2 DOF Laser Pointer Systems Group Report

White-Paper Updated - Mar 13, 2024

Stephen Qiao, ECE, University of British Columbia, Vancouver, BC, Canada

Steven Lee, ECE, University of British Columbia, Vancouver, BC, Canada

Abstract

A 2 DOF spherical wrist is developed. The system consists of two DC gearmotors that drives the pitch and yaw rotation of the attached laser pointer. The PID controller converts the desired angle to move the laser pointer in the XY plane and draws the desired shape.

In this paper, Section 1 describes motor selection. Section 2 describes the mechanical design of the spherical wrist that holds the laser pointer. Section 3 describes the simulation and tuning of the control system in MATLAB. Section 4 describes the electronics interfacing the micro-controller and motor. Section 5 describes the interrupt service routine in the microcontroller. Each section contains requirements, constraints, and goals (RCGs), along with relevant design decisions.

Nomenclature

RCG	Requirements, constraints, and goals
PID	Proportional, integral and derivative
ISR	Interrupt service routine
GPIO	General purpose input output
PWM	Pulse width modulation
CF	Control frequency
RPM	Revolutions per minute
DOF	Degree of freedom
ELEC_TF	Electrical Admittance Transfer Function
MECH_TF	Mechanical Admittance Transfer Function

1. Motor Selection

Table 1: Motor Selection RCGs

Requirement	Constraint	Goal
Power: 12V	~	~
~	Max current < 2.2 A	~
Speed < 350 RPM	~	~
~	Max power < 26.4 W	Operating power of 2.4 W should be reached for maximum efficiency

A motor is selected to have high torque and high RPM to be able to be controlled under high loads, eliminating the need for external custom gear trains. The motor needs to be operated under 12V for maximum speed. The motor is included with an encoder, eliminating the need for an external encoder. The specified motor is a JGA25-371 Geared Motor with Encoder. The motor is 53.6mm long x 25mm ϕ and has a 4mm ϕ shaft. The motor has a maximum speed of 350 RPM and a stall torque of 0.5099458 Nm. The motor with the encoder is shown in Fig. 1.

Figure 1: JGA25-371 Geared Motor with Encoder

Table 2: Motor Parameters

	Jm (Nms)	Km (Nm/A)	Bm (Nms)	Rw (Ω)	Lw (H)
Value	6.76976×10^{-4}	0.32740	8.93268×10^{-4}	6.191	2.516×10^{-3}

The motor parameters provided in Table 2 are found using a RLC meter, and deriving from the datasheet of the motor. A torque constant Km of the motor is determined by finding the equivalent back emf constant Kb (1) from the values in the datasheet. The most common way of finding the back emf constant Kb is by relating the operating voltage V_b and the angular velocity ω it operates at (2).

$$Km = Kb \quad (1)$$

$$Km = \frac{V_b}{\omega} \quad (2)$$

Damping constant Bm is determined by relating the torque τ and speed of the motor ω under no loads (3). Replacing τ with the Km we found and using the current i_w under no loads from the data sheet (4), Bm can be formulated.

$$Bm = \frac{\tau}{\omega} \quad (3)$$

$$\tau = Km \cdot i_w \quad (4)$$

Rotor Inertia, Jm, is calculated from measuring the mechanical time constant τ_m, and formulating with the torque constant Km and the resistance R values (5).

$$Jm = \frac{\tau_m Km^2}{R} \quad (5)$$

2. Spherical Wrist Design

Table 3: Mechanical Design RCGs

Requirement	Constraint	Goal
Must be able to rotate in the pitch and yaw axis	~	~
~	~	Maximize rotational speed
~	Rotation range < 90°	~
~	~	Loads on motor should have minimum inertial mass
Must be able to mount 2 motors and 1 laser pointer	~	~

A mechanical system for a 2 DOF laser pointer is designed in Solidworks. It consists of a stationary base and two main links. The base platform, acting as a mount for the yaw motor, has a larger surface area and greater mass compared to the rest of the mechanical system, ensuring it stays stationary. The dimensions of the base platform are displayed in Fig. 2, where it has equal length and width of 150mm and includes a 26mm diameter hole in the center, allowing the motor to fit.

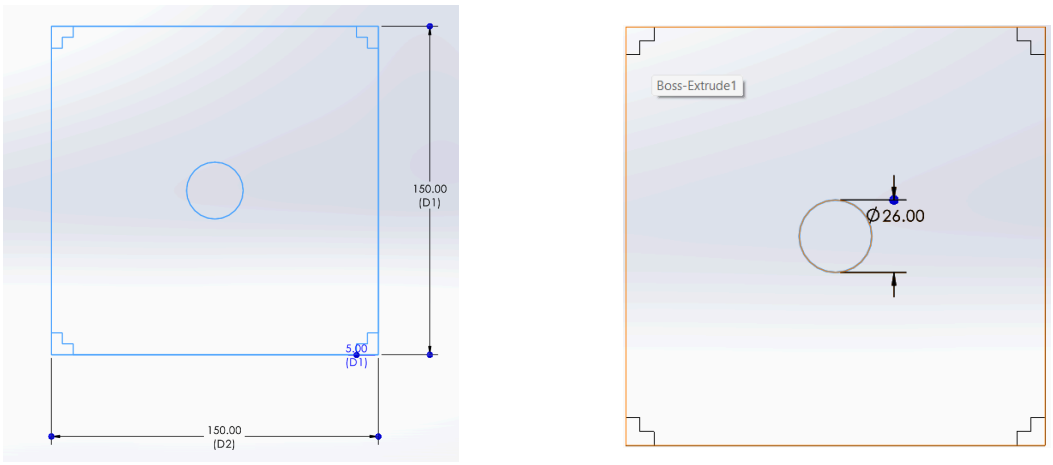

Figure 2: Base Platform Geometry

The rest of the mechanical system's mass is minimized to ensure lower torque is needed for optimal rotational control. Linkage 2 acts as a yaw rotational platform and connects the motor with the rest of the load. The linkage also satisfies the requirement of the laser pointer being able to move in the x-axis. The dimensions of Linkage 2 with both the rotational platform and the pitch motor mount attached is shown in Fig 3. Both components are ensured to have cylindrical holes with diameters that allows the motors to fit.

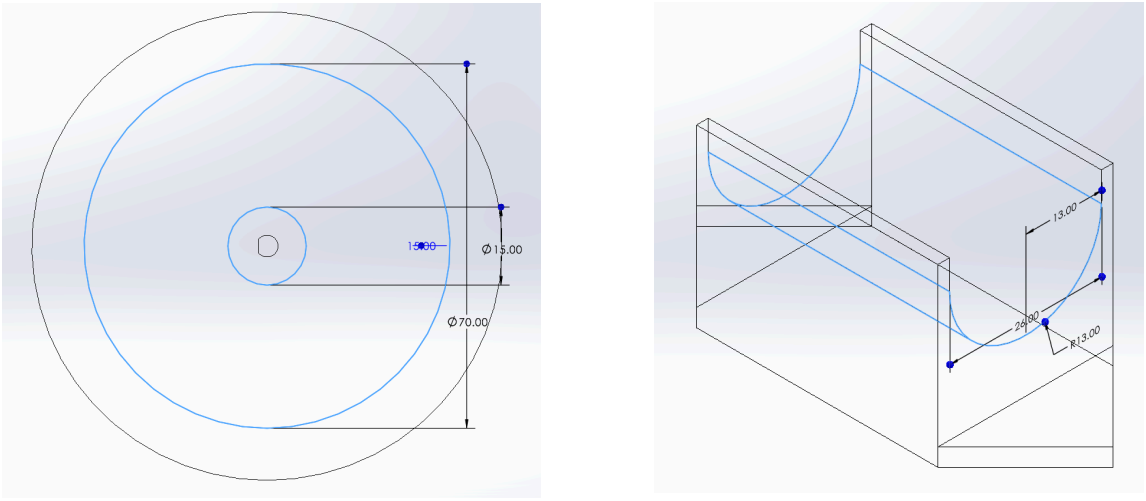

Figure 3: Linkage 2: Rotational Platform (Left), Pitch Motor Mount (Right)

Linkage 3 is a mount for a laser pointer, ensuring no translational movement. The laser pointer is able to rotate through its pitch axis through the gear train attached to the shaft of the mount,

which is freely contained within the laser mount. The dimensions of both the laser pointer mount and shaft is shown in Fig 4, with the mount having a height and length of 25mm and 5mm width. It also satisfies the need for lower friction of the rotation of the shaft by making the diameter of the hole bigger. The laser pointer shaft has a diameter of 4mm and length of 35mm to ensure the laser is able to move in the pitch axis.

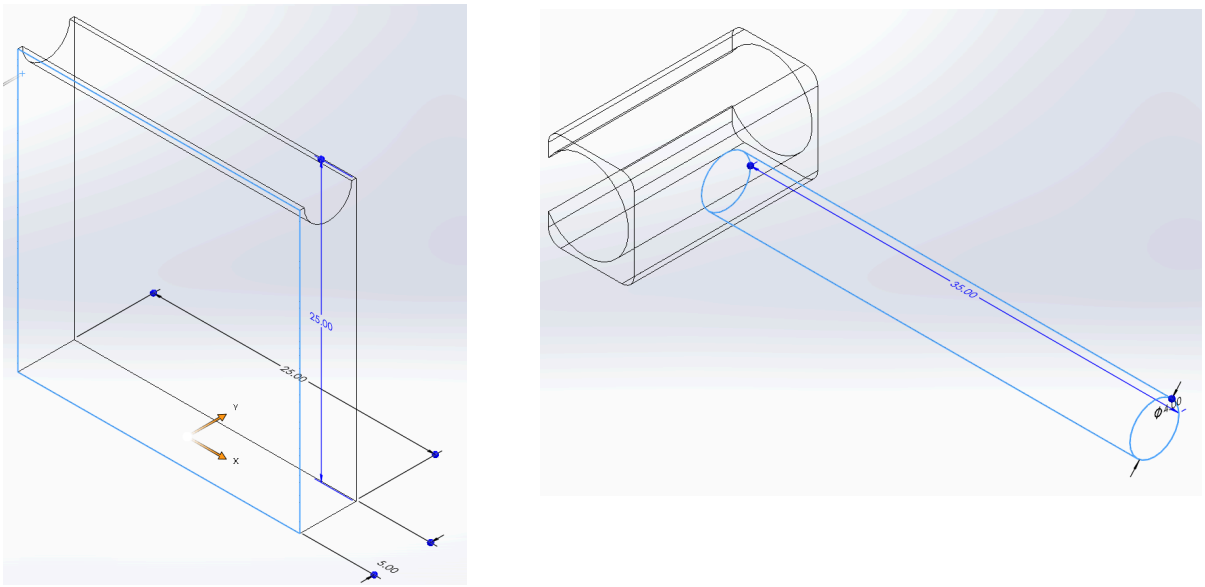

Figure 4: Linkage 3: Laser Pointer Mount (Left) & Laser Pointer Shaft (Right)

A gear train of 2 spur gears with a 1:1 teeth ratio, is used to transfer pitch axis rotations from the pitch motor shaft to the laser pointer shaft. The gear has a pressure angle of 20°, 78 teeths, tooth module of 0.4mm, and face width of 4mm.

Table 4: Mechanical Inertia Mass on Motor

Load	Inertial Mass (Kgm^2)
Laser Pointer Mount	1.6326E-7
Laser Pointer Shaft	1.1913E-7
MOT1 (Pitch Motor)	8.4827E-6
Pitch Motor Mount	6.0584E-6
Rotating Platform	9.54125E-5

Gear 1	1.8733E-7
Gear 2	1.8733E-7
Total = Laser Pointer Mount + Laser Pointer Shaft + MOT1 + Pitch Motor Mount + Gear 1 + Gear 2 + Rotating Platform	2.208466E-4

A mechanical parameter that needs to be considered is the inertial loads on the motors. Inertial loads were determined from using Solidworks. Solidworks was used to closely simulate the PLA material used for 3D printing. In Table 4, the total load on top of the motor is calculated and shown. The added load J_{load} needs to be included into the motor's mechanical admittance Ym (6). The load on the motor's shaft is added to the rotor inertial Jm, decreasing the motor system's Ym (7).

$$Ym = \frac{1}{J_{Total} \cdot s + Bm} \qquad (6)$$

$$J_{Total} = Jm + J_{load} \qquad (7)$$

Any extra damping constant or friction can be neglected, as the values are relatively small.

3. Simulation and Tuning with MATLAB

Table 4: MATLAB Simulation RCGs

Requirement	Constraint	Goal
Settle time < 0.5s	~	Settle time as small as possible
Overshoot < 1 %	~	Overshoot as small as possible
PID Transfer function	~	~
~	~	Steady state error as small as possible

In the control system's model in Figure 5, a motor is the plant that is being controlled. The motor's transfer function is a feedback loop that contains a ELEC_TF, MECH_TF, and Km. A

Vgain is applied after the PID output to convert the duty cycle of 0-100% to a voltage input that is used as the operating voltage for the motor. The PID gain values are found by using ELEC 341's 15-Step Design Process. First the partial dynamics is determined to obtain the temporary gain K0 and the double zeros. Then, K0 is tuned to a desired PM or performance metrics to determine the master gain K. Lastly, the PID gains Kp, Ki, and Kd are heuristically tuned to find the best settle time, rise time and overshoot. All the MATLAB code for these steps can be found in the Appendix. An integrator needs to be applied after the motor's transfer function to ensure the position of the motor can be determined. A sensor is required in the feedback path of the control system, to measure the actual angle of the motor. The encoder is the sensor that has a resolution of 360 ticks per revolution.

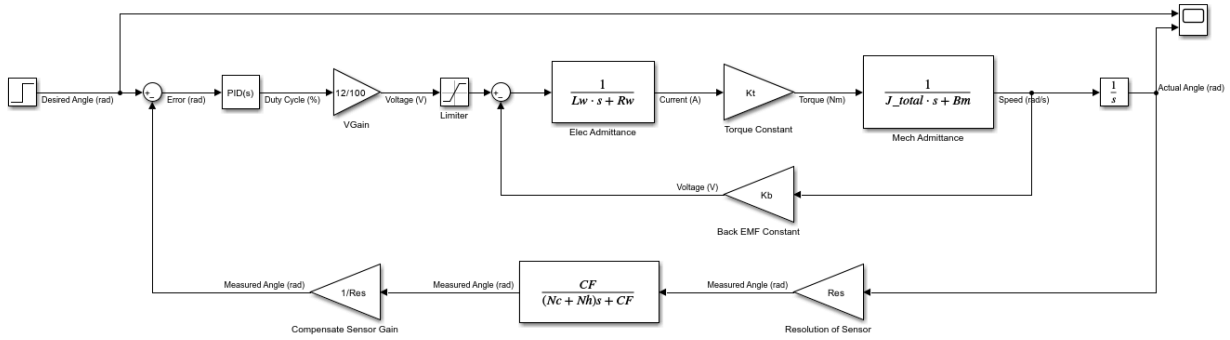

Figure 5: Simulink Model

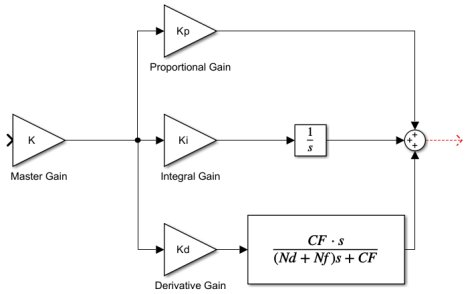

Figure: 6 PID Model

An ideal and actual step response of the motors speed at 12V is shown in Figure 7. The ideal curve in Figure 7 has a settling time of 0.073 seconds, a rise time of 0.043 seconds and 0 overshoot. However, the ideal step response is unachievable with limits in the real world. The

actual step response with limitations has a settling time of 0.2158 seconds, a rise time of 0.08 seconds, and 3.425% overshoot.

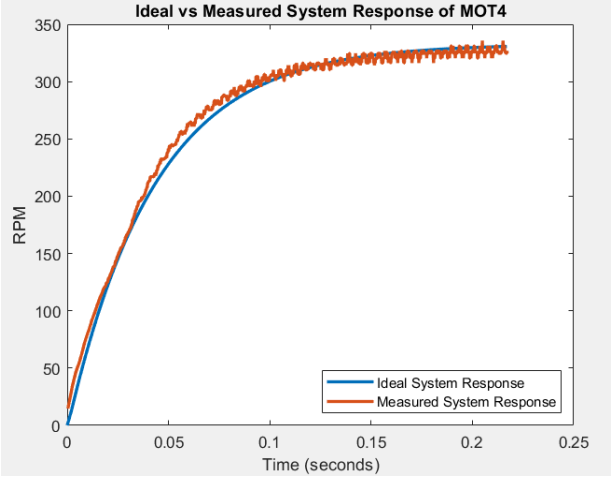

Figure 7: Ideal and Tuned Step Response with Approximated Motor Model

4. Microcontroller

Table 5: Microcontroller RCGs

Requirement	Constraint	Goal
>= 9 GPIO Pins	~	~
5V output	~	ATMega328P outputting 5V when duty cycle is 100%
Through-hole MCU	~	~

The team requires the use of at least nine GPIO pins on the MCU. This is because the decoder transmits the current angle values in a 7-bit resolution, with each bit sent through a separate wire, and the H-bridge needs two pins, each controlling one direction of the motor. The ATMega328P, offering 23 GPIO pins, amply meets our design requirements. Additionally, the H-bridge operates at a specific voltage, and the ATMega328P is capable of supplying 5V at a 100% duty cycle, which is suitable for the H-bridge. Initially, the team considered the PIC32, STM32, or

ATMega328P. However, the PIC32 was ruled out as it can only output 3.3V, failing to meet the design requirement. The STM32 was also discarded because it is not available in a through-hole format. Consequently, the ATMega328P was selected as the most suitable option.

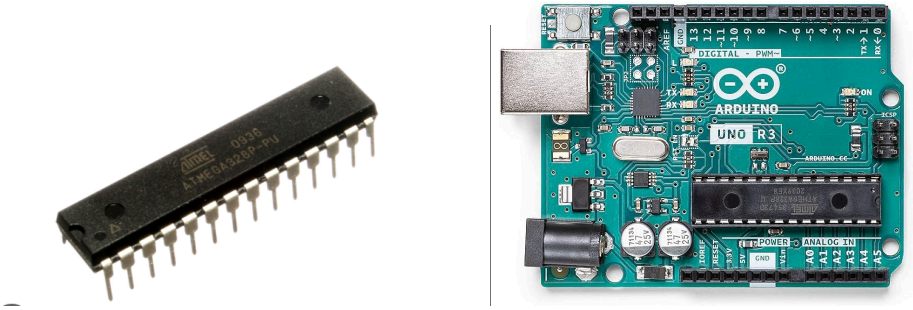

Figure 8: Through-hole ATMega328P Microcontroller (LEFT), Arduino Uno (RIGHT)

5. Interrupt Service Routine (ISR)

Table 6: Interrupt Service Routine RCGs

Requirement	Constraint	Goal
> 100Hz control frequency with ~50% Timer1 ISR Utilization	~	Achieve Timer1 ISR utilization of 50%, and satisfy 100Hz or greater ISR frequency for both Timer1 and 2.
>= 2 Usable Timer ISRs	~	Split processing work between 2 timers
Adjustable duty cycle from 0-100%	Consistent duration of on-time.	Achieve self-adjusting duty cycle functionality from PID control output values.
Timer1 ISR Includes PID calculation and sensor processing	~	Achieve speed control based on PID output

Real-time Weighted Sum Filter		Reduce noise from Derivative term of the PID

The firmware program for the MCU is composed of Timer1 and Timer2 Interrupt Service Routines (ISRs). In the Timer1 ISR, the main calculation processes such as calculations for the current angle, motor speed, and PID control are executed. Meanwhile, the Timer2 ISR is responsible for determining the motor's direction and toggling the pins to generate PWM signals. Therefore, Timer1 ISR utilization is static and needs to be set in a way that would give the main loop ample execution time. The program employs the micros() built-in function to calculate the time difference (dt), which is needed for computing the Integral (I) and Derivative (D) components of the PID control. This function relies on Timer0 to count the time in microseconds.

Timer1 ISR operates in CTC mode, where the timer counts up to OCR1A register value triggering the ISR. The register value has been set so that the ISR will be triggered at 1.25kHz, which is set based on ~50% Timer1 ISR utilization when the ISR run-time is around 400us (2.5kHz -> 1.25kHz @ 50% utilization). 50% Timer1 ISR utilization is set so that the program can give sufficient time for the main loop. See Appendix B for ISR code.

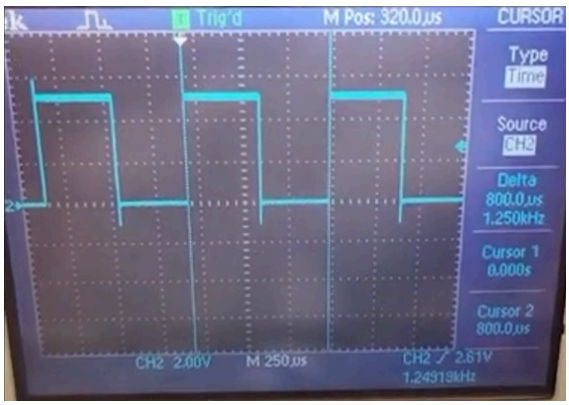

Figure 9: 50% Timer1 ISR Utilization

The Timer1 ISR runs the PID calculation (see Appendix C) using the Proportional, Integral, Derivative equations below:

$$P = error * Kp \qquad (8)$$

$$I = Ki * (I + error * dt) \quad (9)$$

$$D = (error - previous\ error) / dt \quad (10)$$

The integral wind-up needs to be taken account for in that it may cause extreme overshoots and prolonged oscillations due to too much error accumulation. Through numerous trials and errors, clamping values were found to restrict the wind-up.

```
volatile float kp = 0.2;  // Increasing speeds up response and removes overshoot
volatile float ki = 0.90;  // Increasing removes steady state error
volatile float kd = 0.01;    // Increasing slows down response and removes spikes
volatile float integral_max = 10;
volatile float integral_min = -10;
```

Figure 10: Integral Clamping Values

The Finite Differential Derivative Weighted Sum Filter is implemented within the PID function to smoothen the volatility of the Derivative term as the PID is run continuously by the ISR.

Timer2 ISR operates in fast-PWM mode, where the Timer2 counter ascends from 0 to 255, at which the Timer2 ISR is triggered, so the Timer2 ISR is activated every 255 timer count. Using the output of the PID controller, the firmware adjusts the OCR2A register linked to pin 13 or the OCR2B register associated with pin 3, setting values ranging from 0 to 255. Consequently, throughout the maximum 255 timer count duration of the Timer2 ISR, the MCU can activate the pins proportionally to Timer2 ISR max duration, generating a PWM signal. This capability allows the firmware to modulate the duty cycle of the pin outputs from 0 to 100%, while keeping the pulse lengths highly consistent.

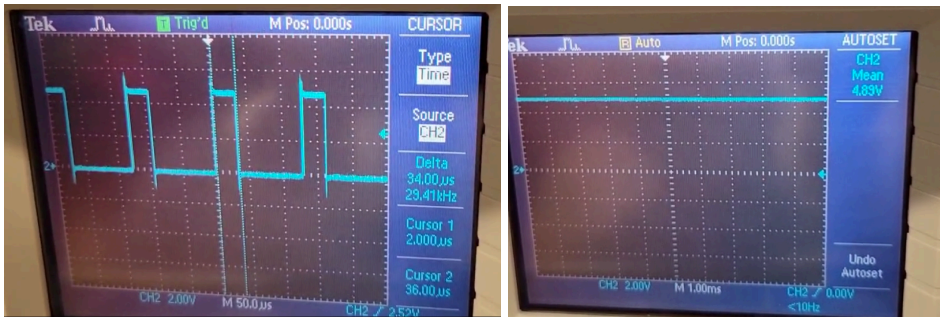

Figure 11: PWM at 25% Duty Cycle (LEFT), PWM at 100% Duty Cycle (RIGHT)

For laser imaging tasks, it is important for the motor to move at a sufficiently high speed. The motor speed requirement is based on intuition, concluding that it is feasible and sufficiently quick to operate the control period in milliseconds. Therefore, the maximum control period is set to 10ms, corresponding to a minimum frequency of 100Hz. The MCU has a clock frequency of 16MHz and uses Timer2 ISR fast-PWM mode to toggle the pins on and off while having the prescaler set to 8. The relevant equation to find the control frequency is as follows:

$$frequency = \frac{Clock\ Frequency}{Prescaler * (Max\ Timer\ Count + 1)} (11)$$

Using this equation, the control frequency of the Timer2 ISR is calculated to be 7.8125kHz, which satisfies the speed requirement.

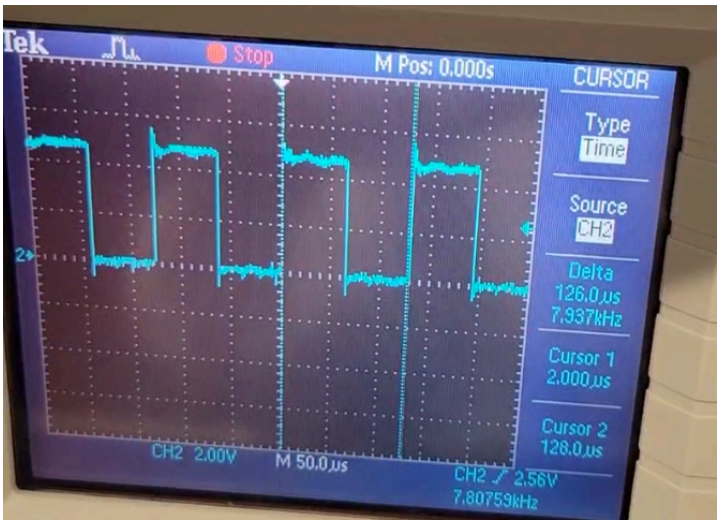

Figure 12: Measuring Control Frequency Experimentally

APPENDIX

Appendix A - MATLAB Modeling Code

```
%% ELEC 391 Linear Model
%% MOT4 Motor Parameters
Jm = 6.76976*10^(-4); % Nms^2
Kb = 0.32740; % Vs/rad
Kt = 0.32740; % Nm/A
Bm = 8.93268*10^(-4); % Nms
Rw = 6.191; % ohms
Lw = 0.002516; % Henrys
Jl = 0.000110235989999999999;
J_total = Jm + Jl;

CF = 1/(124*10^(-6));
Nc = 0.5; % Duty Cycle
Nh = 0.5; % Control Signal Held
Nd = 0.5; % FDD
Nf = (15*7-42)/65; % WSF

s = tf('s');
vgain = 12/100; % 12V per 100% duty cycle

Res = 2*pi/360;

%% Identify Sub Systems
%% Motor TF
Ye = 1 / (Lw*s + Rw);
Ym = 1 / (J_total*s + Bm);
motor_tf = feedback(Ye*Kt*Ym, Kb);

%% Plant TF
Gp = vgain * motor_tf * (1/s);

%% Sensor TF
Hs = CF / ((Nc+Nh)*s + CF);

%% Feedback path
feedback_path = Hs * Res * (1/Res); % Equal to Hs
```

```
Dp = (CF/((Nd+Nf)*s + CF));

[Dp_poles, Dp_zeros] = pzmap(Dp);

poles = Dp_poles';

K0 = margin(Dp*Gp*Hs) % find temporary ultimate gain

nyqlog(K0*Dp*Gp*Hs); % Check for stability

[~,~, wcg, wcp] = margin(K0*Dp*Gp*Hs);

% Get zeros
[Zresult, maxPM] = newtonsCCzeroPID(-wcp, K0*Dp*Gp*Hs)

zeros = [Zresult, conj(Zresult)]

Dz = (s-zeros(1))*(s-zeros(2))/(zeros(1)*zeros(2))

D = Dz * Dp;
```

```
%% Find K for Desired PM
tuned_gain = 0;
best_PM = Inf;
current_gain = 1;

desired_PM = 45;
resolution = 0.1;
```

```
% Iterate over gain values
while true
    % Apply current gain to the system
    sys = current_gain * D * Gp * Hs;

    % Calculate phase margin
    [~, PM] = margin(sys);

    % Check if this gain gives a phase margin closer to the desired value
    if abs(desired_PM - PM) < abs(desired_PM - best_PM)
        tuned_gain = current_gain;
        best_PM = PM;
    end

    % Break the loop if phase margin starts increasing again (assumes unimodal behavior)
    if PM > desired_PM && current_gain > 1
        break;
    end

    % Increase the gain for the next iteration
    current_gain = current_gain + resolution;
end
```

```
%% Heuristic Tune Gains
% Initial PID Control
K = tuned_gain;
init_Kp = 1;
init_Ki = 1;
init_Kd = 1;

init_PID_Controller = pid(init_Kp, init_Ki, init_Kd, 0);
init_cl_sys = feedback(K*init_PID_Controller*Gp, Hs);

figure;
step(init_cl_sys)
init_sys_stepResults = stepinfo(init_cl_sys)

% Tuned PID Control
K = tuned_gain;
Kp = 19;
Ki = 0.8;
Kd = 0.2;

PID_Controller = pid(Kp, Ki, Kd, 0);
cl_sys = feedback(K*PID_Controller*Gp, Hs);

figure;
step(cl_sys)
sys_stepResults = stepinfo(cl_sys)
```

Appendix B - ISR C Code

Timer1 and Timer2 ISR Setup

```
void setup() {
  Serial.begin(9600);

  // Decoder Input Pins
  pinMode(bit0, INPUT);
  pinMode(bit1, INPUT);
  pinMode(bit2, INPUT);
  pinMode(bit3, INPUT);
  pinMode(bit4, INPUT);
  pinMode(bit5, INPUT);
  pinMode(bit6, INPUT);
  pinMode(bit7, INPUT);

  // Output Pins
  pinMode(pwmLeftPin, OUTPUT);
  pinMode(pwmRightPin, OUTPUT);

  // Setup Timer 1
  noInterrupts();  // Disable all interrupts

  // Timer1 Setup
  TCCR1A = 0;
  TCCR1B = 0;
  TCNT1 = 0;
  OCR1A = 1599;              // Set for 1.25kHz Control Freq
  TCCR1B |= (1 << WGM12);    // CTC mode
  TCCR1B |= (1 << CS11);     // Prescaler 8
  TIMSK1 |= (1 << OCIE1A);  // Enable Timer1 interrupt

  // Timer2 Setup
  TCCR2A = 0;  // Clear Timer2 control register A
  TCCR2B = 0;  // Clear Timer2 control register B
  TCNT2 = 0;   // Initialize counter value to 0

  // Set to Fast PWM mode
  TCCR2A |= (1 << WGM21) | (1 << WGM20);

  // Set prescaler to 8 (or choose another suitable prescaler)
  TCCR2B |= (1 << CS21);

  // Non-inverting mode for both OC2A and OC2B
  TCCR2A |= (1 << COM2A1) | (1 << COM2B1);

  OCR2A = 0;
  OCR2B = 0;

  interrupts();  // Enable all interrupts

  // Read initial decoder val
  readDecoder();
  previous_decoder_val = current_decoder_val;
}
```

Timer1 ISR

```
ISR(TIMER1_COMPA_vect) {
  // Get new desired position

  // Read from decoder and get absolute position
  getAbsolutePosition();

  // PID Computation
  pidControl();

  // Set Motor Speed
  setMotorSpeed();
}
```

Timer2 ISR

```
ISR(TIMER2_COMPA_vect) {

  // Motor Direction Control
  motorDirectionControl();

  // Store previous error
  error_previous = error;
}
```

Appendix C - Calculation C Code

PID Controller with Weight Sum Filter

```
void pidControl() {
  // time difference
  current_time = micros();

  delta_time = ((float)(current_time - previous_time))/1.0e6; // in seconds
  previous_time = current_time;

  // error
  error = desired_angle - absolute_angle;

  // derivative
  error_derivative = (error - error_previous) / (delta_time);

  FDD_WSF_PID();

  // integral
  error_integral = error_integral + error * delta_time;

  if(error_integral > integral_max){
    error_integral = integral_max;
  }
  else if(error_integral < integral_min){
    error_integral = integral_min;
  }

  // control signal
  pid_out = k * (kp * error + kd * filtered_error_derivative + ki * error_integral);

}
```

Weighted Sum Filter

```
void FDD_WSF_PID() {
    sum = 0.0;

    // Shift all the derivative samples to the right by 1 index
    for (int i = WSF_SAMPLE_COUNT - 1; i > 0; i--) {
        raw_deriv_samples[i] = raw_deriv_samples[i - 1];
    }

    // Insert new derivative data to the first index
    raw_deriv_samples[0] = error_derivative;

    // Sum all of them
    for (int i = 0; i < WSF_SAMPLE_COUNT; i++) {
        sum += raw_deriv_samples[i] * WSF_CONST_LOOKUP_TABLE[i];
    }

    // Update filtered error derivative
    filtered_error_derivative = sum;
}
```

Encoder Reading Logic

```
void readDecoder() {
  bit0_state = (PINB & (1 << PB2)) >> PB2;
  bit1_state = (PIND & (1 << PD4)) >> PD4;
  bit2_state = (PIND & (1 << PD5)) >> PD5;
  bit3_state = (PIND & (1 << PD6)) >> PD6;
  bit4_state = (PIND & (1 << PD7)) >> PD7;
  bit5_state = (PINB & (1 << PB0)) >> PB0;
  bit6_state = (PINB & (1 << PB1)) >> PB1;
  bit7_state = (PIND & (1 << PD2)) >> PD2;

  binaryToDecimal();

  decoder_deg_sign = bit7_state;
}

void binaryToDecimal() {
  current_decoder_val = bit6_state * 64 + bit5_state * 32 + bit4_state * 16 + bit3_state * 8 + bit2_state * 4 + bit1_state * 2 + bit0_state;
}

void getAbsolutePosition() {
  readDecoder();

  // Determine direction and handle overflows/underflows
  if (decoder_deg_sign == 1) {  // clockwise
    if (previous_decoder_val > current_decoder_val) {
      // Handle overflow
      rotations++;
    }
  } else {  // counterclockwise
    if (previous_decoder_val < current_decoder_val) {
      // Handle underflow
      rotations--;
    }
  }

  previous_decoder_val = current_decoder_val;

  absolute_angle = rotations * (MAX_DECODER_VALUE + 1) + current_decoder_val;
}
```

Appendix D - Motor Control Code

```
void setMotorSpeed() {
  // Get motor speed
  motor_speed = fabs(pid_out);
  if (motor_speed > 255) {
    motor_speed = 255;
  }

  // Get motor direction
  if (pid_out > 0) {  // motor needs to move clockwise
    motor_deg_sign = 1;
  } else {
    // motor needs to move counterclockwise
    motor_deg_sign = 0;
  }

  if (motor_deg_sign == 1) {
    OCR2A = motor_speed;
    OCR2B = 0;
  } else {
    OCR2A = 0;
    OCR2B = motor_speed;
  }
}
```

```
void motorDirectionControl() {
  if (motor_deg_sign == 1) {
    PORTD &= ~(1 << PD3);  // Low
    PORTB |= (1 << PB3);   // High
  } else {
    PORTB &= ~(1 << PB3);  // Low
    PORTD |= (1 << PD3);   // High
  }
}
```