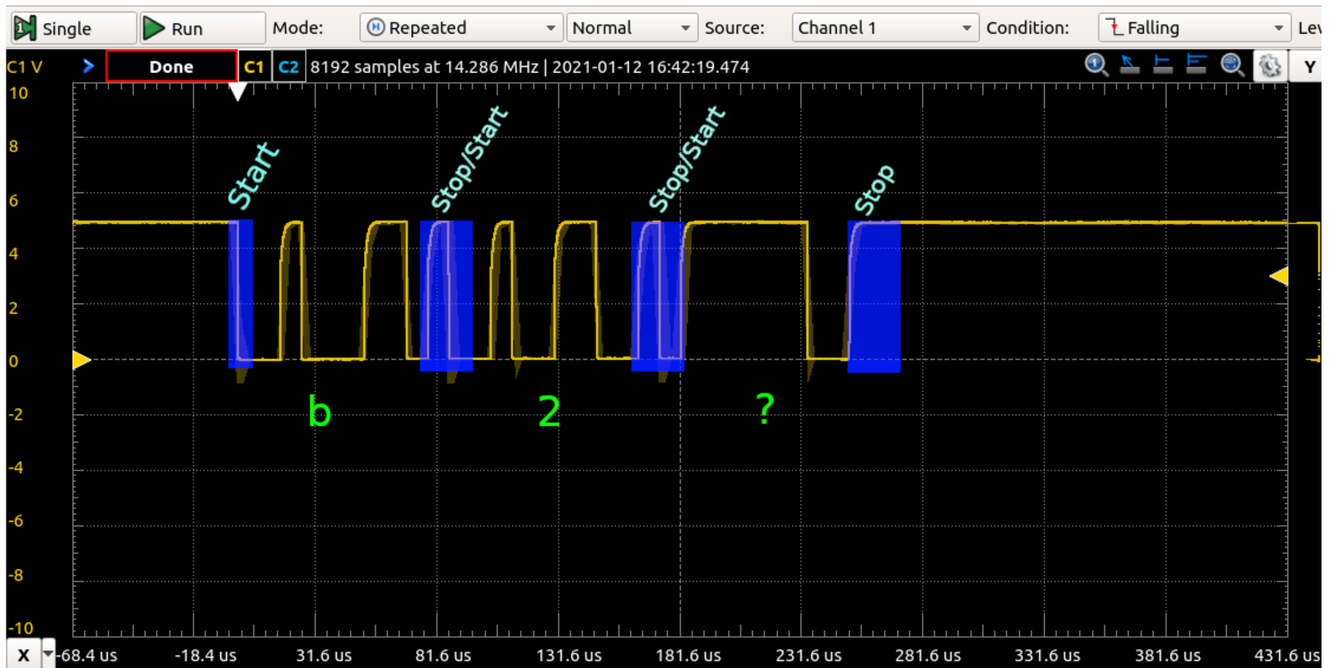

Figure 2: Decoded serial data

Figure 3 is an example screenshot of the Arduino Serial Monitor; containing text forwarded from the Bluetooth module.

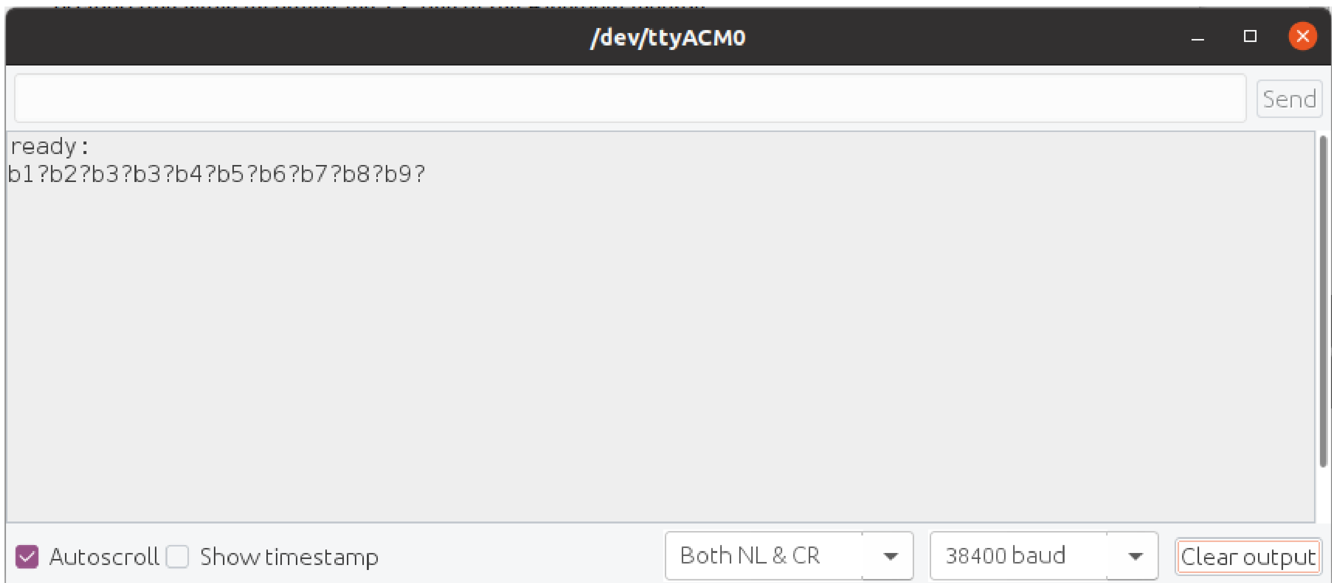

Figure 3: Serial monitor

Part 3 – Custom Software

In this part of the lab, the goal is to modify the previous program so that it will tell you specifically which buttons are being pressed. “Button 1 pressed”, “Button 2 pressed”, etc. The code continuously checks for new data on the Serial1 port, and when it receives some it reads the data into a buffer. The buffer is then examined to see if the received data starts with ‘b’, and if so, the code executes a switch

statement on the next character. The second character is supposed to be a number from 1 to 9, but if the result is invalid the code will print out the bad string and an error. The code is in the appendix, and Figure 4 shows this code in action, having printed results to the serial monitor.

Figure 4: Capture of serial monitor

Conclusion

Appendix

Recieve/Send Program

```
#include <SoftwareSerial.h>

void setup() {
  Serial.begin(38400); // Computer serial
  Serial.println("ready: ");
  // Usually 9600 for BT mode, although it is sometimes 38400:
  Serial1.begin(115200); // Bluetooth module serial
}

void loop() {
  if(Serial1.available())
  {
    Serial.print((char)Serial1.read());
  }
```

```
  if(Serial.available())
  {
    Serial1.print((char)Serial.read());
  }
}
```

Deciphering Program

```
/* Filename: Bluetooth-Modified.ino
 * Author: Zachary Whitlock
 * Class: EE335
 */

#include <SoftwareSerial.h>

#define BT_CHARS 3 // How many characters to look for

void setup() {
  Serial.begin(38400); // Computer serial
  Serial.println("ready: ");
  // Usually 9600 for BT mode, although it is sometimes 38400:
  Serial1.begin(115200); // Bluetooth module serial
}

void loop() {a
  if(Serial1.available())
  {
    // Read serial until we get the ending "?" character
    char fromBT[BT_CHARS + 1];
    Serial1.readBytesUntil('?', fromBT, BT_CHARS);
    // Figure out if we got a button press
    if (fromBT[0] == 'b') {
      // Run our switch statement for which button
      switch (fromBT[1]) {
        case '1':
          Serial.println("Button 1 pressed!");
          break;
         case '2':
          Serial.println("Button 2 pressed!");
          break;
         case '3':
          Serial.println("Button 3 pressed!");
          break;
         case '4':
```

```
        Serial.println("Button 4 pressed!");
        break;
       case '5':
        Serial.println("Button 5 pressed!");
        break;
       case '6':
        Serial.println("Button 6 pressed!");
        break;
       case '7':
        Serial.println("Button 7 pressed!");
        break;
       case '8':
        Serial.println("Button 8 pressed!");
        break;
       case '9':
        Serial.println("Button 9 pressed!");
        break;
       default:
        Serial.println(fromBT);
        Serial.println("Unrecognized button.");
     }
   }
 }

}
```

Lab 2

Objective

The objective of this lab is to interface the Arduino Mega with the motor drivers and motor encoders. Upon the completion of this lab, the Arduin Mega will be able to control each motor individually by way of class objects in C++, and read the status and speed report of the encoders.

Part 1 – Using the Encoders

The encoders are very simple optical encoders from Amazon. They have two prongs that fit around the outside of the encoder disks. Each disk has 20 slots in it. The sensor reports when light gets through one of the slots in the encoder disks, which allows you to determine the period of time between slots passing in front of the encoder's sensor.

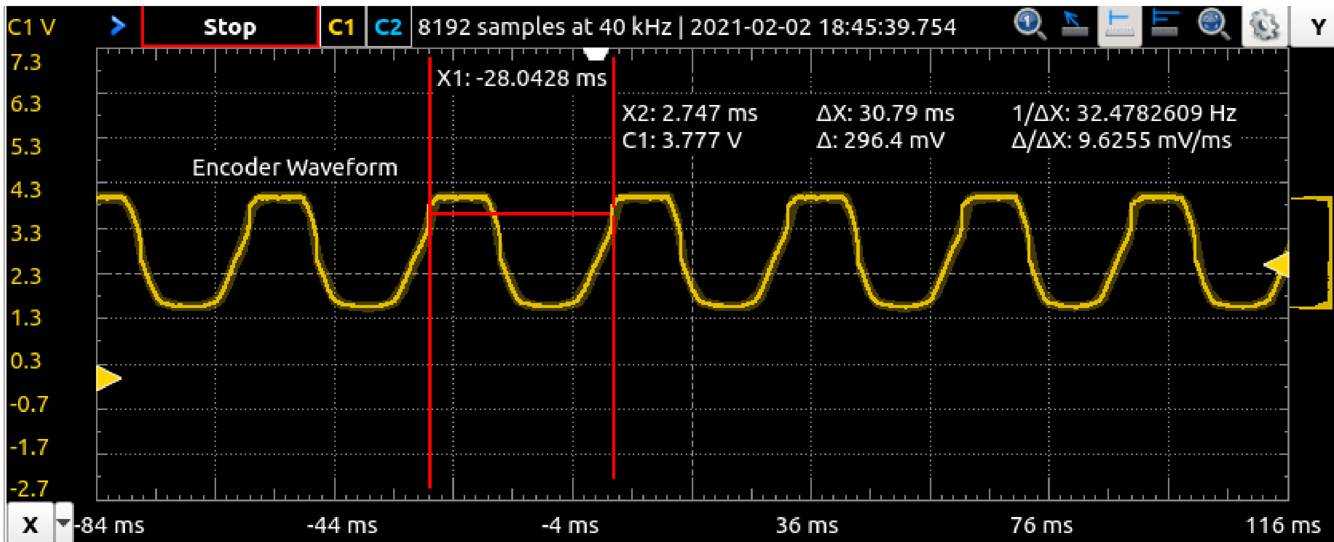

Figure 5: Encoder waveform capture

F

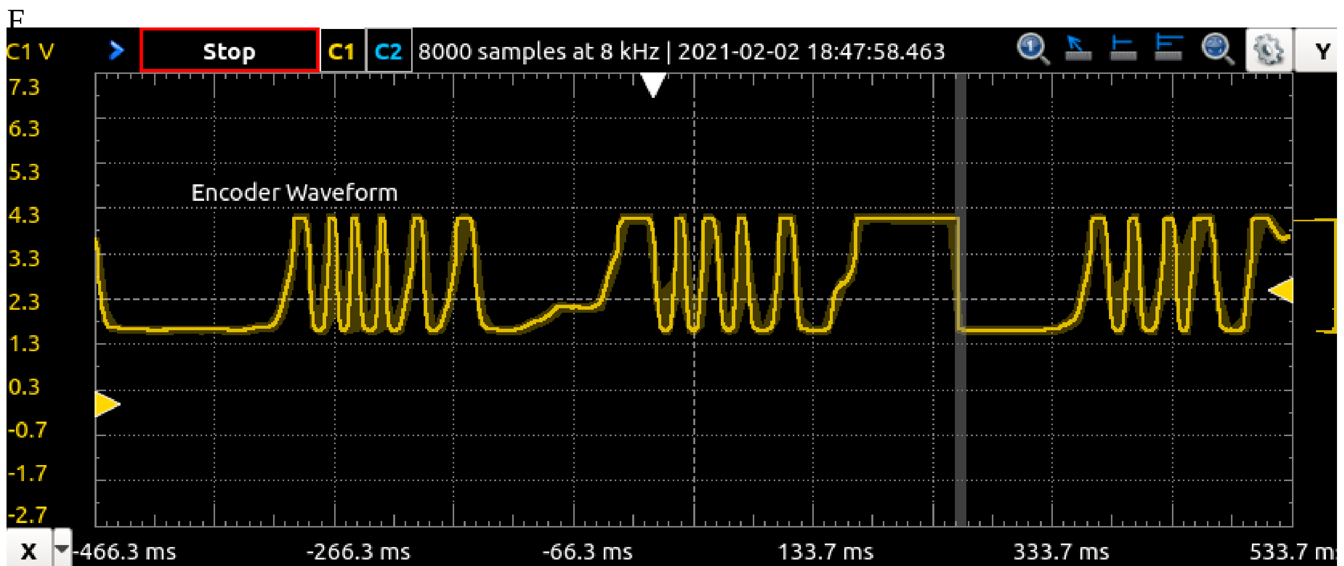

Figure 6: Encoder waveform capture while turning the wheel by hand

Figures 5 and 6 show the oscilloscope captures of the output of the encoder. It’s evident when the motor is turning versus when it is still by the turning ON and OFF of the signal.

The Encoder Class

A sort of “wrapper” around the encoders was written in the form of a C++ class. By using a class, you can monitor more than one encoder and it’s signal. Using a second, third, or Nth encoder is almost as simple as declaring a new instance of the encoder class. As of writing this lab, the following public functions are available in my EncoderClass object:

pinState()	Returns the current state of the encoder pin, primarily for debugging and/or future features.
encRPM()	Returns the current RPM of the motor, based on the averaged period
up(uint32_t n)	Tells the class we got a new rising edge from the arduino Timer.
encPeriodAvg()	Returns the averaged current period of time between high-edges on the encoder.
EncoderClass(int p);	Class constructor, and is given a pin to declare as an INPUT pin, and which it will read when the ‘pinState()’ function is called.

In additional to creating an instance of the EncoderClass with a digital pin, a timer must be tied to the “up()” method of each class so that it can keep track of it’s speed and when it fires.

Upon Rollover

The timer count is placed into a 32 bit variable, essentially nullifying the need for special rollover conditions in the rest of the code, since it would take days for the 32 bit variable to overflow back to 0. This is done by means of making the lower 16 bits of the 32 bit variable equal to the timer counter, and upon an overflow, a single 0x10000 (1 << 17) is added to the 32 bit count to represent the carry bit.

Upon Zero RPM

This is calculated upon reading the speed. The functions responsible will check if there are new edges since last time they checked the speed. If a certain number of checks are performed with no new rising edges being reported by the ‘up()’ method, then the speed is reported to simply be 0.

Timer Prescaler

The maximum prescale value available without using a secondary clock source is 16MHz/1024. This results in a sampling period of about 0.128ms, and a 16bit overflow roughly every 8.4 seconds. The measured minimum pulse width was not less than 9ms, which means that our sampling frequency is more than adequate.

The Circuit

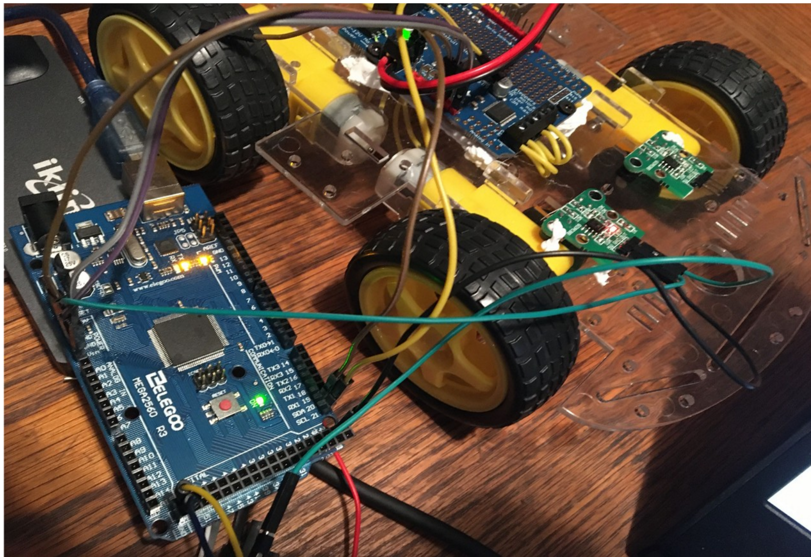

Figure 8: Arduino & Wiring

Figure 7: Encoder and disk

Figures 7 and 8 show the physical setup and wiring.

Serial Monitor Output

```
/dev/ttyACM0
                                                                Send
Current encoder1 RPM: 50
Current encoder1 period: 57ms
Current encoder1 RPM: 52
Current encoder1 period: 57ms
Current encoder1 RPM: 52
Current encoder1 period: 59ms
Current encoder1 RPM: 50
Current encoder1 period: 59ms
Current encoder1 RPM: 50
Current encoder1 period: 57ms
Current encoder1 RPM: 52
Current encoder1 period: 57ms
Current encoder1 RPM: 52
Current encoder1 period: 58ms
Current encoder1 RPM: 51
Current encoder1 period: 57ms
Autoscroll  Show timestamp      Both NL & CR    115200 baud    Clear output
```

Figure 9: Serial monitor output for encoder 1

Figure 9 shows the final output of the program.

Part 2 – Using the Motors

In this part of the lab, I created a C++ class to represent a DC motor. This code was somewhat simpler than the EncoderClass because there were very few if any measurements and calculations. Having a class to represent a DC motor lets us have a single instance of our ChassisMotor class for each of our four motors and control them independently.

Code and Function Explanation

The motor class is actually just an interface to Adafruit's DCMotor class and MotorShield class. The difference is that I've used static variables to instantiate and setup the MotorShield class when the first of my ChassisMotor classes are instantiated.

void setPwm(int spd);	Sets the "speed" of that motor, or more precisely, the PWM value.
void forward();	Tells the motor to turn on and go forwards
void reverse();	Tells the motor to turn on and go backwards
void roll();	Tells the motor to turn off
ChassisMotor(int motor_N);	Class constructor, is given the motor number (1-4 currently)

Full functionality of all motors was tested, varying speeds, stopping, and going forward/backward.

Physical Setup

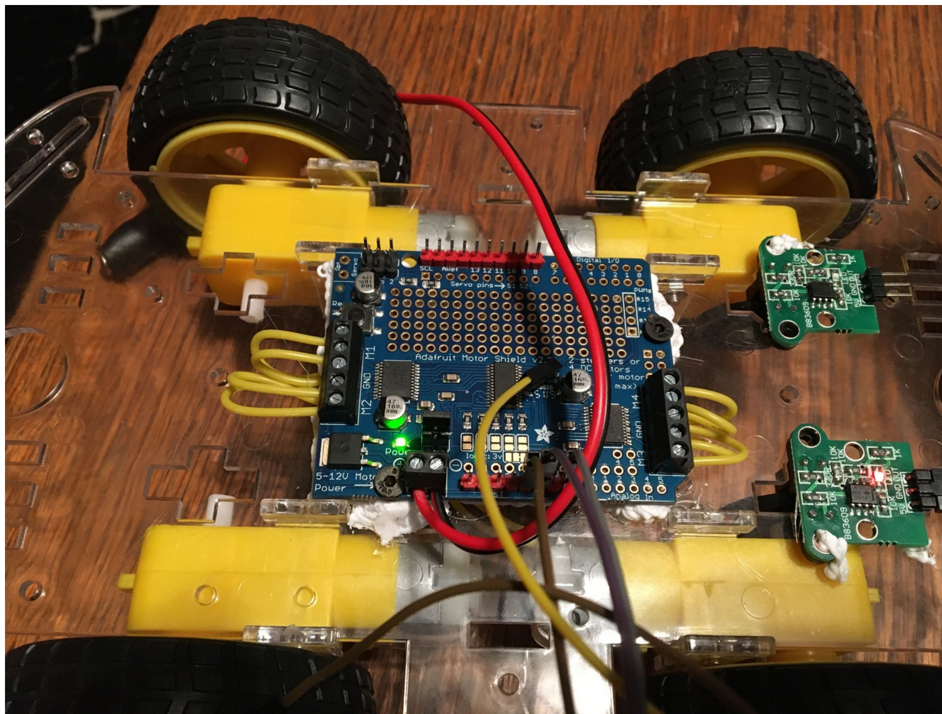

Figure 10: The motorshield wiring and chassis

It's worth noting the SDA/SCL pins were accidentally disconnected before this picture was taken.

Conclusion

In this lab I learned a lot about object-orientated programming C++. I ordinarily write “c-style”, but this lab was well designed for using classes to make streamlined and logical code that will be easy to work on over the course of the term. I also learned about the Timer counters in the Arduino Mega and how you can use an input to capture a count using one. Additionally, I learned how to use many of the features of the Adafruit MotorShield and how to wire it up. I also learned how to use the optical encoders, which should be relevant in other motion-control use cases and possibly user-input.

I did encounter a few challenges along the way. I need to order a new encoder after accidentally connecting power and ground backward to it, or at least get a new comparator chip. I almost damaged my Arduino Mega by leaving the “Vin Jumper” active on the MotorShield, which seems to tie the 5V rail to it’s Vin rail. This led to me temporarily pouring something like half an amp into the Mega’s voltage regulator with my benchtop power supply before I realized what was going on. In terms of the code writing, figuring out how to set up the timers and classes was definitely a challenge too. Sometimes equations wouldn’t work because of mixed data types, the “static” keyword was difficult to figure out how to use in a class, and getting accurate encoder readings took a while.

In short, this was a very education lab and I can now use the MotorShield, these geared DC Motors, optical encoders, and the Arduino Mega in future parts of this lab as well as future projects.

Appendix

Lab2-Part1.ino

```
#include "encoder.h"

#include "motors.h"


EncoderClass encoder1 = EncoderClass(48);

EncoderClass encoder2 = EncoderClass(49);


uint32_t tcount4 = 0;

uint32_t tcount5 = 0;


void setup() {

  // Startup the serial port

  Serial.begin(115200);

  delay(100);
```

```
  // This is only declared here because I want to print
  // stuff in the class constructor.
  ChassisMotor motor3 = ChassisMotor(3);

  // Turn on motor 3 for encoder readings
  motor3.forward();
  motor3.setPwm(200);

  // Turn on timers
  TCCR4A = 0b00000000; // Normal mode.
  TCCR4B = 0b11000101; // Noise cancel, High edge triggering,
  // Normal mode, divide by 1024
  TCCR5A = 0b00000000;
  TCCR5B = 0b11000101;

  // Enable interrupts
  TIMSK4 = 0b00100001; // (Input capture, overflow)
  TIMSK5 = 0b00100001; // (Input capture, overflow)

}

void loop() {
  // Periodically check our encoder output
  delay(100);

  Serial.print("Current encoder1 period: ");
  Serial.print(encoder1.encPeriodAvg());
  Serial.println("ms");
  Serial.print("Current encoder1 RPM: ");
  Serial.println(encoder1.encRPM());
```

```
}

ISR(TIMER4_CAPT_vect) {
  tcount4 = (tcount4 & 0xFFFF0000) | TCNT4;
  encoder1.up(tcount4);
}

ISR(TIMER5_CAPT_vect) {
  tcount5 = (tcount5 & 0xFFFF0000) | TCNT5;;
  encoder2.up(tcount5);
}

ISR(TIMER4_OVF_vect) {
  // Running this vector auto-clears the overflow
  // Add to the count
  tcount4 = tcount4 + 0x10000;
}

ISR(TIMER5_OVF_vect) {
  // Running this vector auto-clears the overflow
  // Add to the count
  tcount5 = tcount5 + 0x10000;
}
```

encoder.cpp

```
#include "encoder.h"

// Upon measuring the minimum pulse width, it seems to be ~10ms.
// The slowest timer speed is about 7.8kHz, or about 0.128ms.
```

```
// The timer rolls over once about every 8.38 seconds.

// I've decided to keep track of each timer in a 32 bit unsigned
// integer. This gives us 549755.8 seconds before overflow, in
// other words, 152.7 hours or 6.3 days :)
// (so essentially I'm just ignoring rollover...)

// What to do about zero speed or very little speed:
// We report that the speed is zero if the "checker" checks
// the speed a certain number of times and finds no new edges.
// (Check "registerEdge" N number of times, report 0)

// log the rising edge
void EncoderClass::up(uint32_t n) {
  timeElapsed = (n - lastCount)*0.128;
  addSample(timeElapsed);

  lastCount = n;

  registerEdge = true;

}

// Reports period in milliseconds, averaged
uint16_t EncoderClass::encPeriodAvg() {
  // Sum the array and return the average
  uint32_t sum;
  for (int n = 0; n < AVG_SIZE; n++) {
    sum = sum + periodAveraged[n];
  }
```

```
  sum = sum/AVG_SIZE;

  // Check if we even have a new edge
  if (registerEdge) {
    registerEdge = false;
    deadCount=0;
  } else { // No new samples!
    deadCount++;
    if (deadCount >= ALLOWED_DEAD_COUNT) {
      sum = 0;
    }
  }



  return sum;
}

uint16_t EncoderClass::encRPM(){
  // RPM, 1 rotation = 360 degrees, 1 "up" edge means
  // we went 1 slot, which is 18 degrees, 360/18 = 20

  // Calculate the RPM
  float temp = (50/float(encPeriodAvg()))*60;
  return temp;
}



// Adds a sample to the average
void EncoderClass::addSample(uint16_t millisSample) {
```

```
  // Shift previous samples to the right
  uint16_t copyArray[AVG_SIZE];  // make copy
  memcpy(copyArray, periodAveraged, AVG_SIZE * sizeof(uint16_t));
  for (int i = 0; i < AVG_SIZE-1; i++) {
    periodAveraged[i+1] = copyArray[i];
  }

  periodAveraged[0] = millisSample;
}



bool EncoderClass::pinState() {
  state = digitalRead(pin);
  return state;
}

// Class constructor
EncoderClass::EncoderClass(int p) {
  pin = p;
  deadCount = 0;

  // Make our pin an input
  pinMode(pin, INPUT);
}
```

encoder.h

```
/* File: encoder.h
 * Author: Zachary Whitlock
 * Description:
```

```
 * Header file for the encoder class object
 */

#include "Arduino.h"

#ifndef ENCODER_H
#define ENCODER_H

#define ALLOWED_DEAD_COUNT 10
#define AVG_SIZE 3

// Alright, so we need to know the time since last

// Encoder class
class EncoderClass {
  public:
    int pin;
    uint32_t count;
    bool pinState();

    // Returns the current RPM based on the averaged period
    uint16_t encRPM();

    // Fires upon a rising-edge
    void up(uint32_t n);

    // Reports the current period of the encoder
    uint16_t encPeriodAvg();

    // Class constructor
```

```
    EncoderClass(int p);

  private:
    // Adds a sample to the average
    void addSample(uint16_t millisSample);

    // States
    bool lastState;
    bool state;

    // Keeps track of how many times we checked without
    // finding new data.
    uint8_t deadCount;

    // Counts and times
    uint32_t lastCount;
    uint32_t countElapsed;
    bool registerEdge;

    // Average of 10 samples
    // TODO: make weighted
    uint16_t periodAveraged[AVG_SIZE];

    // Time between changes
    uint16_t timeElapsed;
};

#endif
```

motors.cpp

```
#include "motors.h"


// Zur Steuerung von Motoren
// (Managing is captilized?)
// Translates more roughly into "controlling"


// Okay so if "Zur" is 'for', and is a conjugate form of "zu" I believe.
// Controlling encoders would be like Zur Steuerung von Encoder, "For control of encoders"
// Von is 'of', which, as a preposition, still isn't conjugated?
// Prepositions are a class of words used to express special or temporal relations, like how and when.
// Germans have 4 different cases that can modify prepositions, nominative, accusative, dative, and gentive.
// Nominative tells us who or what is doing something, in this case I don't really know since it's basically
// just a title, "Zur",
// In accusative, it's very similar to the nominative in the sense that you're talking about something doing something,
// but in an accusative case, you're refering to the thing that is being affected by the action.
// So if your noun is being affected by the verb, you use the accusative form.


// Okay but in short, the cases (nominative, accusative, dative, and gentive, are important)


// Die ("dee") Motoren Code soll (should/is supposed to) anhangen (attach/pin to) die Adafruit Motoren Bibliothek (library)


// These are "static", only declared once :)
bool ChassisMotor::instanciated = false;
Adafruit_MotorShield ChassisMotor::adaShield = Adafruit_MotorShield();
```

```
void ChassisMotor::roll() {
  adaMotor->run(RELEASE);
}

void ChassisMotor::forward() {
  adaMotor->run(FORWARD);
}

void ChassisMotor::reverse() {
  adaMotor->run(BACKWARD);
}



void ChassisMotor::setPwm(int spd) {
  adaMotor->setSpeed(spd);
}

// Constructor for our thing
ChassisMotor::ChassisMotor(int motor_N) {
  // Instantiate adaShield if not instantiated

  // Setup new motor
  adaMotor = adaShield.getMotor(motor_N);

  if (instanciated == false) {
    Serial.println("Instanciating shield");
    adaShield.begin();

    instanciated = true;
  }
```

```
}
```

motors.h

```
#include "Arduino.h"
#include <Adafruit_MotorShield.h>

#ifndef CHASSISMOTOR_H
#define CHASSISMOTOR_H
class ChassisMotor
{
  public:
    ChassisMotor(int motor_N);
    void setPwm(int spd);
    void forward();
    void reverse();
    void roll();

  private:
    static bool instanciated;
    Adafruit_DCMotor *adaMotor;
    static Adafruit_MotorShield adaShield; // Instantiate ONCE

};


#endif
```

Lab 3

Introduction

The objective of this lab is to interface the Arduino Mega (ATMega2560) with a set of peripherals for the robot. The sensors are a pair of line sensors, and a ultrasonic rangefinder. A gripper servo has to be interfaced to the Arduino as well. By the end of this lab, I should have a C++ class for controlling each device individually.

Part 1 – Gripper & Servo

The Gripper Class

The gripper class functions as a wrapper around the gripper mechanism and it's driving servo. Several things make it convenient to have a wrapper: The servo must be limited in rotation, and it is convenient to have single functions for opening and closing the servo. From the lab manual:

1. "Remove the two strews from the gripper arms.
2. Command the servo to move between -90 and +90 degrees.
3. Note the position of the servo gear.
4. Adjust the gripper commands so that the servo only drives through the range of angles required to fully open and close the gripper arms."

Functions/Methods in the Class

Function Name	**Description**
GripperServo(int pin)	The class constructor. It is called with the PWM pin used to command the angle the servo strives to reach.
Open()	Opens the gripper, limits to 145 degrees on the servo for my gripper.
Close()	Closes the gripper; and limits the servo to 72 degrees for my gripper.
HalfOpen()	Sets the gripper to a half-open state.
SetAngle()	Allows you to (unsafely) set the angle of the servo in the gripper.

The only variable in the class is private and is an instance of the Arduino "Servo" class, used as an underlying control interface to the servo itself.

The Circuit

Even though the servo is technically wired to the motor shield; the signal and the power both come from the Arduino itself. This means you have a substantial power draw on your 5V regulator and you have to dedicate a digital PWM pin for each servo. In this lab, I chose to use digital pin 9.

Figures and Results

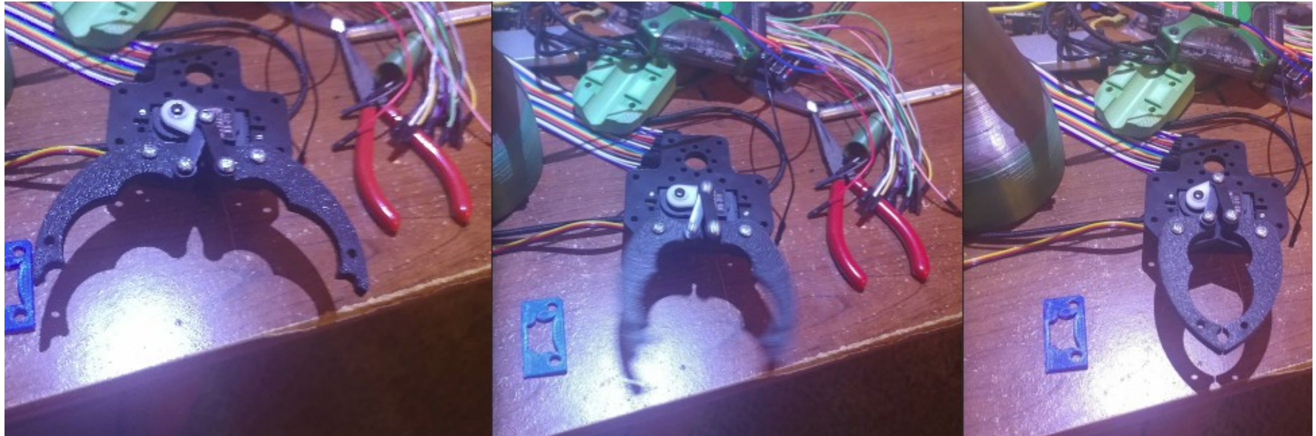

Figure 11: Gripper open, closing, and closed.

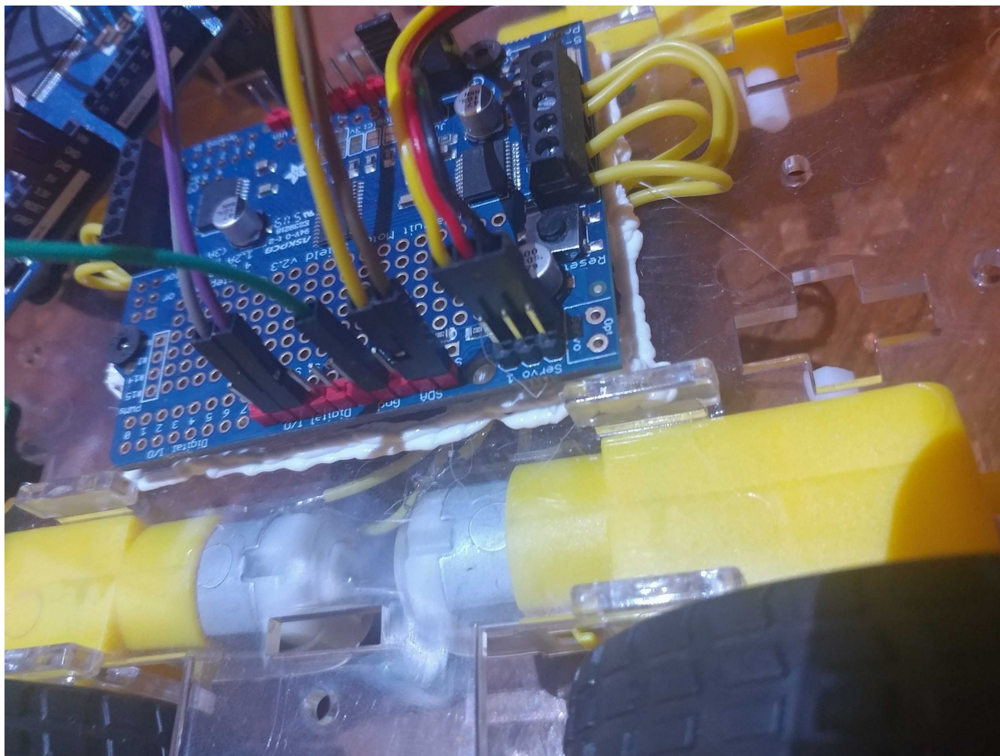

Figure 12: Servo connected to the motor shield

Figures 11 and 12 show the physical operation and wiring of the gripper. In later parts of this lab, the gripper is used as feedback for sensors. Serial monitor output from later parts also shows evidence of the gripper operation.

Part 2 – Line Sensors

Overview

The line sensors are extremely simple in operation. Light is supplied by an IR LED and only allowed to be received by another LED-like device when some surface reflects light back towards the sensor. When the sensor detects a suitable amount of light bouncing back from some surface, it outputs a LOW voltage as long as there is still reflected light. We use this property to detect when a dark (black) surface is in front of the sensor, versus when a light (white) surface is in front of the sensor. We use two sensors in order to be able to follow a black line on a flat surface that the robot will be driving on. The theory is, both sensors should see black at the same time when the robot is over the black line. When only a single sensor is showing black, we know that we are going away from the line in the direction of that sensor which is showing white. We can then make course corrections

The LineSensor Class

The line sensor class provides and easy way to instantiate and monitor one or more light sensors. All you have to do is instantiate the class with a pin to monitor and then call “OnBlack()” to check if the sensor is over a black surface.

Functions/Methods

Function Name	**Description**
LineSensor(int pin)	The class constructor. Called with a pin number to attach to and monitor for HIGH values.
OnBlack()	Returns ‘true’ if the sensor is NOT outputting a LOW.

The only variable used by the class is private and is used to store the pin the class is listening on.

Figures and Results

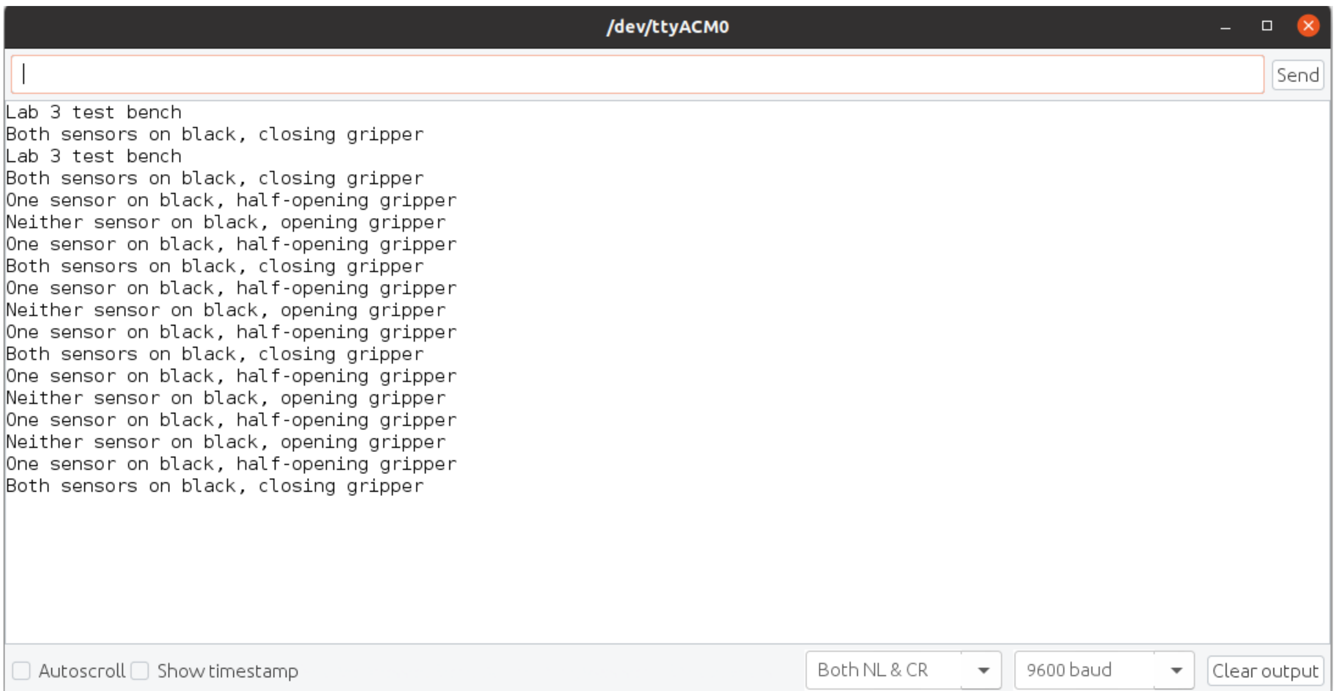

Figure 13: Serial monitor output of the test program running

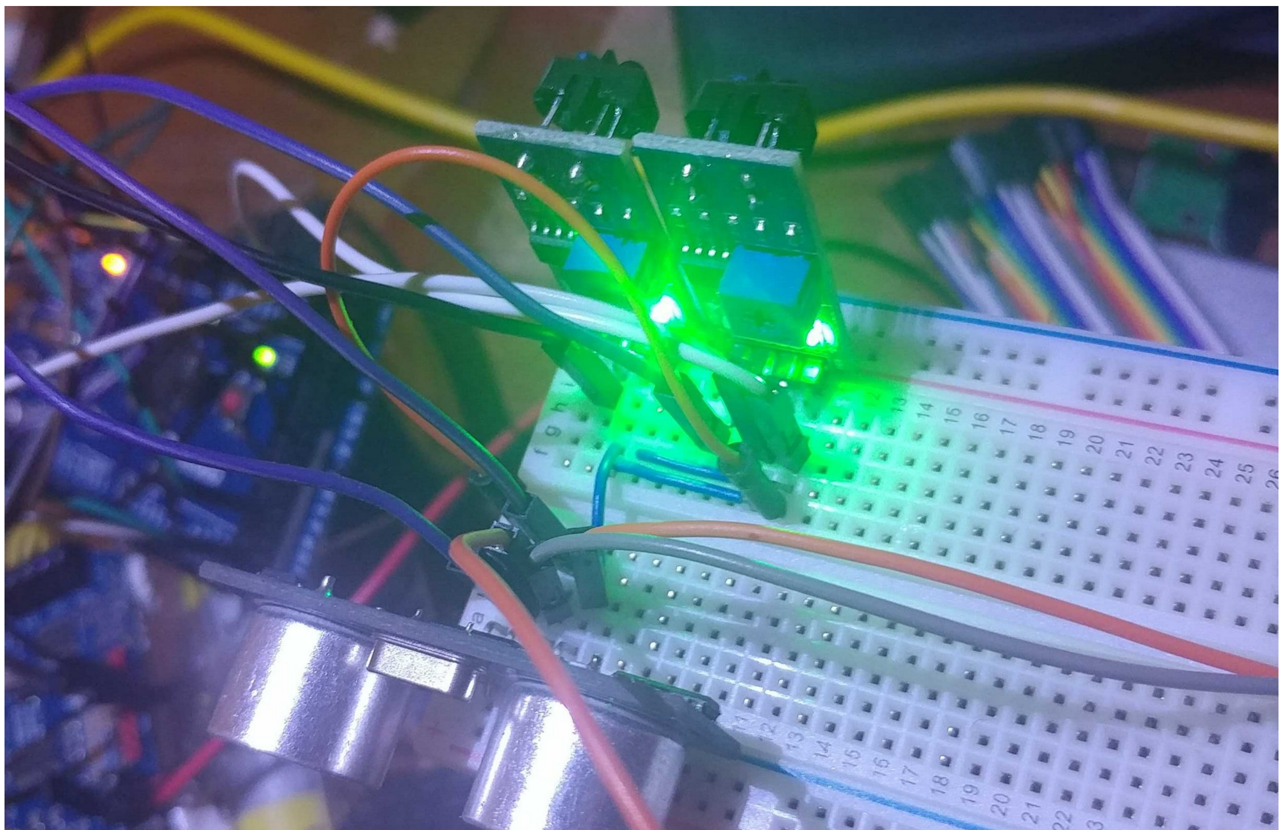

Figure 14: Wiring of the two line sensors

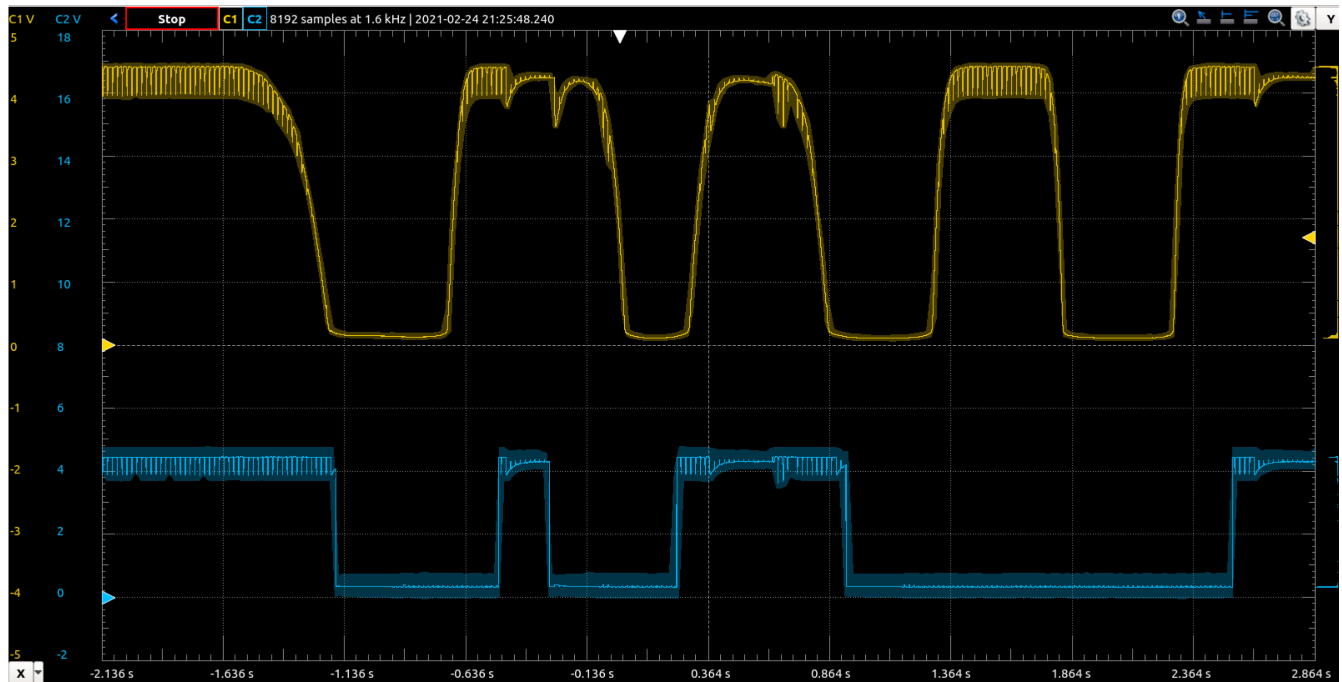

Figure 15: Oscilloscope capture of the sensor outputs while moving a light object in front of them.

Part 3 – Ultrasonic Rangefinder

Overview

The ultrasonic rangefinder works by sending a pulse of high frequency audio and then listening for echos off of surfaces in front of the sensor. Audio pulses are **triggered** by the controlling device, the Arduino. When the sensor senses and echo, it outputs a HIGH pulse on it's "Echo" pin for the duration of time from trigger to echo received. By measuring this pulse, it's possible to calculate how far away the reflecting object is. This sensor works similarly to the line sensor, but with enough precision to measure distance instead of simple binary measures of reflection.

The RangeFinder Class

This C++ is the most complex of this lab. Many problems were encountered while getting this working. The class serves as a way to easily measure distance without having to deal with pins and time measurements directly.

The input capture pins on the Arduino Mega are already used for the wheel encoders. This just leaves us with polling and pin interrupts to monitor the output of the Rangefinder. I used pin interrupts to hopefully allow program optimization should it become necessary.

Function Name	Description
RangeFinder(int trigPin, int echoPin)	The class constructor. It is called with the pin used to trigger the ultrasonic sensor and the pin used to monitor it. It sets the pin modes but not the pin interrupt.
Measure()	Triggers the ultrasonic sensor and returns the resulting pulse time, with a timeout of 2 seconds (should something go wrong). Returns 0 in the event of a timeout.
MeasureInch()	Calls Measure(), then calculates and returns the distance in inches as a floating point variable.
MeasureCm()	Calls Measure(), then calculates and returns the distance in centimeters as a floating point variable. Seems to be more accurate.
EchoEvent()	This method must be called by the pin interrupt ISR. As of writing this lab, I am unable to setup interrupts inside of a class, so the user has to point the ISR at this class method.

The private variables used by the class are:

- int echoPin: The digital pin used for measuring pulse width of an echo signal.
- int trigPin: The digital pin used to trigger the Rangefinder.
- int state: Used internally to determine when the pulse starts and ends.
- unsigned long usDelay: Used internally to store and then return the pulse width of the echo signal.

Figures and Results

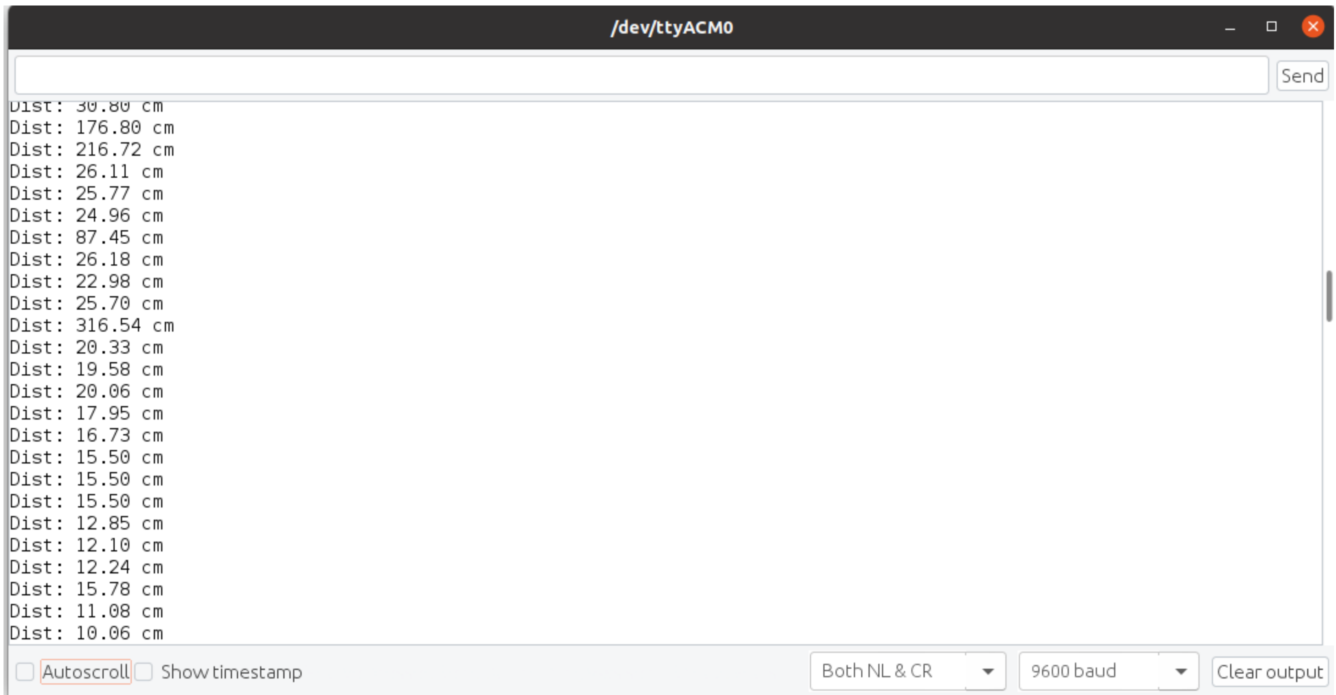

Figure 16: Distance measurements from the rangefinder

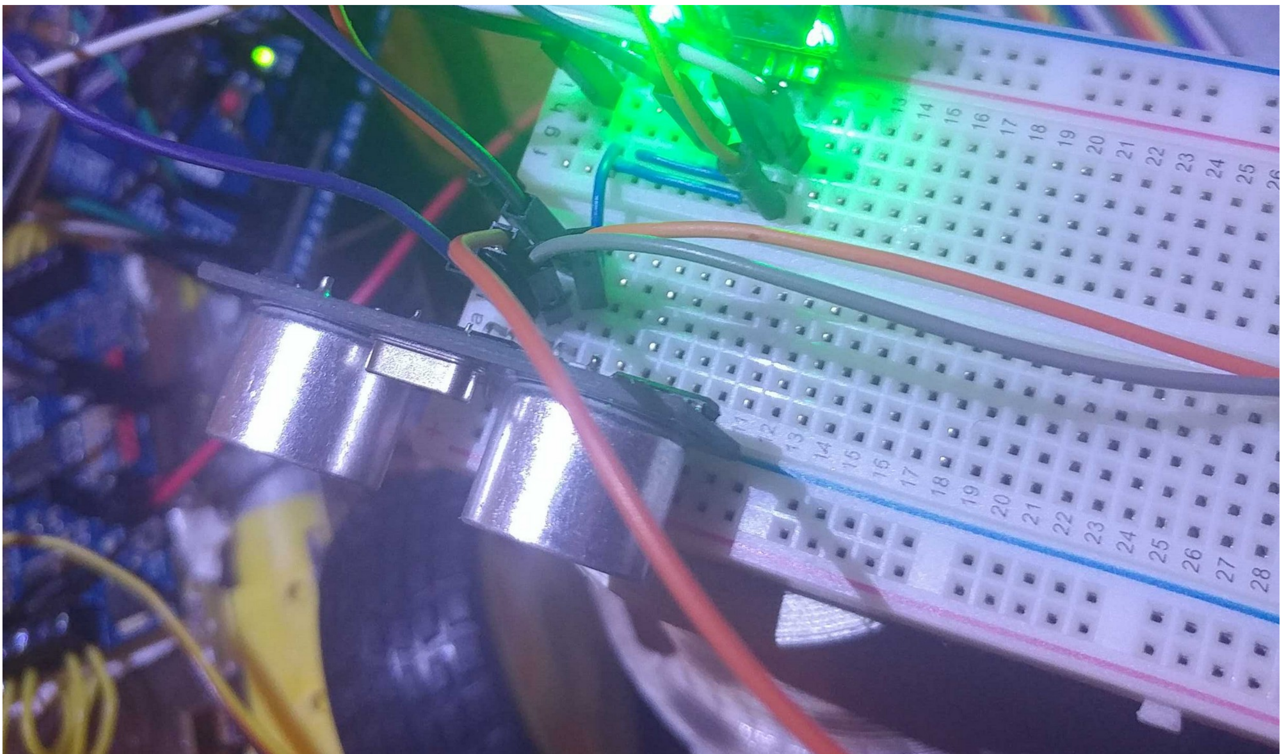

Figure 17: The breadboarded setup of the rangefinder

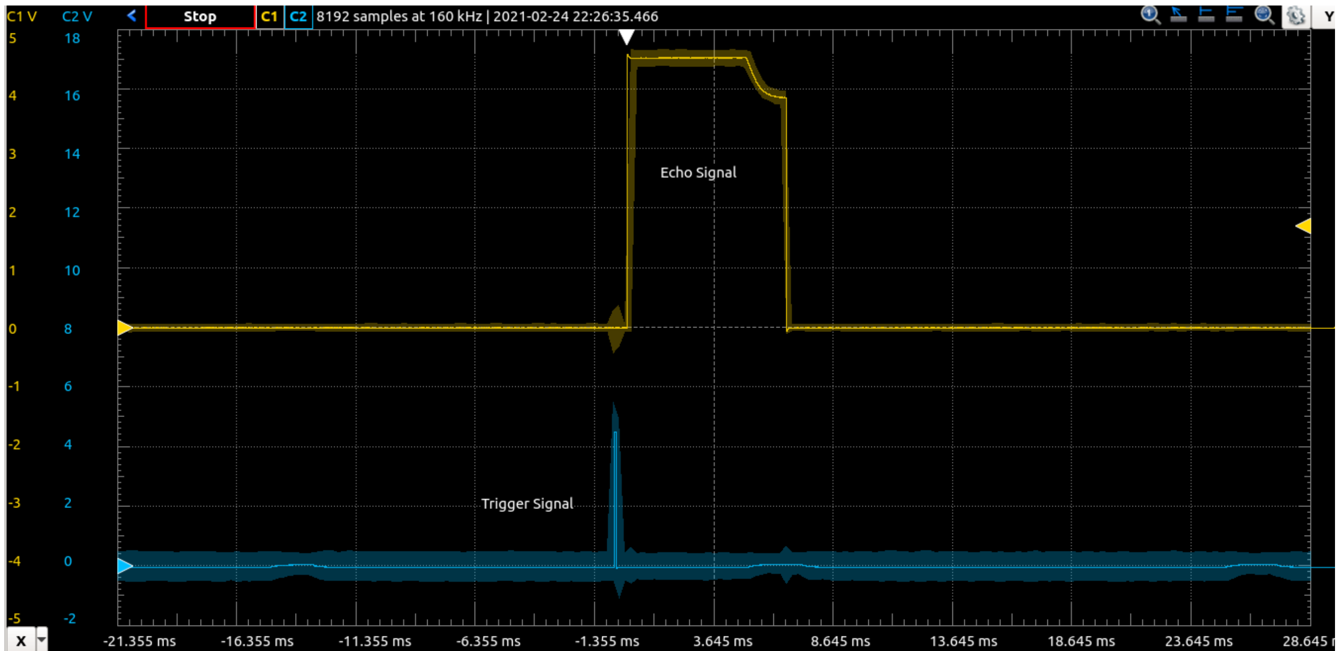

Figure 18: Oscilloscope capture of the rangefinder operation

In figure 18, the yellow trace shows the echo signal from the rangefinder and the blue signal shows the trigger pulse from the Arduino.

Conclusion

Although overall this lab was a success, I did encounter some setbacks along the way. The rangefinder provided most of the problems with it's tight timing requirements. I had issues timing the measure method with the interrupt, and had to try a few things before deciding on a simple state system. I also encountered a problem where the measure method would start returning garbage after a while. This turned out to be due to my usDelay variable was too small and was overflowing. The gripper and line sensors were mostly straightforward, although I did have a line sensor which seemed extremely "blind", even when adjusted. I eventually just swapped that line sensor out with a different one to make a better match with the second line sensor.

In this lab I used how to use three very useful and powerful devices. The two sensors, the rangefinder and the line sensor, can be used in a lot of different applications. The line sensor can also be used for other purposes, including things like heartbeat monitoring, similarly to how smart watches can measure beats per minute. The gripper kit and servo are also very useful building blocks for other projects; they could be used on robotic arms, other robots, tool changers, and a host of other things. Additionally, for every device used thus far, the C++ classes provide an easy future interface for it. Even for other projects. The experiencing *writing* the C++ classes is useful experience for Object Orientated Programming in general. From my experience of reading source code, classes are often used as a way to interface with hardware in the real world while proving an easy abstraction to the programmer.

Appendix

Lab3.ino

```
#include "gripper.h"
#include "lineSensor.h"
#include "rangeFinder.h"

GripperServo *servo;

LineSensor *ls1;
LineSensor *ls2;

RangeFinder *distSensor;

Servo s = Servo();

int lastState = 0;

void setup() {
  Serial.begin(9600);
  Serial.println("Lab 3 test bench");

  // Instantiate servo
  servo = new GripperServo(9);

  // // Instantiate Sensors
  ls1 = new LineSensor(4);
  ls2 = new LineSensor(5);

  // delay(500);
  // servo->Open();
  // delay(500);
  // // servo->Close();

  // Instantiate the distance sensor
  distSensor = new RangeFinder(7, 2);
  attachInterrupt(digitalPinToInterrupt(2), randISR, CHANGE);
  interrupts();
}

void randISR() {
```

```
  distSensor->EchoEvent();
}

void loop() {
  // Close servo when both line sensors see black.
  if (ls1->OnBlack() && ls2->OnBlack()) {
    if (lastState != 1)
      Serial.println("Both sensors on black, closing gripper");
    servo->Close();
    lastState = 1;
  } else if (ls1->OnBlack() || ls2->OnBlack()){
    // Use the distance sensor if only one sees black
    if (lastState != 2)
      Serial.println("One sensor on black, half-opening gripper");
    float dist = distSensor->MeasureCm();

    // Map the distance to the servo
    int angle = map(dist, 3, 22, 72, 145);
    if (angle > 145)
      angle = 145;

    servo->SetAngle(angle);
    lastState = 2;
  } else {
    if (lastState != 3)
      Serial.println("Neither sensor on black, opening gripper");
    servo->Open();
    lastState = 3;
  }
  delay(100);
}
```

gripper.h

```
#include <Servo.h>

#define CLAMP_ANGLE 72
#define OPEN_ANGLE 145

class GripperServo
{
  public:
    // Constructor
    GripperServo(int pin);
```

```
    void Open();
    void Close();
    void HalfOpen();
    void SetAngle(int angle);

  private:
    Servo internalServo;

};
```

gripper.cpp

```
#include "gripper.h"


void GripperServo::Open() {
  internalServo.write(OPEN_ANGLE);
}

void GripperServo::Close() {
  internalServo.write(CLAMP_ANGLE);
}

void GripperServo::HalfOpen() {
  internalServo.write((OPEN_ANGLE - CLAMP_ANGLE)/2 + CLAMP_ANGLE);
}

void GripperServo::SetAngle(int angle) {
  internalServo.write(angle);
}

GripperServo::GripperServo(int pin) {
  internalServo = Servo();
  internalServo.attach(pin);
  while(!internalServo.attached());
}
```

rangeFinder.h

```
#include "Arduino.h"

class RangeFinder
{
  public:
    // Constructor
```

```
    RangeFinder(int trigPin, int echoPin);

    // Call to send pulse
    // Returns time
    uint32_t Measure();
    // Returns inches
    float MeasureInch();
    // Returns cm
    float MeasureCm();

    // Gets called by interrupt
    void EchoEvent();

  private:
    // Private variables
    int echoPin;
    int trigPin;
    volatile int state;
    volatile unsigned long usDelay;
};
```

rangeFinder.cpp

```
#include "rangeFinder.h"


void RangeFinder::EchoEvent() {
  state++;
  if (state == 1) {
    usDelay = micros();
  } else if (state == 2) {
    usDelay = micros() - usDelay;
    state = 3;
  }
}

// Send a trigger
uint32_t RangeFinder::Measure() {
   long timeout = millis();

  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(50);
```

```
  digitalWrite(trigPin, LOW);
  digitalWrite(echoPin, LOW);
  state = 0;
  usDelay = 0;

  while (state != 3) {
    if (millis() - timeout >= 2000) {
      usDelay = 0;
      break;
    }
  }

  return usDelay;
}

// Returns measurement in inches
float RangeFinder::MeasureInch() {
  long result = Measure();
  return result / 74 / 2;
}

// Returns measurement in cm
float RangeFinder::MeasureCm() {
  long result = Measure();
  return result * 0.034 / 2;
}

RangeFinder::RangeFinder(int tPin, int ePin) {
  trigPin = tPin;
  echoPin = ePin;

  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}
```

lineSensor.h

```
class LineSensor
{
  public:
    bool OnBlack();
    // Constructor
    LineSensor(int pin);
```

```
  private:
    // Variables
    int inputPin;

};
```

lineSensor.cpp

```
#include "Arduino.h"
#include "lineSensor.h"


bool LineSensor::OnBlack() {
  if (digitalRead(inputPin)) {
    return true;
  }

  return false;
}

LineSensor::LineSensor(int pin) {
  inputPin = pin;
  // Attach to pin
  pinMode(pin, INPUT);
}
```

Lab 4

Introduction

The objective of this lab is to provide a PID C++ class to run the motors at a consistent speed, regardless of load or input voltage. Additionally, a class is needed to make the robot follow a line using the line sensors set up in a previous lab.

Part 1 – PI(D) Control

Overview

It is not a requirement for this lab to implement the derivative part of the PID control loop. The reason given is that the derivative component tended to be too noisy compared to the integral and proportional components.

The purpose of the PID control class is to take input from the wheel encoders on the front of the robot and use that information to maintain a constant speed at the wheels. Ordinarily, the speed would vary for a given PWM value as voltages and motor loads changed. For example, as the batteries drained, the robot would slow down if pure PWM values were used only.

Code Explanation

Function Name	**Description**
PIDSpeedControl(int motorN, int encPin, int loopPeriod, int motor2N);	The class constructor. Upon initialization, the constructor adds new instances of the ChassisMotor classes required for controlling two motors (one for each side). It instantiates an EncoderClass with the given pin to monitor the motor encoder, and sets the initial values for the PID control loop and debug print mode.
void setSpeed(int rpm);	Sets the desired RPM that the PID loop will attempt to reach. RPM values can be negative to command a reverse direction.
void update();	Updates the PID loop and recalculates everything. If allowed, PID values will be printed at this point. The PID loop reads the current RPM, calculates the error from the requested RPM, calculates the integral and proportional values, adds them together, maps to 0-255, and applies the new drive level to the motors. It will also shut down (roll) motors if the requested speed is 0, and set them to go forward or backward depending on the sign of the requested RPM.
void setkI();	Sets the integral coefficient of the PID loop.
void setkP();	Sets the proportional coefficient of the PID loop.
void setkD();	Sets the derivative coefficient of the PID loop.
void encUp(uint32_t n);	Tells the underlying encoder class that a rising edge occurred, and passes the current count to the class as well (n).

int getRPM();	Returns the current RPM as returned by the encoder.
uint16_t getPwm();	Returns the current requested PWM value by the PID loop. (Used by the slave motors that don't have encoders)
void togglePrint();	Toggles the internal boolean value for whether or not to print out the requested speed, the actual speed, and the current PWM value.

For the most part, the update() method does all of the work for the PID control class. It has to be called externally and is not setup to be called on an interrupt.

Tuning

Tuning was done as per the guide available at https://pidexplained.com/how-to-tune-a-pid-controller/.

1. Tune up the proportional coefficient until the control starts to oscillate. Take the value that causes oscillations, cut it in half, and use that as the initial proportional coefficient. A good value for mine seems to be about 3.0. Major oscillations seem occur around 10.0, but minor oscillations would occur at low RPMs as low as 6.0.
2. Slowly increment the integral coefficient until a desired responsiveness is achieved, without the control oscillating. Mine turned out to be about 2.0.
3. Done! Test and tune further as needed.

Figures and Results

Figure 19: PID control loop results for a complex command pattern

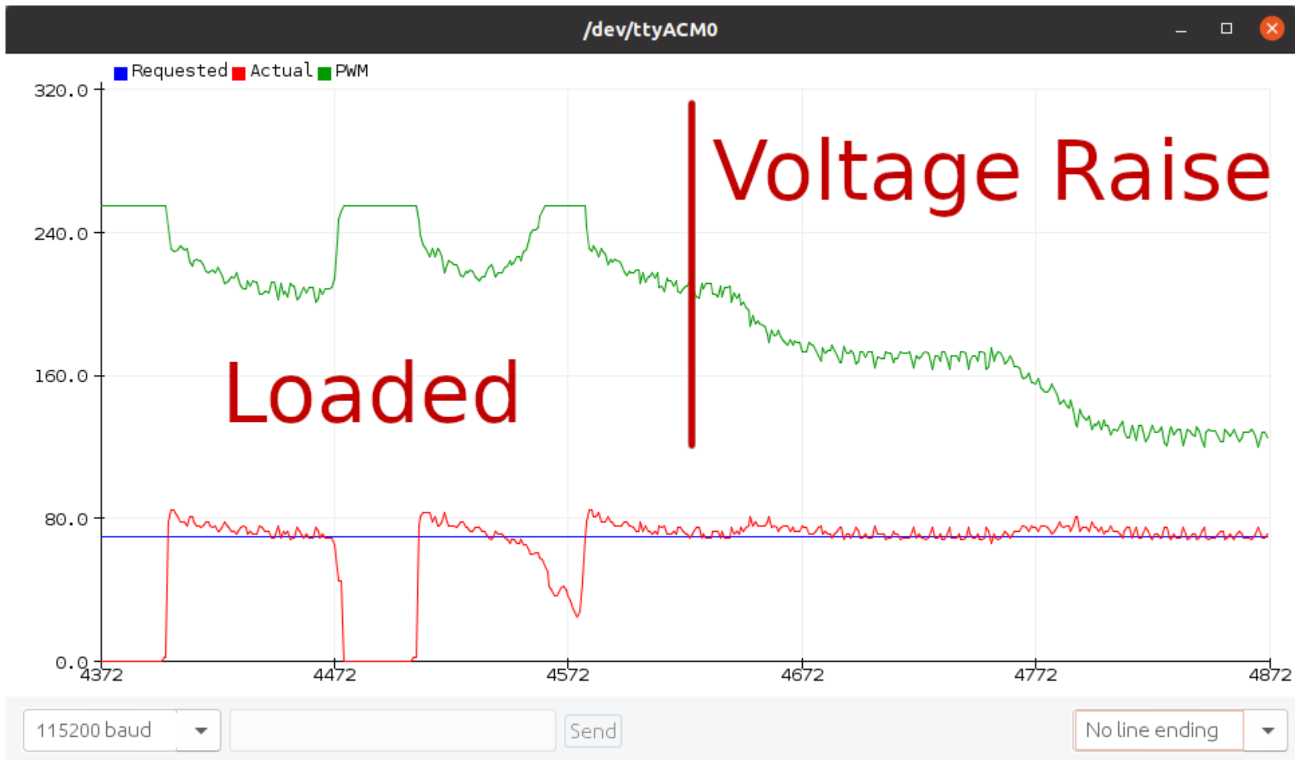

Figure 20: PID control in action - Loading the wheel until stall, and raising the voltage on the power supply.

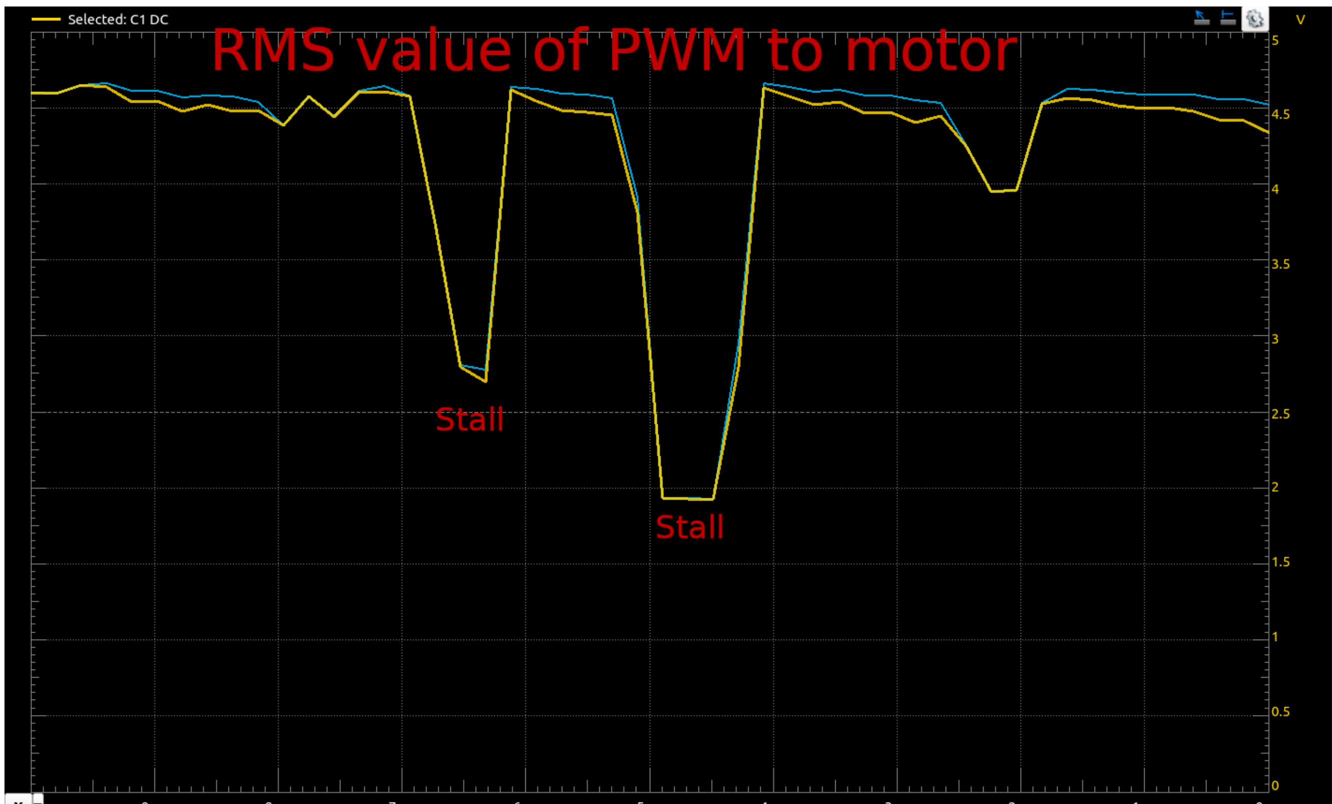

Figure 21: Waveforms logger capture of the RMS value of the PWM going to the motors, yellow line.

You can see in Figure 21 how the power supply drops the voltage when the motors stall, due to over-current protection. On the very right hand side of Figure 21 you can start to see the load on the motors diminishing and the PWM going down appropriately to maintain the same RPM.

Part 2 – Line Following

Overview

The line follower class has to control both sides of the robot – all four wheels. Two of motors are slaves to the motors with encoders, and are handled in the PID control class. Therefore, the line follower class just has to command two class instances to adjust speed and position.

Code Explanation

My approach to this wasn't entirely in-spec to the lab manual. The original objective was to have both line sensors centered in the robot and normally off, when over the black electrical tape used as the line. This way, you can know for sure when you are on the line, and you turn away from lighter readings to return to the line. In this way, you can even lose the line entirely, and so long as you know which way the line went, you can find it again.

In the end, I opted to place the line sensors on either side of the line. This way, if the line strays under one of the sensors, you turn towards the sensor that read the line, placing the line back between the sensors. Instead of polling the sensors, I attached a digital interrupt to each to improve accuracy and hopefully make the robot less likely to wander off.

The line following routine itself consists of two core control principles, slowing down the inside wheels and speeding up the outside wheels. I found that for reliability across multiple speeds the inside should slow down while the outside speeds up, and the differential should increase the longer the line is lost. With my control algorithm it's very important to respond as quickly as possible, and to never let the line pass completely under a sensor.

Function Name	Description
LineFollower(PIDSpeedControl *control1, PIDSpeedControl *control2, int loopPeriod);	The class constructor. The two PID control classes are passed to the constructor as pointers, and are placed into private internal pointers. Digital interrupts are setup for the line sensors, initial speed is set, loop variables are set, and the debug is disabled.
void update();	Updates the control loop and sets the requested speed of the sub classes being controlled.
void setSpeed(int newSpeed);	Requested RPM to cruise at.
void enable();	Enables the line-following functionality. Otherwise, if disabled, the line follower class just commands both sides to run at the same RPM.
void disable();	Disables the line following functionality.

void togglePrint()	Enables/Disables printing off the loop variables every time update() is called.
Private Functions	
static void interruptLineSensor();	Called when one of the digital pins attached to the line sensors changes states. Updates variables checked by the control loop to determine the status of the line sensors.
void speedUpSide1();	Speeds up the outer wheels in a turn, and slows down the inner wheels.
void speedUpSide2();	Speeds up the outer wheels in a turn, and slows down the inner wheels.
void resetSides();	Requests that both sides return to the requested RPM.

Figures and Results

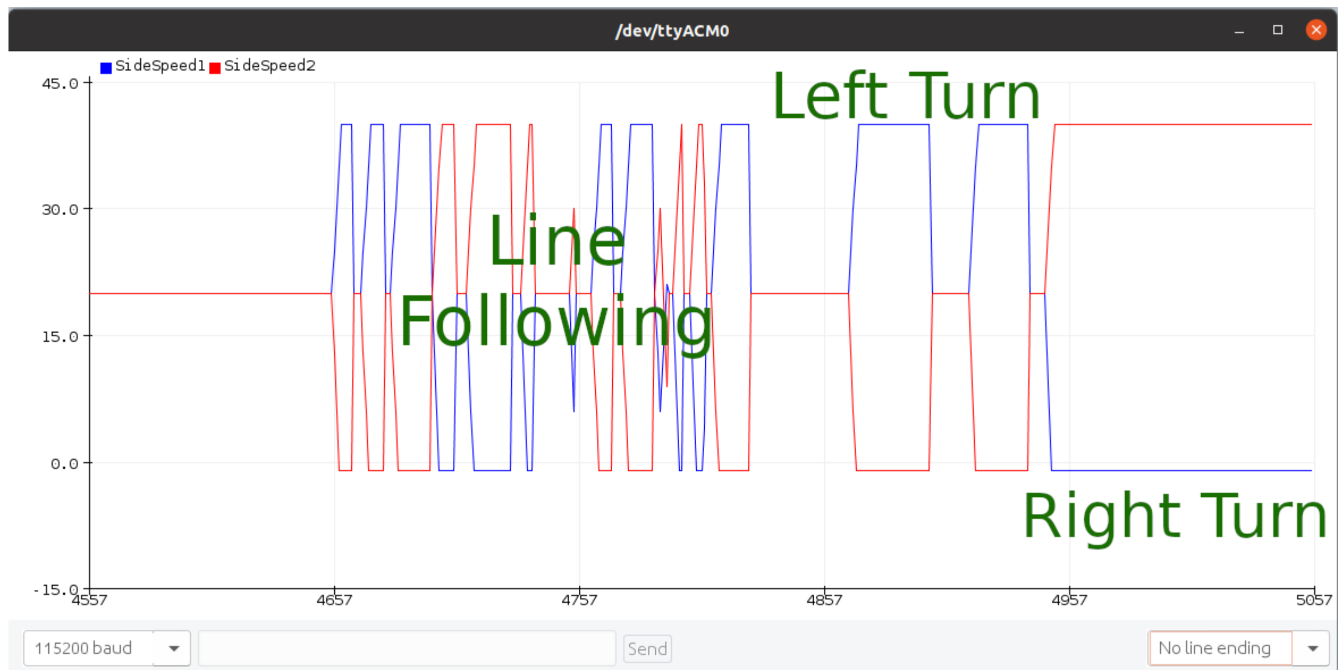

Figure 22: RPM on the line follower

Figure 22 shows the line follower class in action from the serial monitor. The lines are labeled in the upper left of the figure, and represent the RPM as requested per-side by the line follower class. Note that the slope is different for the side slowing down and speed up.

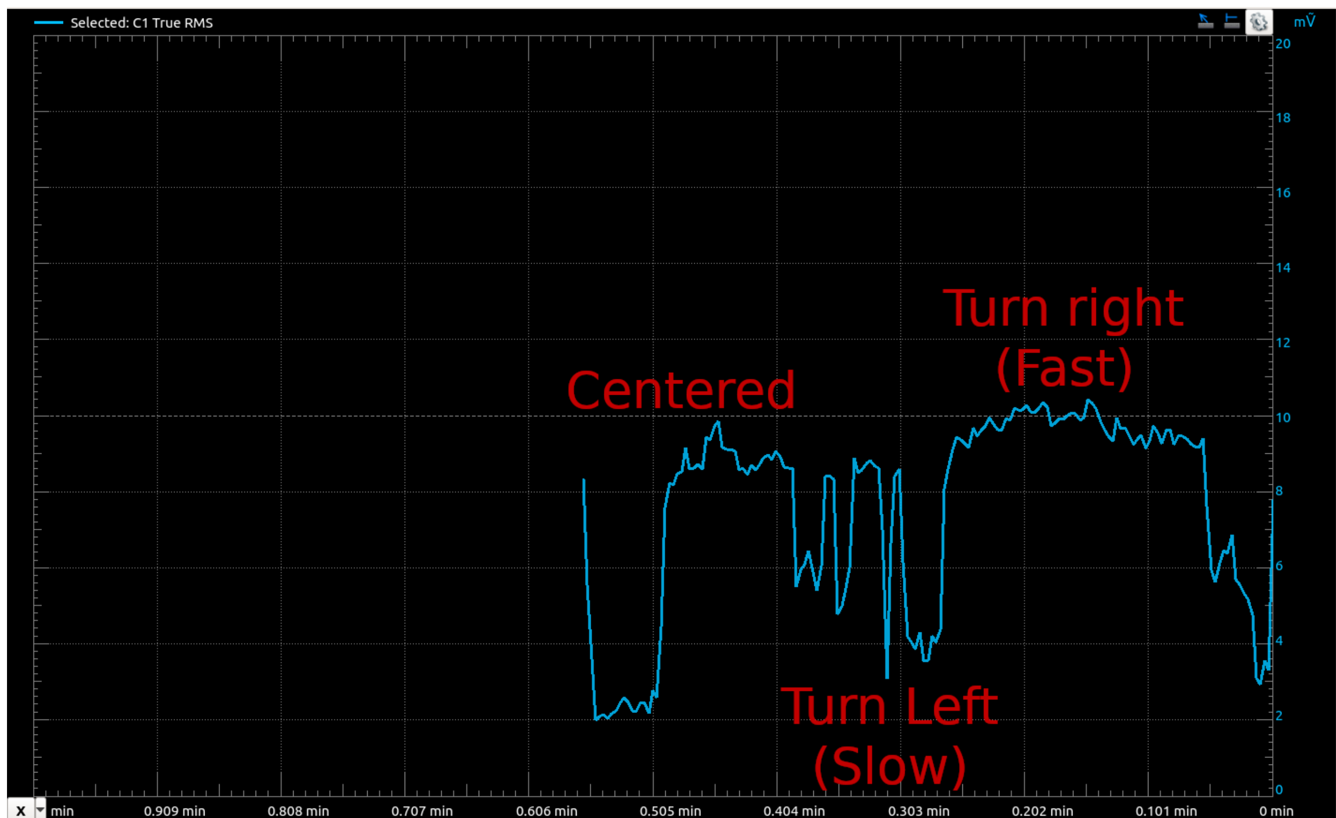

Figure 23: DC average (RMS) of the wheel drive during turns

Figure 23 shows the true response of the motors as the robot turns. Notice how the inside slows drastically (and in reality, reverses) while the outside wheels only increase in speed slightly. If the robot is traveling quickly enough, both sides will actually slow from the requested value to prevent overshooting the line.

Tuning

The majority of the turn is caused by the inside wheel slowing and reversing. Hence, tuning consists of determining the rate of deceleration required to make reliable corners without overshooting the turn. The rate of deceleration is increased until the robot can follow a sharp, 90 degree corner. The radius of my test corner is about 3-4 inches, slightly less than the length of the robot.

Conclusion

So far, this has been the most educational and the most difficult lab in this project. I learned how to get a rudimentary PID control loop working with essentially servo control of motors. I also learned a lot about line following and the challenges involved with it. This lab was great experience in motion control since I had to hit a defined speed and follow a defined path with real world, physically limited, hardware.

The line follower took at least as much time as the PID loop, and really put the rest of my code through the wringer. When trying to have both line sensors centered on the line, the robot would easily lose the line because both line sensors would lose the line at the same without knowing which sensor saw it last. I tried to combat this issue by adding pin interrupts to the line sensor code instead of simple polling, but even that didn’t work. I believe I didn’t properly handle the pin interrupts, and they may have worked in the end, but I made hardware changes to place the sensors on both sides of the line and went with the new method anyway.

The PID loop was mostly straightforward. Tuning proved difficult when dealing with fast response times, slow events were quite easy to deal with, but to follow a line with any amount of speed I had to tweak the code to be as fast as possible. This mostly consisted of making the encoder return quicker, and clamping the PID values so they didn’t wander off to large values while the motors were stalled at full power already. I would have liked to get an even more responsive loop set up, but I was running into issues where the motors would lurch mechanically and essentially oscillate as they sped up. I think I would have needed higher quality hardware to get a faster PID system. Also, adding in the derivative term would have helped a lot too, which probably also required more accurate hardware and faster loop times.

Overall, this lab was very educational and the challenges I encountered taught me many lessons. I continued learning and implementing new things in C++ and now the robot is nearing completion with two of it’s major features operational.

Appendix

Lab4.ino

```
#include "PIDSpeedControl.h"
#include "lineFollow.h"

PIDSpeedControl *leftMotors;
PIDSpeedControl *rightMotors;

LineFollower *lineFollow;

uint32_t tcount4 = 0;
uint32_t tcount5 = 0;

uint32_t tFlop = 0;
int lastSpeed = 70;

uint32_t lastTime = 0;

void setup() {
  Serial.begin(115200);
  delay(100);

  // Instantiate classes
  leftMotors = new PIDSpeedControl(4, 48, 50, 1);
  //leftMotors->setSpeed(70); // RPM
  rightMotors = new PIDSpeedControl(3, 49, 50, 2);
  //rightMotors->setSpeed(70); // RPM
  lineFollow = new LineFollower(leftMotors, rightMotors, 50);
  lineFollow->setSpeed(20);
  lineFollow->enable();

  //leftMotors->setSpeed(20);
  lineFollow->togglePrint();
  //leftMotors->togglePrint();
}

void loop() {
  lineFollow->update();
  //leftMotors->update();
  /*
```

```
  if (millis() - 3000 > lastTime) {
    /*
    leftMotors->setSpeed(0);
    for (int x = 0; x < 30; x++) {
      delay(50);
      leftMotors->update();
    }*/
    /*
    for (int x = 0; x < 100; x++) {
      delay(50);
      leftMotors->setSpeed(x);
      leftMotors->update();
    }
    leftMotors->setSpeed(70);
    lastTime = millis();
  }*/

  delay(50);
}

// Timer setups
ISR(TIMER4_CAPT_vect) { // Pin 49
  tcount4 = (tcount4 & 0xFFFF0000) | TCNT4;
  leftMotors->encUp(tcount4);
}

ISR(TIMER5_CAPT_vect) { // Pin
  tcount5 = (tcount5 & 0xFFFF0000) | TCNT5;;
  rightMotors->encUp(tcount5);
}

ISR(TIMER4_OVF_vect) {
  // Running this vector auto-clears the overflow
  // Add to the count
  tcount4 = tcount4 + 0x10000;
}

ISR(TIMER5_OVF_vect) {
  // Running this vector auto-clears the overflow
  // Add to the count
  tcount5 = tcount5 + 0x10000;
}
```

lineFollow.h

```
#include "Arduino.h"
#include "PIDSpeedControl.h"
#include "lineSensor.h"
// PID speed control
#define INTEGRAL_MAX   6000
#define INTEGRAL_MIN  -6000

#define SPEED_UP 5
#define SPEED_DOWN 7

#define MAX_RPM 40
#define MIN_RPM -1

class LineFollower {
public:
  // Constructor
  LineFollower( PIDSpeedControl *control1, PIDSpeedControl *control2, int loopPeriod);

  // Update Function
  void update();

  // Set requested Speed
  void setSpeed(int newSpeed);

  // Enable/Disable
  void enable();
  void disable();

  // Debugging
  void togglePrint();

private:
  // Static method called via an interrupt ISR.
  static void interruptLineSensor();
  // Other line sensor variables
  static bool side1;
  static bool side2;

  // Correction methods
  void speedUpSide1();
  void speedUpSide2();
```

```
  void resetSides();

  // Ms
  uint32_t stepTime;

  // Class pointers
  PIDSpeedControl *motor1;
  PIDSpeedControl *motor2;
  static LineSensor *ls1;
  static LineSensor *ls2;

  // Speeds are in RPM
  int requestedSpeed;
  int turnSpeed;
  double speedSide1;
  double speedSide2;

  // PID control variables
  int loopTime;
  int timeError;

  // State/Status variables
  bool enabled;
  uint8_t sideLastSeen;

  // Debugging
  bool debugPrint;

};
```

lineFollow.cpp

```
#include "lineFollow.h"


LineSensor *LineFollower::ls1 = new LineSensor(18);
LineSensor *LineFollower::ls2 = new LineSensor(19);

bool LineFollower::side1 = false;
bool LineFollower::side2 = false;

void LineFollower::interruptLineSensor() {
  side1 = !ls1->OnBlack();
  side2 = !ls2->OnBlack();
```

```
}

// Update control
void LineFollower::update() {
  // Check if we are on the line,
  if (side1 && side2) {
    // Go normal speed
    sideLastSeen = 0;
    resetSides();
  } else if (side1 || side2) {
    // Go towards the correct side
    // Record the side we saw last
    if (side1) {
      sideLastSeen = 1;
      speedUpSide2(); // Go towards side 1
      //speedSide1 += 10;
    } else {
      sideLastSeen = 2;
      speedUpSide1(); // Go towards side 2
      //speedSide2 += 10;
    }
  } else { // Both on white
    if (sideLastSeen == 1) {
      speedUpSide2(); // Go towards side 1
    } else if (sideLastSeen == 2) {
      speedUpSide1(); // Go towards side 2
    } else if (sideLastSeen == 0) {
      // Lost the line, stop the robot
      speedSide1 = 0;
      speedSide2 = 0;
    }
  }

  if (enabled) {
    motor1->setSpeed(speedSide1);
    motor2->setSpeed(speedSide2);
  } else {
    motor1->setSpeed(requestedSpeed);
    motor2->setSpeed(requestedSpeed);
  }

  motor1->update();
```

```
  motor2->update();

  if (debugPrint) {
    Serial.print(speedSide1);
    Serial.print(", ");
    Serial.println(speedSide2);
  }
}

// Update the requested speed
void LineFollower::setSpeed(int newSpeed) {
  requestedSpeed = newSpeed;
}

// Increase speed on motors side 1
void LineFollower::speedUpSide1() {
  speedSide1 += SPEED_UP;
  if (speedSide1 > MAX_RPM) {
    speedSide1 = MAX_RPM;
  }

  speedSide2 -= SPEED_DOWN;
  if (speedSide2 < MIN_RPM) {
    speedSide2 = MIN_RPM;
  }
}

// Increase speed on motors side 2
void LineFollower::speedUpSide2() {
  speedSide2 += SPEED_UP;
  if (speedSide2 > MAX_RPM) {
    speedSide2 = MAX_RPM;
  }
  speedSide1 -= SPEED_DOWN;
  if (speedSide1 < MIN_RPM) {
    speedSide1 = MIN_RPM;
  }
}

void LineFollower::resetSides() {
  speedSide1 = requestedSpeed;
  speedSide2 = requestedSpeed;
```

```
}

void LineFollower::enable() {
  enabled = true;
}

void LineFollower::disable() {
  enabled = false;
}

void LineFollower::togglePrint() {
  debugPrint = !debugPrint;
  if (debugPrint) {
    Serial.println("SideSpeed1, SideSpeed2");
  }
}

// Class Constructor
LineFollower::LineFollower( PIDSpeedControl *control1, PIDSpeedControl *control2,
                            int loopPeriod) {

  // Tie in our motors
  motor1 = control1;
  motor2 = control2;
    // Setup line sensors
  attachInterrupt(digitalPinToInterrupt(18), interruptLineSensor, CHANGE);
  attachInterrupt(digitalPinToInterrupt(19), interruptLineSensor, CHANGE);

  loopTime = loopPeriod;

  // Control variables
  enabled = false;

  resetSides();
  debugPrint = false;
}
```

PIDSpeedControl.h

```
#include "motors.h"
#include "encoder.h"
#include "Arduino.h"

#ifndef PIDCONTROL_H
```

```
#define PIDCONTROL_H
// PID speed control
#define INTEGRAL_MAX   3000
#define INTEGRAL_MIN  -3000

#define PROP_MAX 3000
#define PROP_MIN -3000


#define DEFAULT_KP 3
#define DEFAULT_KI 2
#define DEFAULT_KD 0

// Basic form of PID loop
/*
Err = Sp – PV

P = kP x Err

It = It + (Err x kI x dt)

D = kD x (pErr – Err) / dt

pErr = Err

Output = P + It + D
*/

// PID Speed control
// Calculates the proper PWM value for a motor based on a commanded speed and an encoder input.
class PIDSpeedControl {
  public:
    // Class constructor
    PIDSpeedControl(int motorN, int encPin, int loopPeriod, int motor2N);

    // Set desired speed
    void setSpeed(int rpm);
    // Update/Recalculate
    void update();
    // Return current error
    // Set integral coefficient
    void setkI();
```

```
    // Set proportional coefficient
    void setkP();
    // Set derivative coefficient
    void setkD();
    // Encoder edge
    void encUp(uint32_t n);
    // Return encoder reading
    int getRPM();
    // Return current PWM
    uint16_t getPwm();
    // Toggle Printing
    void togglePrint();

  private:
    ChassisMotor *motor;
    ChassisMotor *motorSlave;
    EncoderClass *motorEncoder;
    uint8_t encPin;

    // Speeds are in RPM
    int actualSpeed;
    int requestedSpeed;

    // PID control variables
    int pError;   // Previous Error
    double kP;
    double kI;
    double kD;
    int integral;
    int proportional;
    int derivative;
    int loopTime;
    int drive;

    // Other
    bool printData;
};

// Line follower control
// Steers the robot based on the line follower sensors and by commanding
// speeds with the PID speed control
#endif PIDCONTROL_H
```

PIDSpeedControl.cpp

```
#include "PIDSpeedControl.h"

// Recalculate the required PWM value to maintain our request speed.
void PIDSpeedControl::update() {
  // Read actual speed:
  // If our previous speed was zero, the check function will take a while to
  // return because it will try and wait for the encoder to update. To respond
  // faster in the these situations, we simply say if the last motor encoder
  // check returned 0, to just set the actual speed to 0. Then, next time
  // we actually read the encoder
  actualSpeed = motorEncoder->encRPM();

  // PID Calculations
  int error = 0;
  if (requestedSpeed > 0 ) {
    error = requestedSpeed - actualSpeed;
  } else {
    error = -1*requestedSpeed - actualSpeed;
  }
  proportional = kP * error * loopTime/10;
  integral += kI * error;
  //derivative = kD * (pError - error)/loopTime;

  // Cap the integral
  if (integral > INTEGRAL_MAX) {
    integral = INTEGRAL_MAX;
  } else if (integral < INTEGRAL_MIN) {
    integral = INTEGRAL_MIN;
  }

  // Cap proportional
  if (proportional > PROP_MAX) {
    proportional = PROP_MAX;
  } else if (proportional < PROP_MIN) {
    proportional = PROP_MIN;
  }

  drive = integral + proportional;

  // Map drive to 0-255
  drive = (255.0*drive)/3000.0;
```

```
if (drive > 255) {
  drive = 255;
} else if (drive < 0) {
  drive = 0;
}

pError = error;

if (printData) {
  Serial.print(requestedSpeed);
  Serial.print(",");
  Serial.print(actualSpeed);
  Serial.print(",");
  Serial.println(drive);
}
/*
Serial.print("Error: ");
Serial.println(error);

Serial.print("Drive: ");
Serial.println(drive);

Serial.print("Integral: ");
Serial.println(integral);

Serial.print("Proportional: ");
Serial.println(proportional);

Serial.print("Actual: ");
Serial.println(actualSpeed);*/

// Command the motors
if (requestedSpeed == 0 ) {
  motor->roll();
  motor->setPwm(0);
  motorSlave->roll();
  motorSlave->setPwm(0);
} else if (requestedSpeed > 0){
  motor->forward();
  motor->setPwm(drive);
  motorSlave->forward();
```

```
    motorSlave->setPwm(drive);
  } else if (requestedSpeed < 0){
    motor->reverse();
    motor->setPwm(drive);
    motorSlave->reverse();
    motorSlave->setPwm(drive);
  }

}


int PIDSpeedControl::getRPM() {
  return motorEncoder->encRPM();
}

uint16_t PIDSpeedControl::getPwm() {
  return drive;
}

void PIDSpeedControl::setSpeed(int rpm) {
    requestedSpeed = rpm;
}

void PIDSpeedControl::encUp(uint32_t n) {
  motorEncoder->up(n);
}

void PIDSpeedControl::togglePrint() {
    Serial.println("Requested, Actual, PWM");
    printData = !printData;
}



// Class constructor
PIDSpeedControl::PIDSpeedControl(int motorN, int encPin, int loopPeriod, int motor2N) {
    // Instantiate Motor & Encoder
    motor = new ChassisMotor(motorN);
    motorSlave = new ChassisMotor(motor2N);
    motorEncoder = new EncoderClass(encPin);

    kP = DEFAULT_KP;
```

```
    kI = DEFAULT_KI;
    kD = DEFAULT_KD;

    integral = 0;
    proportional = 0;
    derivative = 0;
    loopTime = loopPeriod;

    printData = false;

}
```

Lab 5

Objective

This is the last chapter of this lab book. The objective of the fifth and final lab is to test the complete robotic system in an obstacle course designed to make use of all of the previous labs.

Part 1 – Testing PID Speed Control

Overview

The objective for this part of the lab is to drive the robot on a treadmill with two different degrees of inclination. Once the robot's speed is matched to the treadmill, the incline is increased and the robot is supposed to maintain it's position by means of PI control compensating for the increased load on the motors. The rover operates entirely autonomously in this obstacle.

Approach

The PID loop was tuned on a flat surface and with the Serial Plotter to maintain a constant speed under load. The rover was then aseembled completely and placed on the test treadmill. The speed of the rover was matched as closely as possible to speed of the treadmill and then the incline of the treadmill was increased from 0 to 15 degrees.

Outcome

The rover drifted backwards slowly even with the PID control enabled, and over multiple speed ranges while I attempted to find a sweet spot. I'm not 100% why this happened but I don't have time to do further tuning of the robot. The PID control seemed to slightly outperform the PWM control on the treadmill, and both were recorded and placed on YouTube.

- PID Control: https://youtu.be/8CBSukmk5eM
- PWM Control (Straight drive): https://youtu.be/sF4mimWGHQ8

Two images from each video were analyzed (two from the PID control video are shown, figures 24 and 25).

- PID control drift: 55 pixels over 23 seconds
- PWM control drift: 158 pixels over only 13 seconds.

As such, the PID control was about 80% better than the raw PWM drive, and with a little bit of tuning could probably stay completely stationary.

Figure 24: Lower measurement of PID test

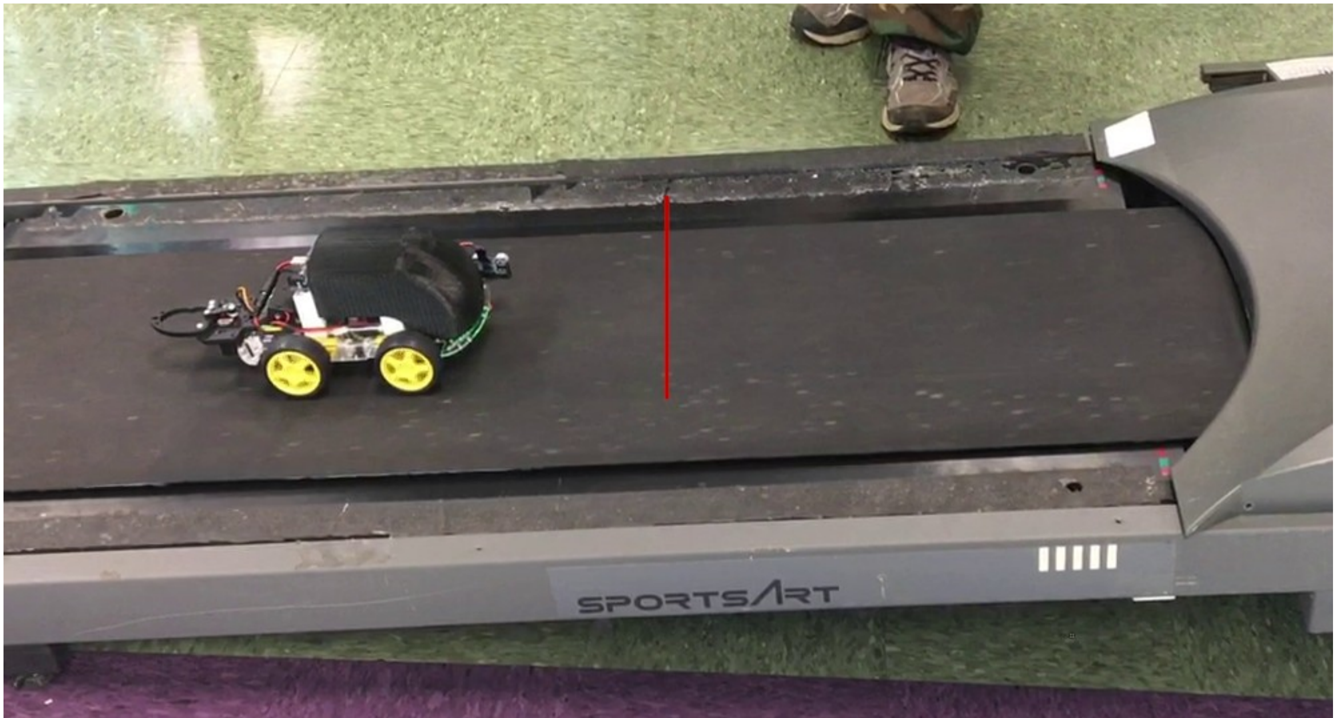

Figure 25: Upper measurement of PID test

Part 2 – Testing Line Following

Overview

In this obstacle, the rover is required to follow a black line around in a complicated path on a butcher-paper covered floor. The rover operates entirely autonomously in this obstacle and uses the line sensors on it's underside to keep track of the line.

Approach

Line sensors were placed on each side of the black line, attached to the underside of the rover. Ordinarily, both sensors are supposed to see white. However, when one sensor sees the black line, the robot turns to recenter the line between the line sensors. This approach requires the robot to go very slowly in order to not lose the line, but proved effective with a relatively simple algorithm.

A second approach was to use three sensors, two on either side, and a third in the center to detect the line. I did actually end up implementing this approach but didn't have time to tune it very well. The rover would wildly over-correct if it drifted off the line and would oscillate all the way around the path. However, it was impossible for the robot to lose the line on sharp corners, unlike the previous method. This means that much higher speeds could be used with proper tuning (more time). This approach required that the line sensors be separated from each-other with black plastic so that they wouldn't interfere. I also added the required obstacle aversion feature in this version, by means of utilizing the distance sensor. The line following obstacle is the only one which requires me to change the hardware of the rover. I have to move the third optical sensor to the bottom of the rover and move the distance sensor from the side to the front. As such, the rover's shell can't be mounted and this is observed in the second video below.

Outcome

The robot initially made it around the path in about one minute and 39 seconds (1:39). The video of the rover following the line is available here:

- Rover line follow: https://youtu.be/SrPtGkbW1HY
- Rover line follow with obstacle: https://youtu.be/HvU9DxCiQkg

The second algorithm and attempt made it around the circle in the counter-clockwise direction in about one minute and 21 seconds (1:21). The clockwise direction took about the same amount of time at 1:19. The rover does stop if an obstacle is placed in its path.

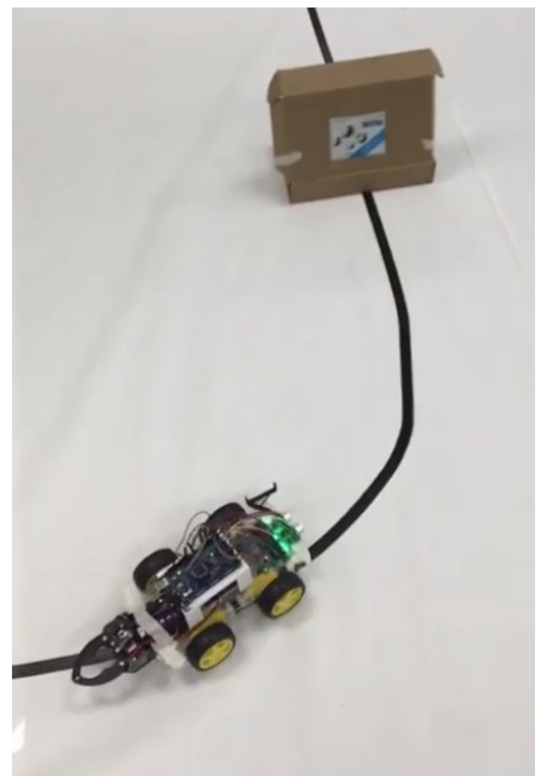

Figure 26: Rover about to encounter an obstacle

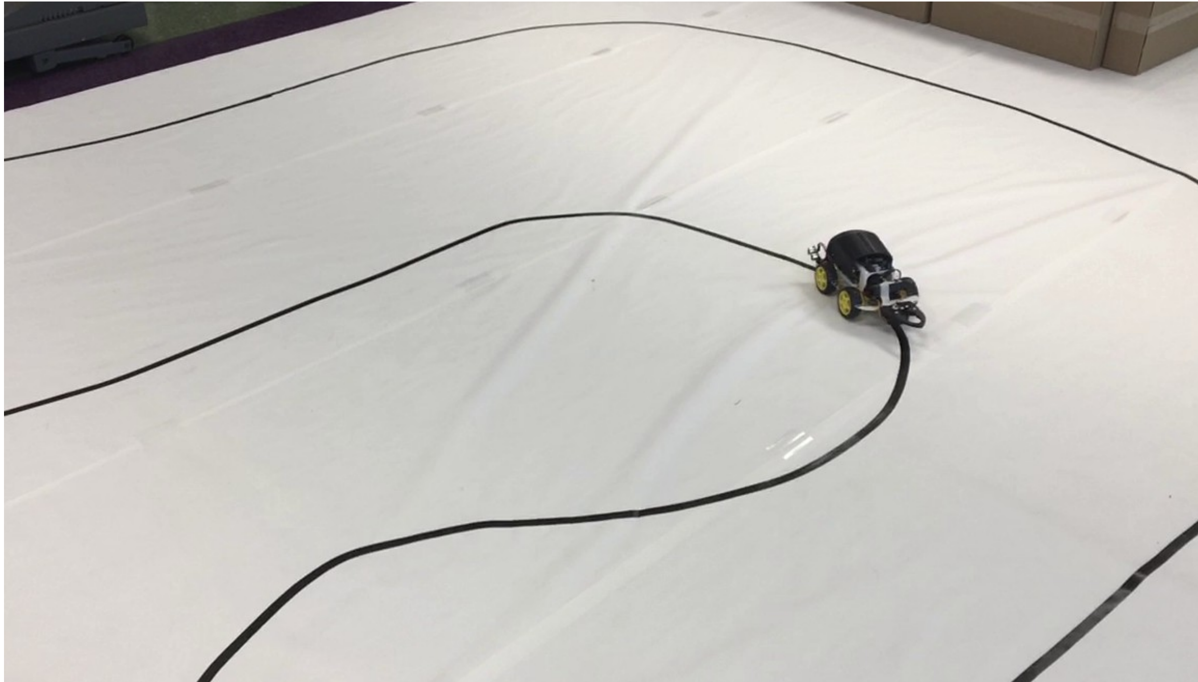

Figure 27: Rover navigating the obstacle autonomously

Part 3 – Testing Remote Control & Gripper

Overview

This obstacle requires remove control of the rover via Bluetooth. The objective is to collect three balls from a cache of them around a 90 degree bend on a table course. A ball has to be fetched and placed into the receptacle (a trash can in this case) three times.

Approach

All of the obstacles have a single Arduino sketch, and each mode can be enabled and disabled via Bluetooth for ease of testing and development (I,e, manually driving out of the maze and such). The rover was manually driven in PWM mode and the gripper was used to collect balls from a tray at the end of the short course. After a ball was retrieved I drove the rover back to the starting point and dropped off the ball, before returning for the next one.

Outcome

The rover was able to fetch all three balls and place them in the receptacle under remote control. I was able to accomplish this task in two minutes and 35 seconds (2:35). I wasn’t really attempting to go as fast as possible since the rover had already fallen off the table once.

- Video: https://youtu.be/RV81xrQot78

Later on, the turning performance was increased for both the PID control and the PWM control. To really dial in this challenge I would have liked to have acceleration profiles set up such that for high speeds the robot wouldn’t instantly lurch forward. I found that often times, fast movements would

Figure 28: Ball Fetch

cause the robot to drop the ball it was holding. The obstacle was not re-attempted.

Part 4 – Testing Maze Navigation

Overview

This obstacle consists of a simple maze of 90 degree corners and no dead ends. The rover has to complete this obstacle completely autonomously. The maze was created out of light cardboard boxes and was build on the same rover-friendly paper surface as the line following course.

Approach

At first, I attempted to use a single rangefinder mounted on the front of the robot to locate walls and avoid them. This required making precise 90 degree turns that were nearly impossible to guarantee, and the rover would inevitably run into a wall at some point in trying to navigate the maze. Rather than purchasing one or two more distance sensors to make the rover aware of every surrounding wall, I mounted the rangefinder on the side of the rover. This way, I measure the distance to the left wall and I follow it, keeping the distance between the rover and the left wall constant at about 8 inches. I mounted a third line following sensor pointing straight ahead on the front of the rover to detect when it was about to run headfirst into a wall if the maze made a 90 degree corner to the right. When the light sensor was triggered, the rover backs up a small amount and turns roughly 90 degrees to the right before continuing on. This method worked great and was very easy to code compared to other solutions.

Outcome

The rover completed the maze forwards in about 28 seconds, and in reverse in about 31 seconds. The rover did collide with the walls often, but just slightly. This happened because the light sensor wasn't calibrated correctly and was detecting the wall too late. Additionally, the gripper was mounted farther forward than was desirable.

- Video: https://youtu.be/T4Vi6kKKdlI

Figure 28 shows the setup of the rover as it navigates the maze in the forward direction. It probably would have been possible to use just the distance sensor to follow the left wall, especially if it had been mounted to gripper such that it could be rotated to look around.

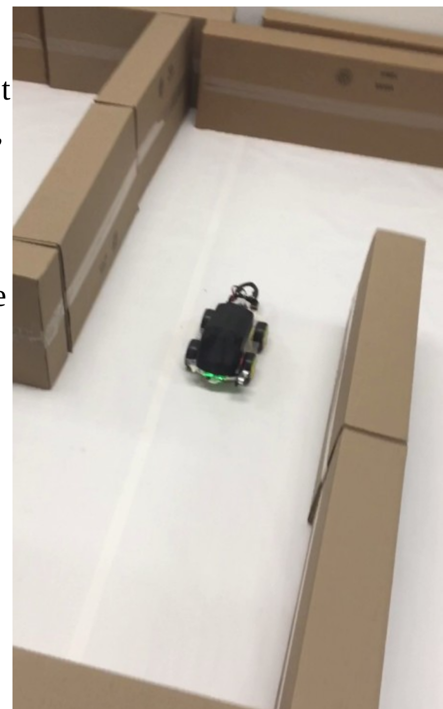

Figure 29: Rover navigating the maze

BOM

The following table represents each item and it's cost required to build my rover.

Item	**Price**	**QTY**
Line Sensor	$1.1/ea	3
DIY Robot Smart Car Chassis Kit	$17.99	1
Ultrasonic Rangefinder	$1.66/ea	1
18650 cell Battery (Includes Charger)	$10/ea	2
18650 Battery Holder	$2	1
Arduino Mega	$15.99	1
Adafruit Motor Shield V2	$22.76	1
Small breadboard	$1.5	1
Jumper Wires (~40)	$5	1
3d Printer Filament (Optional)	~$1	1
Wheel Encoder	$5	2
Gripper	$10	1
Servo	$12	1
Miscellaneous Hardware (Optional)	$2	1
Bluetooth Module	$26.99	1
Total:	**$152.19**	

Prints / Parts

Several things were designed and printed for this rover. Most notably the cover for the electronics and holders for the distance sensor and line sensors. A 3d printer pen was also used for some miscellaneous parts such as the how the motor shield was held to the frame. FreeCAD was used to design the parts and Cura was used to slice them. Smaller parts were printed on my Wife’s Monoprice Mini 3d printer and larger parts were printed on my Monoprice Maker Select V2 printer with a 0.6mm nozzle.

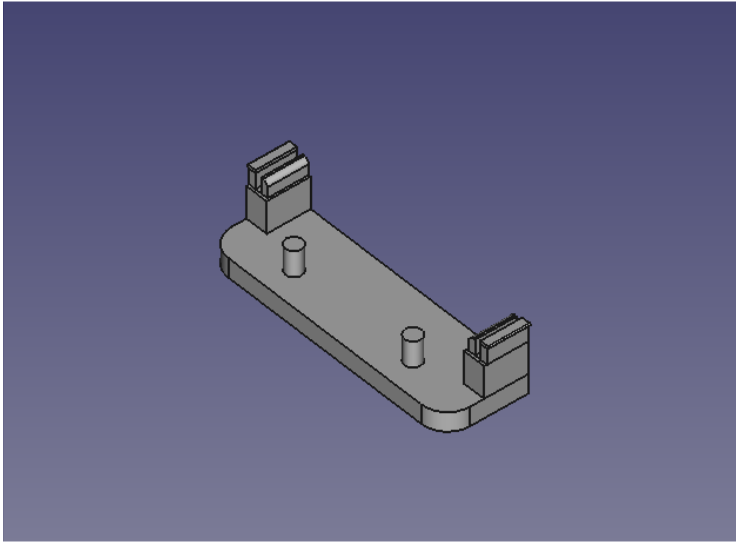

Figure 30: Holder for the gripper

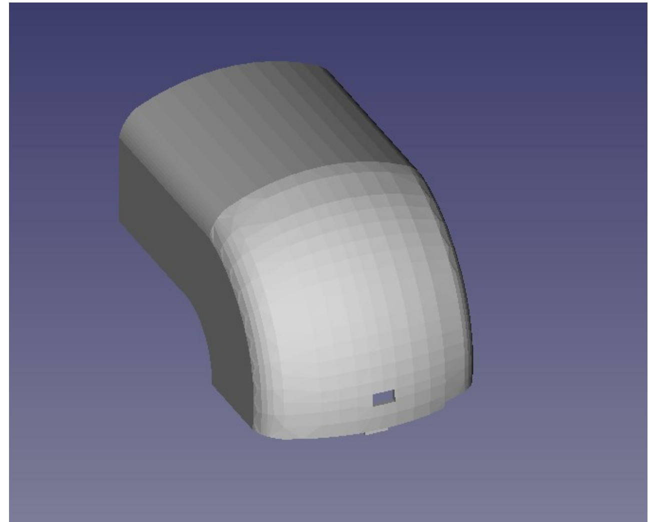

Figure 31: Electronics Cover

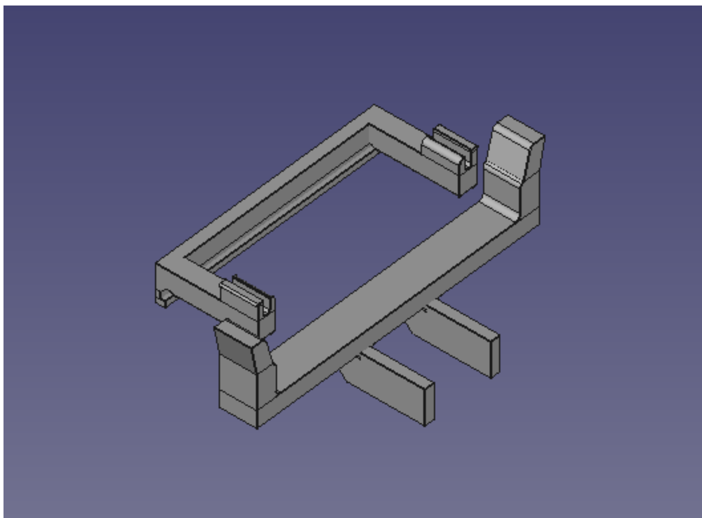

Figure 32: Line Sensor clamp and mount

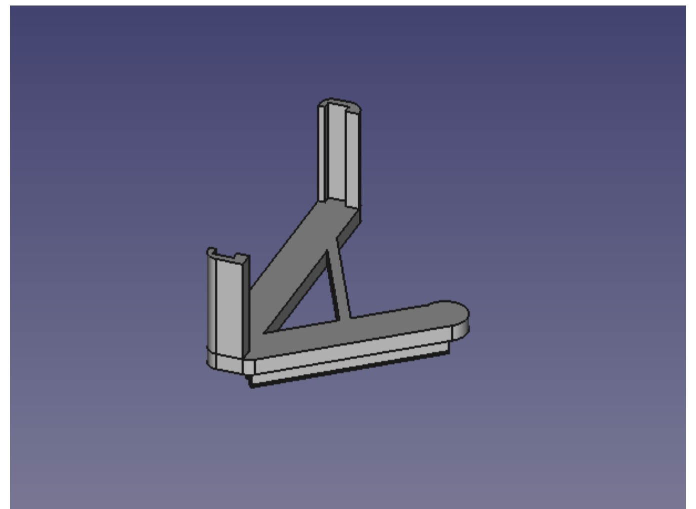

Figure 33: Distance Sensor Mount

Conclusion

In conclusion, I learned how to write and tune a PID control loop, how to write and tune a line following algorithm, how to implement remote control with a Bluetooth module, how to physically build a small rover, how to use Ultrasonic Distance sensor and optical line sensors, how to use the Adafruit motor shield, and how to use the ATMega2560's internal timers and serial busses. This was a very educational project and I'm left with a functioning device as well as modules I can use for other projects.

For future research or projects, it would be interesting to make this robot with tracks to ensure that two encoders could accurately control the robot. It would also be interesting to add a rotation or compass sensor for determining the heading of the robot. If the heading of the robot could be determined, obstacles such as the maze would be much easier to navigate with perfect-degree turns. The surrounding area could even be scanned and mapped by slowly turning around in circles and using the ultrasonic sensor to take measurements. I would also like to investigate why the PI(D) control didn't hold the robot in position perfectly.

This concludes the class project. For one-hundred and fifty-two dollars I built a rover capable of both remote control and autonomous operation including Proportional-Integral control of it's motors. The rover was able to compensate slightly for motor loads by means of encoder feedback, able to measure distances and react to them using the Ultrasonic Distance sensor, able to follow a line using optical sensors, able to grab and carry objects using a servo-controlled gripper, and able to be driven around remotely using Bluetooth.

Appendix

The final code for this project is included here.

Lab5.ino

```
#include "PIDSpeedControl.h"
#include "lineFollow.h"
#include "gripper.h"
#include "rangeFinder.h"

PIDSpeedControl *leftMotors;
PIDSpeedControl *rightMotors;

RangeFinder *distSensor;

LineFollower *lineFollow;

GripperServo *servo;
```

```
uint32_t tcount4 = 0;
uint32_t tcount5 = 0;

uint32_t t50ms;
uint32_t tFlop = 0;
int lastSpeed = 70;

uint32_t lastTime = 0;

/*  Remote Control */
// Buffer for serial data incoming from the bluetooth control
char fromBT[5];
int speed;
int lSpeed = 0;
uint8_t turnFactor = 10;
int rSpeed = 0;
bool servoTimersOn = false;
bool lineTimersOn = false;

// Maze mode stuff
bool mazeMode = false;
uint8_t spd = 15;
uint16_t turnLoop = 24;
uint16_t turnLoop180 = 49;
char lastTurn = 'n';
uint8_t distLimit = 20;

void setup() {
  Serial.begin(115200);
  delay(100);

  // Instantiate classes
  rightMotors = new PIDSpeedControl(3, 49, 50, 2);
  leftMotors = new PIDSpeedControl(4, 48, 50, 1);
  rightMotors->setkP(5.0);
  leftMotors->setkP(5.0);
  distSensor = new RangeFinder(7, 2);
  attachInterrupt(digitalPinToInterrupt(2), randISR, CHANGE);
  //leftMotors->setSpeed(70); // RPM
  //rightMotors->setSpeed(70); // RPM
  lineFollow = new LineFollower(leftMotors, rightMotors, 50, distSensor);
  lineFollow->setSpeed(0);
```

```
  lineFollow->disable();
  t50ms = millis();
  speed = 20;

  // Instantiate servo
  // This is annoying, it appear to use timers 4 & 5 (among others) - I went
  // and looked at the source code.
  servo = new GripperServo(9);
  servo->Close();
  servoTimersOn = true;

  //leftMotors->setSpeed(20);
  //lineFollow->togglePrint();
  //leftMotors->togglePrint();
  //rightMotors->togglePrint();

  // Setup Remote Control
  Serial2.begin(115200);
  //setupLineTimers();

  interrupts();
}

void loop() {
  //leftMotors->update();

  if (Serial2.available()) {
    char readCommand;
    readCommand = Serial2.read();
    Serial.print("From BT: ");
    Serial.println(readCommand);

    if (lineTimersOn) {
      turnFactor = 30;
    } else {
      turnFactor = 10;
    }

    // Iterate over the recieved command and process each command individually.
    // The right-most command is the most recent so this is good.
    if (readCommand == 'S') {
      lineFollow->setSpeed(0);
```

```
} else if (readCommand == 'F') {
  lineFollow->setSpeed(speed);
} else if (readCommand == 'B') {
  lineFollow->setSpeed(speed*-1);
} else if (readCommand == 'G') { // Right side faster (Turn Left)
  lineFollow->setLSpeed(speed);
  lineFollow->setRSpeed(speed+turnFactor);
} else if (readCommand == 'I') { // Left side faster (Turn right)
  lineFollow->setLSpeed(speed+turnFactor);
  lineFollow->setRSpeed(speed);
} else if (readCommand == 'J') { // (Turn right, Backward)
  lineFollow->setLSpeed(-1*(speed+turnFactor));
  lineFollow->setRSpeed(-1*speed);
} else if (readCommand == 'H') { // (Turn left, Backward)
  lineFollow->setLSpeed(-1*speed);
  lineFollow->setRSpeed(-1*(speed+turnFactor));
} else if (readCommand == 'L') { // (Turn Left)
  lineFollow->setLSpeed(-(turnFactor+speed/3));
  lineFollow->setRSpeed(turnFactor+speed/3);
} else if (readCommand == 'R') { // (Turn Left)
  lineFollow->setLSpeed(turnFactor+speed/3);
  lineFollow->setRSpeed(-(turnFactor+speed/3));
} else if (readCommand == 'W') { // Enable line follower
  if (lineTimersOn == false) {
    servo->detach();
    setupLineTimers();
    lineTimersOn = true;
    servoTimersOn = false;
  }
  lineFollow->enable();
} else if (readCommand == 'w') { // Disable line follower
  lineFollow->disable();
} else if (readCommand == 'U') { // Open gripper
  if (servoTimersOn == false) {
    setupServoTimers(); // attachs servo again
    servoTimersOn = true;
    lineTimersOn = false;
  }
  servo->Open();
} else if (readCommand == 'u') { // Close gripper
  servo->Close();
} else if (readCommand == 'X') { // Enable maze mode
```

```
      mazeMode = true;
    } else if (readCommand == 'x') { // Disable maze mode
      mazeMode = false;
      lineFollow->setSpeed(0);
    } else if (readCommand >= 48 && readCommand <= 57) { // Check if it's a number
      // Change speed based on slider position
      speed = (readCommand - 48) * 10;
    }
  }


  if (mazeMode) {
    float dist = distSensor->MeasureCm();

    // We want to keep a constant distance from the wall on the left.
    // The Sensor is mounted pointing left, towards the left-hand wall.
    lineFollow->setSpeed(spd);

    // We also use a line sensor (optical sensor) to check if we are running
    // headfirst into a wall. If we are, we stop, and turn right.
    if (digitalRead(37) == LOW) {
      turnRight90();
    } else if (dist > distLimit + 2) {
      // Turn towards the wall
      lineFollow->setRSpeed(spd+20);
      lineFollow->setLSpeed(spd);
    } else if (dist < distLimit - 2) {
      // Turn away from the wall
      lineFollow->setLSpeed(spd+20);
      lineFollow->setRSpeed(spd);
    }

  }

  // Update line follower every 50ms
  if (millis() - t50ms >= 50) {
    lineFollow->update();
    t50ms = millis();
    //Serial.print(tcount4);
    //Serial.print(", ");
    //Serial.println(tcount5);
```

```
    //float dist = distSensor->MeasureCm();
    //Serial.print("Dist: ");
    //Serial.print(dist);
    //Serial.println("cm");

    //Serial.println(digitalRead(37));
  }
}

void randISR() {
  distSensor->EchoEvent();
}

// Maze functions
void turnLeft180() {
  // Do a bulldozer turn for a certain loop count
  for (int c = 0; c < turnLoop180; c++) {
    lineFollow->setLSpeed(-10);
    lineFollow->setRSpeed(10);
    lineFollow->update();
    delay(50);
  }
  lineFollow->setSpeed(0);
  delay(100);
}

void turnRight90() {
  // Do a bulldozer turn for a certain loop count
  for (int c = 0; c < turnLoop; c++) {
    lineFollow->setLSpeed(10);
    lineFollow->setRSpeed(-10);
    lineFollow->update();
    delay(50);
  }
  lineFollow->setSpeed(0);
  delay(100);
}
// These next two functions are necessary because the two features use the same
// timers on the arduino.
void setupLineTimers() {
  // Turn on PID control
  leftMotors->enable();
```

```
  rightMotors->enable();

  // Turn on timers
  TCCR4A = 0b00000000; // Normal mode.
  TCCR4B = 0b11000101; // Noise cancel, High edge triggering,
  // Normal mode, divide by 1024
  TCCR5A = 0b00000000;
  TCCR5B = 0b11000101;

  // Enable interrupts
  TIMSK4 = 0b00100001; // (Input capture, overflow)
  TIMSK5 = 0b00100001; // (Input capture, overflow)
}

void setupServoTimers() {
  servo->attach();
  leftMotors->disable();
  rightMotors->disable();
}


// Timer setups
ISR(TIMER4_CAPT_vect) { // Pin 49
  tcount4 = (tcount4 & 0xFFFF0000) | TCNT4;
  leftMotors->encUp(tcount4);
}

ISR(TIMER5_CAPT_vect) { // Pin 48
  tcount5 = (tcount5 & 0xFFFF0000) | TCNT5;;
  rightMotors->encUp(tcount5);
}

ISR(TIMER4_OVF_vect) {
  // Running this vector auto-clears the overflow
  // Add to the count
  tcount4 = tcount4 + 0x10000;
}

ISR(TIMER5_OVF_vect) {
  // Running this vector auto-clears the overflow
  // Add to the count
  tcount5 = tcount5 + 0x10000;
```

```
}
```

PIDSpeedControl.cpp

```
#include "PIDSpeedControl.h"

// Recalculate the required PWM value to maintain our request speed.
void PIDSpeedControl::update() {
  // Read actual speed:
  // If our previous speed was zero, the check function will take a while to
  // return because it will try and wait for the encoder to update. To respond
  // faster in the these situations, we simply say if the last motor encoder
  // check returned 0, to just set the actual speed to 0. Then, next time
  // we actually read the encoder
  actualSpeed = motorEncoder->encRPM();

  if (enabled) {
    // PID Calculations
    int error = 0;
    if (requestedSpeed > 0 ) {
      error = requestedSpeed - actualSpeed;
    } else {
      error = -1*requestedSpeed - actualSpeed;
    }

    proportional = kP * error * loopTime/10;
    integral += kI * error;
    //derivative = kD * (pError - error)/loopTime;

    // Cap the integral
    if (integral > INTEGRAL_MAX) {
      integral = INTEGRAL_MAX;
    } else if (integral < INTEGRAL_MIN) {
      integral = INTEGRAL_MIN;
    }

    // Cap proportional
    if (proportional > PROP_MAX) {
      proportional = PROP_MAX;
    } else if (proportional < PROP_MIN) {
      proportional = PROP_MIN;
    }

    drive = integral + proportional;
```

```
    // Map drive to 0-255
    drive = (255.0*drive)/3000.0;

    if (drive > 255) {
      drive = 255;
    } else if (drive < 0) {
      drive = 0;
    }

    pError = error;
  } else {
    // No PI control

    // Map input speed to PWM. (kinda.)
    if (requestedSpeed > 0) {
      drive = requestedSpeed*5;
    } else {
      drive = -1*requestedSpeed*5;
    }
    if (drive > 200) {
      drive = 200;
    }
  }

  if (printData) {
    Serial.print(requestedSpeed);
    Serial.print(",");
    Serial.print(actualSpeed);
    Serial.print(",");
    Serial.println(drive);
  }
  /*
  Serial.print("Error: ");
  Serial.println(error);

  Serial.print("Drive: ");
  Serial.println(drive);

  Serial.print("Integral: ");
  Serial.println(integral);
```

```
  Serial.print("Proportional: ");
  Serial.println(proportional);

  Serial.print("Actual: ");
  Serial.println(actualSpeed);*/

  // Command the motors
  if (requestedSpeed == 0 ) {
    motor->roll();
    motor->setPwm(0);
    motorSlave->roll();
    motorSlave->setPwm(0);
  } else if (requestedSpeed > 0){
    motor->forward();
    motor->setPwm(drive);
    motorSlave->forward();
    motorSlave->setPwm(drive);
  } else if (requestedSpeed < 0){
    motor->reverse();
    motor->setPwm(drive);
    motorSlave->reverse();
    motorSlave->setPwm(drive);
  }
}


int PIDSpeedControl::getRPM() {
  return motorEncoder->encRPM();
}

uint16_t PIDSpeedControl::getPwm() {
  return drive;
}

void PIDSpeedControl::setSpeed(int rpm) {
    requestedSpeed = rpm;
}

void PIDSpeedControl::encUp(uint32_t n) {
  motorEncoder->up(n);
}
```

```
void PIDSpeedControl::togglePrint() {
    Serial.println("Requested, Actual, PWM");
    printData = !printData;
}

void PIDSpeedControl::enable() {
  enabled = true;
}

void PIDSpeedControl::disable() {
  enabled = false;
}

void PIDSpeedControl::setkP(float newval) {
  kP = newval;
}



// Class constructor
PIDSpeedControl::PIDSpeedControl(int motorN, int encPin, int loopPeriod, int motor2N) {
    // Instantiate Motor & Encoder
    motor = new ChassisMotor(motorN);
    motorSlave = new ChassisMotor(motor2N);
    motorEncoder = new EncoderClass(encPin);

    kP = DEFAULT_KP;
    kI = DEFAULT_KI;
    kD = DEFAULT_KD;

    integral = 0;
    proportional = 0;
    derivative = 0;
    loopTime = loopPeriod;

    printData = false;
    enabled = true;

}
```

PIDSpeedControl.h

```
#include "motors.h"
#include "encoder.h"
```

```
#include "Arduino.h"

#ifndef PIDCONTROL_H
#define PIDCONTROL_H
// PID speed control
#define INTEGRAL_MAX   3000
#define INTEGRAL_MIN  -3000

#define PROP_MAX 3000
#define PROP_MIN -3000


#define DEFAULT_KP 3
#define DEFAULT_KI 2
#define DEFAULT_KD 0

// Basic form of PID loop
/*
Err = Sp – PV

P = kP x Err

It = It + (Err x kI x dt)

D = kD x (pErr – Err) / dt

pErr = Err

Output = P + It + D
*/

// PID Speed control
// Calculates the proper PWM value for a motor based on a commanded speed and an encoder input.
class PIDSpeedControl {
  public:
    // Class constructor
    PIDSpeedControl(int motorN, int encPin, int loopPeriod, int motor2N);

    // Set desired speed
    void setSpeed(int rpm);
    // Update/Recalculate
    void update();
```

```
  // Return current error
  // Set integral coefficient
  void setkI();
  // Set proportional coefficient
  void setkP(float newval);
  // Set derivative coefficient
  void setkD();
  // Encoder edge
  void encUp(uint32_t n);
  // Return encoder reading
  int getRPM();
  // Return current PWM
  uint16_t getPwm();
  // Toggle Printing
  void togglePrint();

  // Enables and disables speed PI-ness (motors still operate)
  void enable();
  void disable();

private:
  ChassisMotor *motor;
  ChassisMotor *motorSlave;
  EncoderClass *motorEncoder;
  uint8_t encPin;

  // Speeds are in RPM
  int actualSpeed;
  int requestedSpeed;

  // PID control variables
  int pError;   // Previous Error
  double kP;
  double kI;
  double kD;
  int integral;
  int proportional;
  int derivative;
  int loopTime;
  int drive;

  // Other
```

```
  bool printData;
  bool enabled;
};

// Line follower control
// Steers the robot based on the line follower sensors and by commanding
// speeds with the PID speed control
#endif PIDCONTROL_H
```

lineFollow.cpp

```
#include "lineFollow.h"


LineSensor *LineFollower::ls1 = new LineSensor(19);
LineSensor *LineFollower::ls2 = new LineSensor(18);

bool LineFollower::side1 = false;
bool LineFollower::side2 = false;

volatile uint8_t LineFollower::sideLastSeen = 0;

void LineFollower::interruptLineSensor() {
 side1 = !ls1->OnBlack();
 side2 = !ls2->OnBlack();
 if (side1 && !side2) {
  sideLastSeen = 1;
 } else if (side2 && !side1) {
  sideLastSeen = 2;
 }
}

// Update control
void LineFollower::update() {
 // Check if we are on the line,
 if (enabled) {
  bool onLine = digitalRead(37);
  if (blockDetect->MeasureCm() < 10) {
    speedSide1 = 0;
    speedSide2 = 0;
  } else if (onLine) {
   // Go normal speed
   //sideLastSeen = 0;
   resetSides();
```

```
    /*} else if (side1 != side2) {
      // Go towards the correct side
      // Record the side we saw last
      if (side1) {
        //sideLastSeen = 1;
        speedUpSide2(); // Go towards side 1
        //speedSide1 += 10;
      } else {
        //sideLastSeen = 2;
        speedUpSide1(); // Go towards side 2
        //speedSide2 += 10;
      }*/
    } else { // Not on the line
      if (sideLastSeen == 1) {
        speedUpSide2(); // Go towards side 1
      } else if (sideLastSeen == 2) {
        speedUpSide1(); // Go towards side 2
      } else if (sideLastSeen == 0) {
        // Lost the line, stop the robot
        //speedSide1 = 0;
        //speedSide2 = 0;
      }
    }
  }

  motor1->setSpeed(speedSide1);
  motor2->setSpeed(speedSide2);

  motor1->update();
  motor2->update();

  if (debugPrint) {
    Serial.print(speedSide1);
    Serial.print(", ");
    Serial.println(speedSide2);
  }
}

// Update the requested speed
void LineFollower::setSpeed(int newSpeed) {
  requestedSpeed = newSpeed;
  speedSide1 = newSpeed;
```

```
  speedSide2 = newSpeed;
}

// Increase speed on motors side 1
void LineFollower::speedUpSide1() {
  speedSide1 += SPEED_UP;
  if (speedSide1 > MAX_RPM) {
    speedSide1 = MAX_RPM;
  }

  speedSide2 -= SPEED_DOWN;
  if (speedSide2 < MIN_RPM) {
    speedSide2 = MIN_RPM;
  }
}

// Increase speed on motors side 2
void LineFollower::speedUpSide2() {
  speedSide2 += SPEED_UP;
  if (speedSide2 > MAX_RPM) {
    speedSide2 = MAX_RPM;
  }
  speedSide1 -= SPEED_DOWN;
  if (speedSide1 < MIN_RPM) {
    speedSide1 = MIN_RPM;
  }
}

void LineFollower::setLSpeed(int newSpeed) {
  speedSide1 = newSpeed;
}

void LineFollower::setRSpeed(int newSpeed) {
  speedSide2 = newSpeed;
}

void LineFollower::resetSides() {
  speedSide1 = requestedSpeed;
  speedSide2 = requestedSpeed;
}

void LineFollower::enable() {
```

```
  enabled = true;
}

void LineFollower::disable() {
  enabled = false;
}

void LineFollower::togglePrint() {
  debugPrint = !debugPrint;
  if (debugPrint) {
    Serial.println("SideSpeed1, SideSpeed2");
  }
}

// Class Constructor
LineFollower::LineFollower( PIDSpeedControl *control1, PIDSpeedControl *control2,
                            int loopPeriod, RangeFinder *detector) {

  // Tie in our motors
  motor1 = control1;
  motor2 = control2;

  // setup distance sensor
  blockDetect = detector;

    // Setup line sensors
  attachInterrupt(digitalPinToInterrupt(18), interruptLineSensor, CHANGE);
  attachInterrupt(digitalPinToInterrupt(19), interruptLineSensor, CHANGE);
  pinMode(37, INPUT); // On-the-line sensor

  loopTime = loopPeriod;

  // Control variables
  enabled = false;

  resetSides();
  debugPrint = false;
}
```

lineFollow.h

```
#include "Arduino.h"
#include "PIDSpeedControl.h"
#include "lineSensor.h"
```

```
#include "rangeFinder.h"
// PID speed control
#define INTEGRAL_MAX   6000
#define INTEGRAL_MIN  -6000

#define SPEED_UP 5
#define SPEED_DOWN 6

#define MAX_RPM 40
#define MIN_RPM -1

class LineFollower {
public:
  // Constructor
  LineFollower( PIDSpeedControl *control1, PIDSpeedControl *control2, int loopPeriod, RangeFinder
*blockDetect);

  // Update Function
  void update();

  // Set requested Speed
  void setSpeed(int newSpeed);
  void setLSpeed(int newSpeed);
  void setRSpeed(int newSpeed);

  // Enable/Disable
  void enable();
  void disable();

  // Debugging
  void togglePrint();

private:
  // Static method called via an interrupt ISR.
  static void interruptLineSensor();
  // Other line sensor variables
  static bool side1;
  static bool side2;

  // Correction methods
  void speedUpSide1();
  void speedUpSide2();
```

```
  void resetSides();

  // Ms
  uint32_t stepTime;

  // Class pointers
  PIDSpeedControl *motor1;
  PIDSpeedControl *motor2;
  static LineSensor *ls1;
  static LineSensor *ls2;
  RangeFinder *blockDetect;

  // Speeds are in RPM
  int requestedSpeed;
  int turnSpeed;
  double speedSide1;
  double speedSide2;

  // PID control variables
  int loopTime;
  int timeError;

  // State/Status variables
  bool enabled;
  static volatile uint8_t sideLastSeen;

  // Debugging
  bool debugPrint;

};
```

encoder.cpp

```
#include "encoder.h"

// Upon measuring the minimum pulse width, it seems to be ~10ms.
// The slowest timer speed is about 7.8kHz, or about 0.128ms.
// The timer rolls over once about every 8.38 seconds.

// I've decided to keep track of each timer in a 32 bit unsigned
// integer. This gives us 549755.8 seconds before overflow, in
// other words, 152.7 hours or 6.3 days :)
// (so essentially I'm just ignoring rollover...)
```

```
// What to do about zero speed or very little speed:
// We report that the speed is zero if the "checker" checks
// the speed a certain number of times and finds no new edges.
// (Check "registerEdge" N number of times, report 0)

// log the rising edge
void EncoderClass::up(uint32_t n) {
  timeElapsed = (n - lastCount)*0.128;
  addSample(timeElapsed);

  lastCount = n;

  registerEdge = true;

}

// Reports period in milliseconds, averaged
uint16_t EncoderClass::encPeriodAvg() {
  // Sum the array and return the average
  uint32_t sum;
  for (int n = 0; n < AVG_SIZE; n++) {
    sum = sum + periodAveraged[n];
  }

  sum = sum/AVG_SIZE;

  // Check if we even have a new edge
  if (registerEdge) {
    registerEdge = false;
    deadCount=0;
  } else { // No new samples!
    deadCount++;
    if (deadCount >= ALLOWED_DEAD_COUNT) {
      sum = 0;
    }
  }


  return sum;
}

uint16_t EncoderClass::encRPM(){
```

```
  // RPM, 1 rotation = 360 degrees, 1 "up" edge means
  // we went 1 slot, which is 18 degrees, 360/18 = 20

  // Calculate the RPM
  float temp = (50/float(encPeriodAvg()))*60;
  return temp;
}


// Adds a sample to the average
void EncoderClass::addSample(uint16_t millisSample) {
  // Shift previous samples to the right
  uint16_t copyArray[AVG_SIZE];  // make copy
  memcpy(copyArray, periodAveraged, AVG_SIZE * sizeof(uint16_t));
  for (int i = 0; i < AVG_SIZE-1; i++) {
    periodAveraged[i+1] = copyArray[i];
  }

  periodAveraged[0] = millisSample;
}


bool EncoderClass::pinState() {
  state = digitalRead(pin);
  return state;
}

// Class constructor
EncoderClass::EncoderClass(int p) {
  pin = p;
  deadCount = 0;

  // Make our pin an input
  pinMode(pin, INPUT);
}
```

encoder.h

```
/* File: encoder.h
 * Author: Zachary Whitlock
 * Description:
 * Header file for the encoder class object
 */
```

```
#include "Arduino.h"

#ifndef ENCODER_H
#define ENCODER_H

#define ALLOWED_DEAD_COUNT 2
#define AVG_SIZE 3

// Alright, so we need to know the time since last

// Encoder class
class EncoderClass {
  public:
    int pin;
    uint32_t count;
    bool pinState();

    // Returns the current RPM based on the averaged period
    uint16_t encRPM();

    // Fires upon a rising-edge
    void up(uint32_t n);

    // Reports the current period of the encoder
    uint16_t encPeriodAvg();

    // Class constructor
    EncoderClass(int p);

  private:
    // Adds a sample to the average
    void addSample(uint16_t millisSample);

    // States
    bool lastState;
    bool state;

    // Keeps track of how many times we checked without
    // finding new data.
    uint8_t deadCount;

    // Counts and times
```

```
    uint32_t lastCount;
    uint32_t countElapsed;
    bool registerEdge;

    // Average of 10 samples
    // TODO: make weighted
    uint16_t periodAveraged[AVG_SIZE];

    // Time between changes
    uint16_t timeElapsed;
};

#endif
```

motors.cpp

```
#include "motors.h"

// Zur Steuerung von Motoren
// (Managing is captilized?)
// Translates more roughly into "controlling"

// Okay so if "Zur" is 'for', and is a conjugate form of "zu" I believe.
// Controlling encoders would be like Zur Steuerung von Encoder, "For control of encoders"
// Von is 'of', which, as a preposition, still isn't conjugated?
// Prepositions are a class of words used to express special or temporal relations, like how and when.
// Germans have 4 different cases that can modify prepositions, nominative, accusative, dative, and
gentive.
// Nominative tells us who or what is doing something, in this case I don't really know since it's
basically
// just a title, "Zur",
// In accusative, it's very similar to the nominative in the sense that you're talking about something
doing something,
// but in an accusative case, you're refering to the thing that is being affected by the action.
// So if your noun is being affected by the verb, you use the accusative form.

// Okay but in short, the cases (nominative, accusative, dative, and gentive, are important)

// Die ("dee") Motoren Code soll (should/is supposed to) anhangen (attach/pin to) die Adafruit Motoren
Bibliothek (library)

// These are "static", only declared once :)
bool ChassisMotor::instanciated = false;
Adafruit_MotorShield ChassisMotor::adaShield = Adafruit_MotorShield();
```

```
void ChassisMotor::roll() {
  adaMotor->run(RELEASE);
}

void ChassisMotor::forward() {
  adaMotor->run(FORWARD);
}

void ChassisMotor::reverse() {
  adaMotor->run(BACKWARD);
}


void ChassisMotor::setPwm(int spd) {
  adaMotor->setSpeed(spd);
}

// Constructor for our thing
ChassisMotor::ChassisMotor(int motor_N) {
  // Instantiate adaShield if not instantiated

  // Setup new motor
  adaMotor = adaShield.getMotor(motor_N);

  if (instanciated == false) {
    Serial.println("Instanciating shield");
    adaShield.begin();

    instanciated = true;
  }

}
```

motors.h

```
#include "Arduino.h"
#include <Adafruit_MotorShield.h>

#ifndef CHASSISMOTOR_H
#define CHASSISMOTOR_H
class ChassisMotor
{
```

```
  public:
    ChassisMotor(int motor_N);
    void setPwm(int spd);
    void forward();
    void reverse();
    void roll();

  private:
    static bool instanciated;
    Adafruit_DCMotor *adaMotor;
    static Adafruit_MotorShield adaShield; // Instantiate ONCE

};

#endif
```

rangeFinder.cpp

```
#include "rangeFinder.h"

void RangeFinder::EchoEvent() {
  state++;
  if (state == 1) {
    usDelay = micros();
  } else if (state == 2) {
    usDelay = micros() - usDelay;
    state = 3;
  }
}

// Send a trigger
uint32_t RangeFinder::Measure() {
   long timeout = millis();

  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(50);
  digitalWrite(trigPin, LOW);
  digitalWrite(echoPin, LOW);
  state = 0;
  usDelay = 0;

  while (state != 3) {
```

```
    if (millis() - timeout >= 2000) {
      usDelay = 0;
      break;
    }
  }

  return usDelay;
}

// Returns measurement in inches
float RangeFinder::MeasureInch() {
  long result = Measure();
  return result / 74 / 2;
}

// Returns measurement in cm
float RangeFinder::MeasureCm() {
  long result = Measure();
  return result * 0.034 / 2;
}

RangeFinder::RangeFinder(int tPin, int ePin) {
  trigPin = tPin;
  echoPin = ePin;

  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}
```

rangeFinder.h

```
#include "Arduino.h"

#ifndef RANGEFINDER_H
#define RANGEFINDER_H

class RangeFinder
{
  public:
    // Constructor
    RangeFinder(int trigPin, int echoPin);

    // Call to send pulse
    // Returns time
```

```
    uint32_t Measure();
    // Returns inches
    float MeasureInch();
    // Returns cm
    float MeasureCm();

    // Gets called by interrupt
    void EchoEvent();

  private:
    // Private variables
    int echoPin;
    int trigPin;
    volatile int state;
    volatile unsigned long usDelay;
};

#endif
```

gripper.cpp

```
#include "gripper.h"


void GripperServo::Open() {
  internalServo.write(OPEN_ANGLE);
}

void GripperServo::Close() {
  internalServo.write(CLAMP_ANGLE);
}

void GripperServo::HalfOpen() {
  internalServo.write((OPEN_ANGLE - CLAMP_ANGLE)/2 + CLAMP_ANGLE);
}

void GripperServo::SetAngle(int angle) {
  internalServo.write(angle);
}

void GripperServo::detach() {
  internalServo.detach();

}
```

```
void GripperServo::attach() {
  //internalServo = Servo();
  internalServo.attach(myPin);
  while(!internalServo.attached());
}

GripperServo::GripperServo(int pin) {
  myPin = pin;
  internalServo = Servo();
  internalServo.attach(myPin);
  while(!internalServo.attached());
}
```

gripper.h

```
#include <Servo.h>

#define CLAMP_ANGLE 72
#define OPEN_ANGLE 145

class GripperServo
{
  public:
    // Constructor
    GripperServo(int pin);
    void Open();
    void Close();
    void HalfOpen();
    void SetAngle(int angle);
    void attach();
    void detach();

  private:
    int myPin;
    Servo internalServo;

};
```

lineSensor.cpp

```
#include "Arduino.h"
#include "lineSensor.h"


bool LineSensor::OnBlack() {
```

```
  if (digitalRead(inputPin)) {
    return true;
  }

  return false;
}

LineSensor::LineSensor(int pin) {
  inputPin = pin;

  // Attach to pin
  pinMode(pin, INPUT);
}
```

lineSensor.h

```
#ifndef LINESENSOR_H
#define LINESENSOR_H

class LineSensor
{
  public:
    bool OnBlack();
    // Constructor
    LineSensor(int pin);

  private:
    // Variables
    int inputPin;
    LineSensor *self;

};

#endif
```