POE - Lab 3

Ben Kahle and Ry Horsey

September 2014

1 Introduction

In this lab we utilized an IR sensor with an encoder and a brushed DC motor to create a controlled DC motor. The first part of the lab focuses on understanding the IR sensor and calibrating its output to produce accurate measurements of the movement of the encoder. Once the calibration process was completed and tested, we built a simple PI control loop allowing us to set desired positions and sweep across ranges sinusoidally. We then tuned our PI controller in order to decrease our steady state error and increase our noise rejection.

2 DC Motor Modeling, Circuit Diagram, and Mounting

We began by measuring the basic properties of our motor. Although this process does not come close to producing the values necessary for estimating the transfer function of the motor, it does allow some insight to the motor. We measured the resistance of the windings to be 5.8 Ω, as measured by a multimeter. The data-sheet stated the operating voltage of the motor as 3-12 V DC, and the maximum speed of the motor as 25-100 RPM. We further found the stall current of the motor to be 230 mA.

The circuit diagram provided by the teaching team in the lab report can be seen in Figure 1. It was implemented on a mini-breadboard as seen in Figure 2. The DC motor was connected directly to the DC voltage outputs of the Adafruit Arduino Motor Shield.

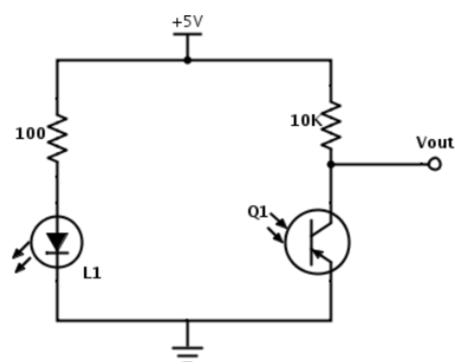

Figure 1: This figure shows the circuit diagram provided by the teaching team for the recommended circuit diagram for the IR sensor.

For our motor mount, we used a simple wood design with a base plate and cut-out for the encoder to create a easily modifiable mount while providing strong support for the motor. Our encoder sat very close to our IR sensor, as we found that the IR sensor had difficultly measuring the encoder if place further than 5 mm in distance from the sensor. An additional feature of our mount is that it provided passive noise rejection by limiting the impact of signal noise from the surrounding lights by partially enclosing the IR sensor.

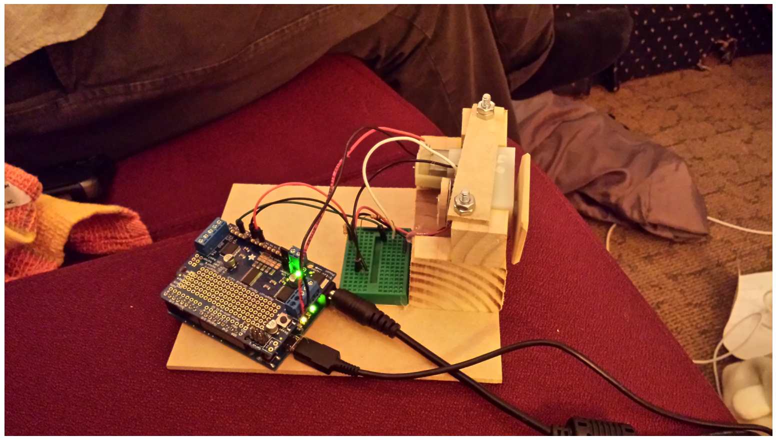

Figure 2: The stand we constructed to hold the motor and motor encoder in place. The motor encoder is composed of the wheel on the left side of the motor and the IR sensor held between the motor and the wheel. The inside of the wheel has a radial encoder pattern of 40 black and white bands attached. The IR sensor circuitry is connected on the green mini-breadboard and the motor is connected directly to the Arduino motor controller output.

3 The IR Encoder Calibration

To calibrate the IR sensor, we ran the motor slowly while logging the sensor values. As seen in Figure 3, while the encoder markers did not provide consistent readings, a cutoff of 70 allowed us to differentiate between the white bands and the black bands. We used this information to count when the encoder switched from "high" to "low". Because the encoder pattern we used had 40 bands, each transition marked a 9°rotation of the motor.

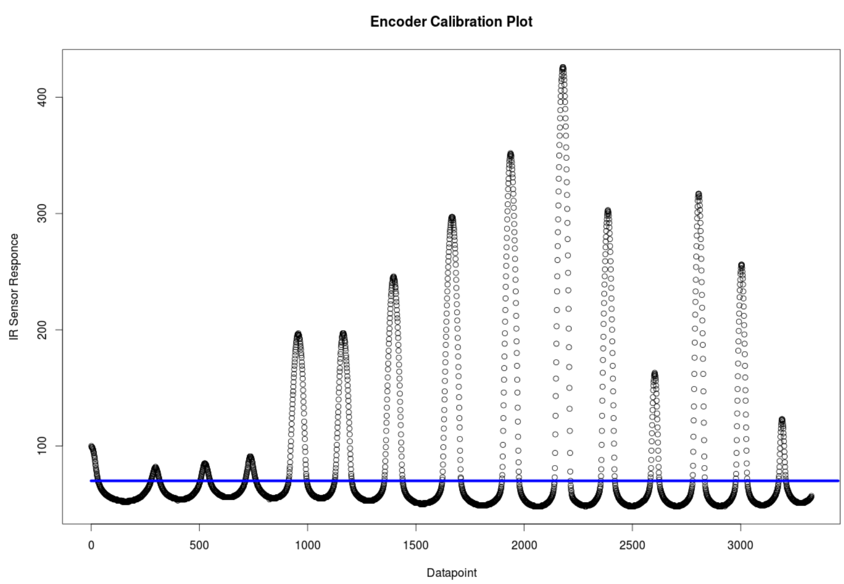

Figure 3: A plot showing the IR sensor output over time while the motor spins.

4 Step Response

We utilized a PI controller for the step response functionality we built into our code. Although a D controller would have allowed us to significantly increase our proportional gain in a stable manner, to do so in a discrete system introduces additional noise which would have been difficult to correct for without a far greater understanding of digital signal processing and digital controls. Figure 4 shows the response of the system to a 90° Heaviside step function. This graph nicely demonstrates several issues we faced in this lab.

One interesting feature to note in Figure 4 is the increase in the green line over time as the encoder fails to update. This is the integral controller attempting to compensate for the granularity in the encoder's measurement. As can be seen around $t = .9$ and $t = 1.4$, this effect can result in accelerated responses of the measured output, that is to say the encoder. This follows prescribed behavior.

Another interesting bug to note in Figure 4 is the relative linearity of the blue line. Although it does have a variable rate of change, it tends to fit a linear model well. This is representative of the proportional gain of the control loop being set low. The reason for setting this gain low is that should it be set much higher the encoder can begin to skip. This is a distinct limitation of the physical design implemented for this project. By placing the IR sensor so close to the motor shaft, the width of the radial encoder bands were greatly limited, leading to limitations in the sensor's granularity, i.e. field of focus. Although our code's sample rate could be increased, we were unable to retrieve more information from this increased sample rate. As a result, if we caused the encoder to rotate to quickly, which is to say as quickly as a typical controls stratagem for PI controls would indicate, we would not be able to accurately read the encoder.

Despite the shortcomings mentioned above, it is worthwhile to emphasize the power of PI controls in simple applications. While a transfer function was never developed from a fully characterized motor, the proportional gain constant of the controller was set lower that it should have been, and the encoder was undersized, we were able to achieve satisfactory behavior with relatively little hardship. This emphasizes that, although PI and PID controls may be looked down on in the controls community for their inability to address the underlying issues of phase margin and disturbance rejection, they can serve a highly valuable purpose in general use.

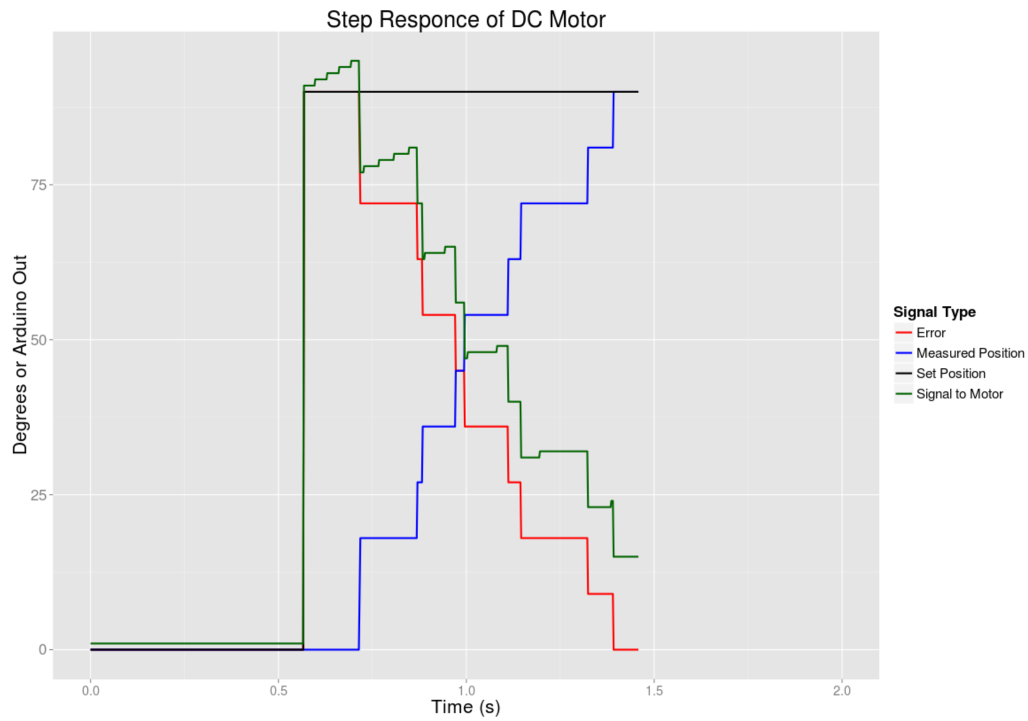

Figure 4: A plot demonstrating the step response of the PI controller to a set position of 90 degrees. The green line shows the motor control signal. The large decreases are caused by the proportional term decreasing as the motor passes encoder marks. The small increases in the green line are a result of the integral term slowly adding to the control signal.

5 Sinusoidal Response

The issues surrounding the tracing of a sinusoidal input closely resemble those issues identified in the step response, as expected. The primary issue seen here is that a low gain, or low proportional constant, has lead to unnecessary lag time in the system. This could be fixed in part by a simple increase in the proportional gain constant, however the issues with such a decision have been outlined above. Otherwise, the general behavior of the PI controller appears very reasonable and reflects the expected behavior of such a system in a tracking application. Once again, significant improvements could be made should the field of view of the IR sensor be decreased.

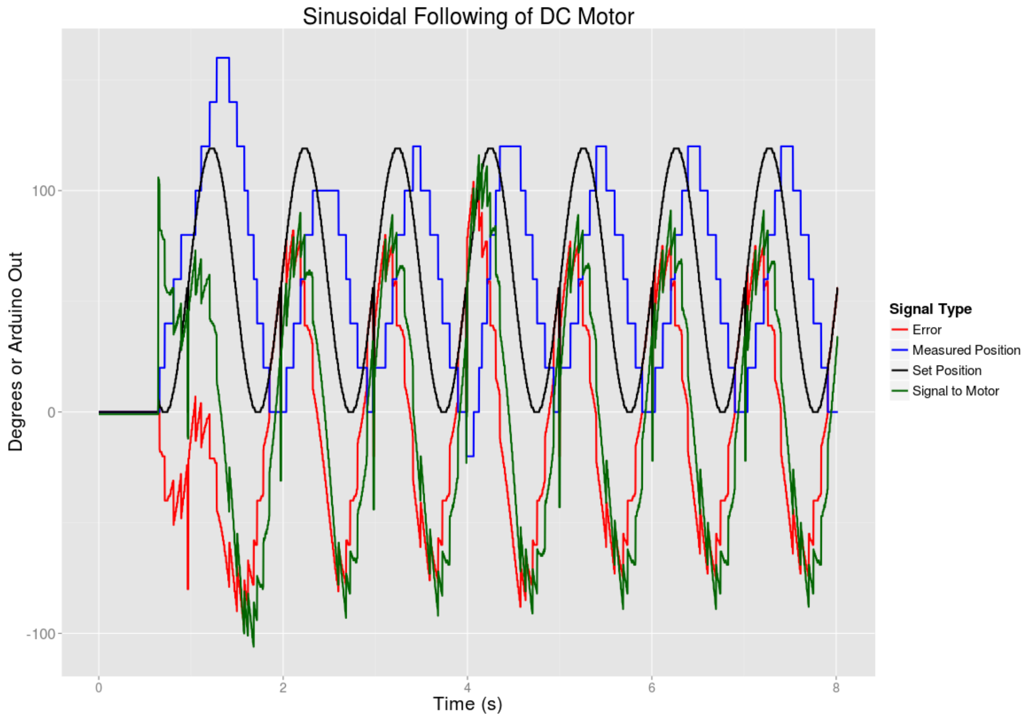

Figure 5: A plot showing the PI controller's response to a constantly moving target: a sinusoidal wave. After the initial hunting caused by the beginning of change in target position, the controller caught up with the target and followed it within a margin of roughly 1/8th of a second.

6 Conclusion

In this lab we implemented a simple digital PI control system for a low-cost brushed DC motor. We saw that through implementing a PI loop with an optical encoder it was relatively easy to command the motor to any arbitrary position. In this lab we were able to avoid the issues of windup and drift, both of which deserve attention in broader circumstances, however were still able to demonstrate the usefulness of PI controls in application. Additionally, this lab taught appreciation for motors with encoders built into them, as well as emphasizing the necessity of understanding the use cases of sensors as seen throgh the difficulties caused by the physical placement our encoder relative to the motor shaft.

7 Appendix

7.1 Arduino Code

```
#include <Wire.h>
#include <Adafruit_MotorShield.h>
#include "utility/Adafruit_PWMServoDriver.h"

int sensorPin = A0;

const int encoderLightCutoff = 70; // Value of encoder between "high" and "low"
const int encoderRes = 9; // Degress per encoder band
const int loopDelay = .2; // Time between controller actions
const float sweepFreq = 1; // Sweeps per second (sinusoid following)

int sensorValue = 0;

```

```
int motorPos = 0;
int motorDir = 1; // 1 = forward, -1 = reverse
int encoderPos = 0; // 0 = light (high), 1 = dark(low)

int commandMode = 0; // 0 = step, 1 = sweep
int command = 0; // command from serial
int positionGoal = 0; // target degree parsed from serial command
int error = 0;
int motorCommand = 0; // Controller signal

long errorSum = 0;
float kp;
float ki;

int loopCounter = 0;
float loopsPerSecond = 1000/loopDelay;
float loopsPerSweepStep = loopsPerSecond/120; // # loops between sinusoid periods

// Discretized sinusoid
static int sinWave[120] = {
  0x7ff, 0x86a, 0x8d5, 0x93f, 0x9a9, 0xa11, 0xa78, 0xadd, 0xb40, 0xba1,
  0xbff, 0xc5a, 0xcb2, 0xd08, 0xd59, 0xda7, 0xdf1, 0xe36, 0xe77, 0xeb4,
  0xeec, 0xf1f, 0xf4d, 0xf77, 0xf9a, 0xfb9, 0xfd2, 0xfe5, 0xff3, 0xffc,
  0xfff, 0xffc, 0xff3, 0xfe5, 0xfd2, 0xfb9, 0xf9a, 0xf77, 0xf4d, 0xf1f,
  0xeec, 0xeb4, 0xe77, 0xe36, 0xdf1, 0xda7, 0xd59, 0xd08, 0xcb2, 0xc5a,
  0xbff, 0xba1, 0xb40, 0xadd, 0xa78, 0xa11, 0x9a9, 0x93f, 0x8d5, 0x86a,
  0x7ff, 0x794, 0x729, 0x6bf, 0x655, 0x5ed, 0x586, 0x521, 0x4be, 0x45d,
  0x3ff, 0x3a4, 0x34c, 0x2f6, 0x2a5, 0x257, 0x20d, 0x1c8, 0x187, 0x14a,
  0x112, 0xdf, 0xb1, 0x87, 0x64, 0x45, 0x2c, 0x19, 0xb, 0x2,
  0x0, 0x2, 0xb, 0x19, 0x2c, 0x45, 0x64, 0x87, 0xb1, 0xdf,
  0x112, 0x14a, 0x187, 0x1c8, 0x20d, 0x257, 0x2a5, 0x2f6, 0x34c, 0x3a4,
  0x3ff, 0x45d, 0x4be, 0x521, 0x586, 0x5ed, 0x655, 0x6bf, 0x729, 0x794
};

Adafruit_MotorShield AFMS = Adafruit_MotorShield();
Adafruit_DCMotor *motor = AFMS.getMotor(1);

void setup() {
  Serial.begin(9600);
  AFMS.begin();  // create with the default frequency 1.6KHz
}

/*
* Wraps the motor API to allow specifying speed and direction through positive
* and negative integers
*/
void powerMotor(int speed) {
  if (speed < 0) {
    motorDir = -1;
    motor->run(BACKWARD);
  } else {
    motorDir = 1;
    motor->run(FORWARD);
  }
  motor->setSpeed(abs(speed));
}

/*
* Given a proportional control constant (kp) and integral control constant (ki)
* calculates the controller signal to the moter based on current motor position
* and target position
*/
void setMotor(float kp, float ki) {
  error = positionGoal - motorPos;
  errorSum += error;
  int pTerm = error+kp;
  int iTerm = errorSum*ki;
  motorCommand = min(pTerm + iTerm, 255);
```

```
  powerMotor(motorCommand);
}

/*
* Set control constants and call for a controller signal based on a step response
*/
void step() {
  kp = 1.2; // Proportional control constant for step response
  ki = 0.0007; // Integral control constant for step response
  setMotor(kp, ki);
}

/*
* Set the control constants and call for a controller signal based on a
* sinusoidal sweep to a certain target position.
*/
void sweep(int sweepPos, int command) {
  kp = 165; // Proportional control constant for sweep response
  ki = 0.008; // Integral control constant for sweep response
  int waveIndex = sweepPos/loopsPerSweepStep; // index into sinusoid period
  float fractionalPos = (float)sinWave[waveIndex]/4096; // sine value at index scaled to
      fraction of 1
  positionGoal = fractionalPos*command;
  setMotor(kp, ki);
}

/*
* recalculate the motor's position based on a change in encoder value
*/
void adjustPosition(int encPos) {
  motorPos += encoderRes*motorDir; //incr. or decr. pos based on direction
  encoderPos = encPos;
}

void printInfo() {
  Serial.print(positionGoal);
  Serial.print(",");
  Serial.print(motorPos);
  Serial.print(",");
  Serial.print(error);
  Serial.print(",");
  Serial.println(motorCommand);
}

void loop() {
  sensorValue = analogRead(sensorPin);
  // Check for a change in motor position based on encoder value and last state
  if (sensorValue < encoderLightCutoff && encoderPos == 1) {
    adjustPosition(0);
  } else if (sensorValue > encoderLightCutoff && encoderPos == 0) {
    adjustPosition(1);
  }
  // Check for a command over serial port
  if (Serial.available() > 0) {
    command = Serial.parseInt();
    errorSum = 0; // Reset error sum when a new command is given
    if (command >= 0) { // Positive command = Go to <commmand> degrees
      commandMode = 0; // Step response
      positionGoal = command;
    } else { // Negative command = Sweep from 0 to <command> degrees
      commandMode = 1; // Sweep response
      command = -command; // Flip command to positive degrees
    }
  }
  if (!commandMode) {
    step();
  } else {
    sweep(loopCounter, command);
```

```
  }
  printInfo();
  // Increment loop counter and reset to zero after each sinusoid sweep
  loopCounter = ++loopCounter%((int)(121*loopsPerSweepStep));
  delay(loopDelay); // Allow motor to act before updating control
}
```

7.2 Processing and Plotting Code

```

#Post processing script for POE Fall 2013 Lab 3
#Initialize the required packages
require(ggplot2)
require(grid)

#Load in data for the sweep function and reformat it for ggplot2
setwd("~/gitRepos/fall2014/poe/labThree/poe_motorcontroller")
sweep_df = read.csv(file = 'sweep.csv', header = FALSE)
colnames(sweep_df) = c('set_pos', 'meas_pos', 'error', 'signal')
sweep_time = data.frame()
sweep_time[1:nrow(sweep_df),"time"] = seq(0, 0.002*(nrow(sweep_df)-1), by = 0.002)
ploting_df = data.frame()
for(j in 1:ncol(sweep_df)){
  str_val = colnames(sweep_df)[j]
  init_ind = (j-1)*i
  for(i in 1:nrow(sweep_df)){
    ploting_df[(i+init_ind),'value'] = sweep_df[i,j]
    ploting_df[(i+init_ind),'time'] = sweep_time[i,'time']
    ploting_df[(i+init_ind),'type'] = colnames(sweep_df)[j]
  }
}
#Plot the sweep data
ggplot(data = ploting_df, aes(x = time, y = value, colour = factor(type)))+
  geom_line(size = rel(1))+
  labs(x = "Time␣(s)", y = "Degrees␣or␣Arduino␣Out", title = "Sinusoidal␣Following␣of␣DC␣Motor")+
  theme(legend.text=element_text(size=rel(1.25)),axis.title.x = element_text(size=rel(1)), axis.title.y = element_text(size=rel(1)), title = element_text(size=rel(1.75)), axis.text.x = element_text(size=rel(1.5)), axis.text.y = element_text(size=rel(1.75)))+
  scale_color_manual(name = 'Signal␣Type', values = c("Red", "Blue", "Black", "DarkGreen"), labels = c('Error', 'Measured␣Position', 'Set␣Position', 'Signal␣to␣Motor'))

#Load in and reformat for ggplot2 the step data
step_df = read.csv(file = 'step90.csv', header = FALSE)
step_df = step_df[1:1000,]
colnames(step_df) = c('set_pos', 'meas_pos', 'error', 'signal')
step_time = data.frame()
step_time[1:nrow(step_df),"time"] = seq(0, 0.002*(nrow(step_df)-1), by = 0.002)
ploting_df = data.frame()
for(j in 1:ncol(step_df)){
  str_val = colnames(step_df)[j]
  init_ind = (j-1)*i
  for(i in 1:nrow(step_df)){
    ploting_df[(i+init_ind),'value'] = step_df[i,j]
    ploting_df[(i+init_ind),'time'] = step_time[i,'time']
    ploting_df[(i+init_ind),'type'] = colnames(step_df)[j]
  }
}
#Plot the step function
ggplot(data = ploting_df, aes(x = time, y = value, colour = factor(type)))+
  geom_line(size = rel(1))+
  labs(x = "Time␣(s)", y = "Degrees␣or␣Arduino␣Out", title = "Step␣Responce␣of␣DC␣Motor")+
  theme(legend.text=element_text(size=rel(1.25)),axis.title.x = element_text(size=rel(1)), axis.title.y = element_text(size=rel(1)), title = element_text(size=rel(1.75)), axis.text.x = element_text(size=rel(1.5)), axis.text.y = element_text(size=rel(1.75)))+
```

```
  scale_color_manual(name = 'Signal Type', values = c("Red", "Blue", "Black", "DarkGreen"),
      labels = c('Error', 'Measured Position', 'Set Position', 'Signal to Motor'))

#Load the encoder data and plot
encoder = read.csv(file = 'encoder.csv', header = FALSE)
plot(x = c(1:nrow(encoder)), y = encoder[1:nrow(encoder),], xlab = 'Datapoint', ylab = 'IR
    Sensor Responce', main = 'Encoder Calibration Plot')
lines(x = c(0, 3450), y = c(70, 70), col = 'blue', lwd = 5)
```