Project
ME 5245 Mechatronic Systems

BONUS TASKS

Author: Tirth Patel, Joshua Yoo

Task 1 (FFT)

FFT of the rotor speed data and the dominant frequency components in the same are shown below.

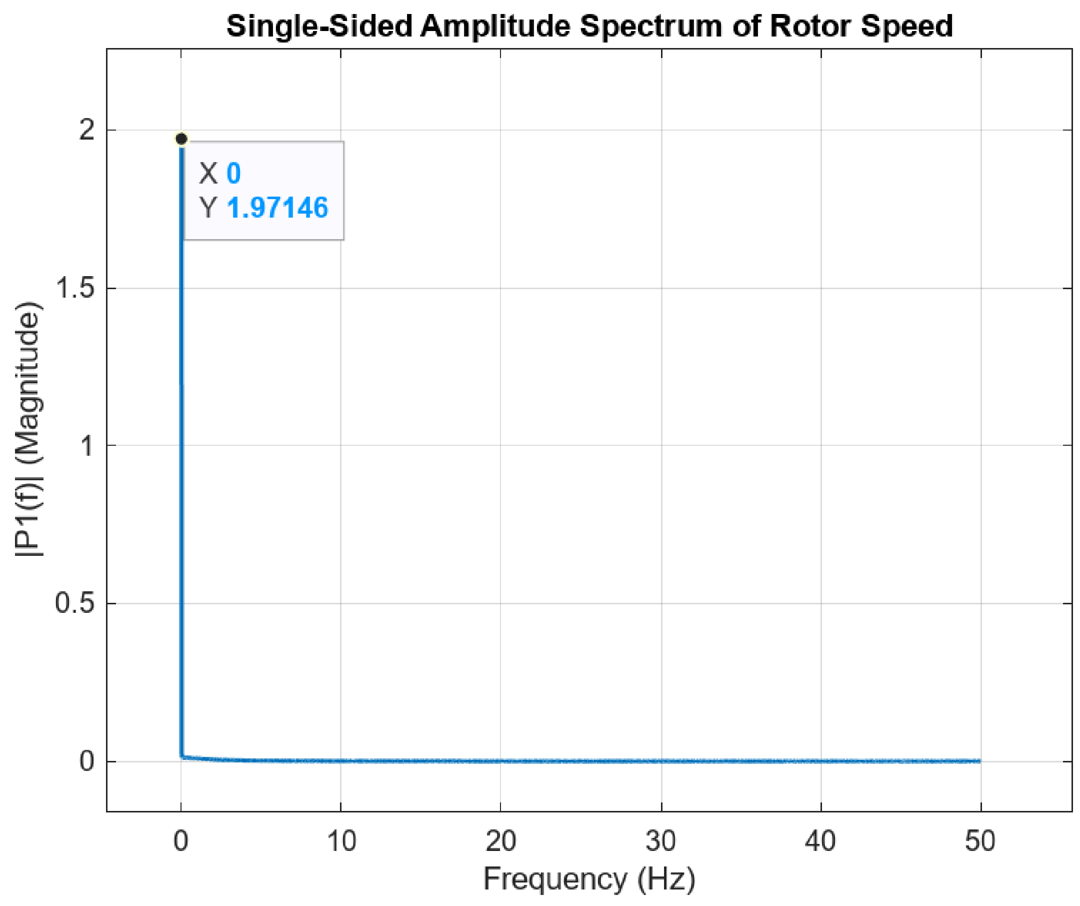

Figure 1: FFT of the rotor speed data

In the above plot, high magnitude near 0Hz represents the fundamental motion or slow drifts. Here the maximum frequency we can observe is the Nyquist frequency, which is half the sampling rate ($F_s/2$). Hence, the arduino is running at 100Hz, because we can reliably see frequencies up to 50Hz here.

Task 2 (Low Pass Filter)

The noise is primarily broadband quantization noise from the encoder steps. Since the signal is mostly DC motor speed and the noise is high frequency jitter the cutoff frequency we choose is 10H. Because the motor's physical inertia prevents it from reacting meaningfully to commands faster than ~5-10Hz. Anything above this noise.

$$Time\ Constant\ \tau = \frac{1}{2\pi f_c} = \frac{1}{2\pi(10)} \approx 0.016\ seconds$$

Modified Simulink model and comparison plot are as follows:

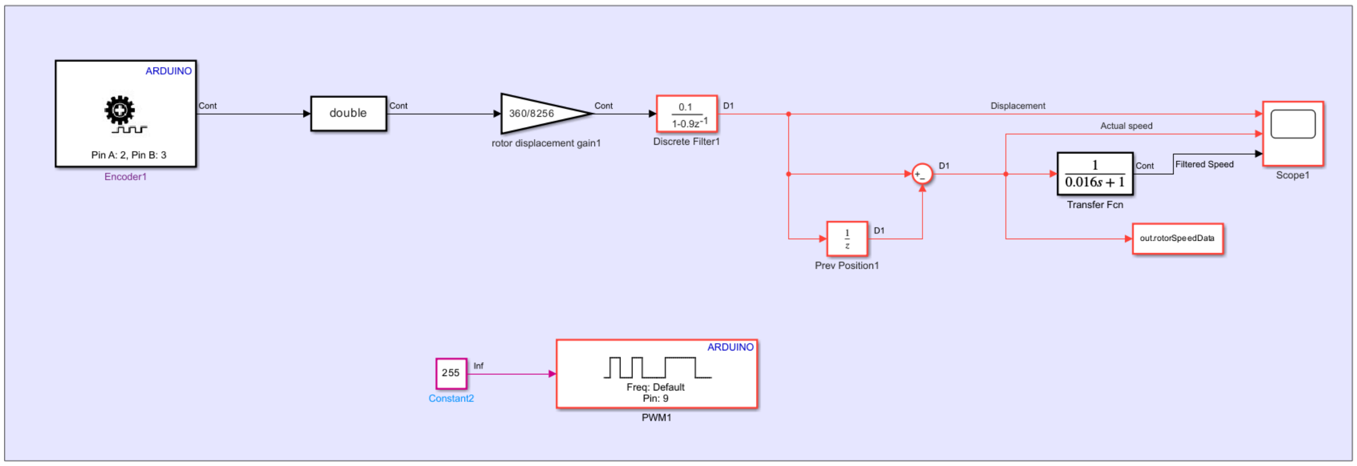

Figure 2: Modified Simulink model with transfer function

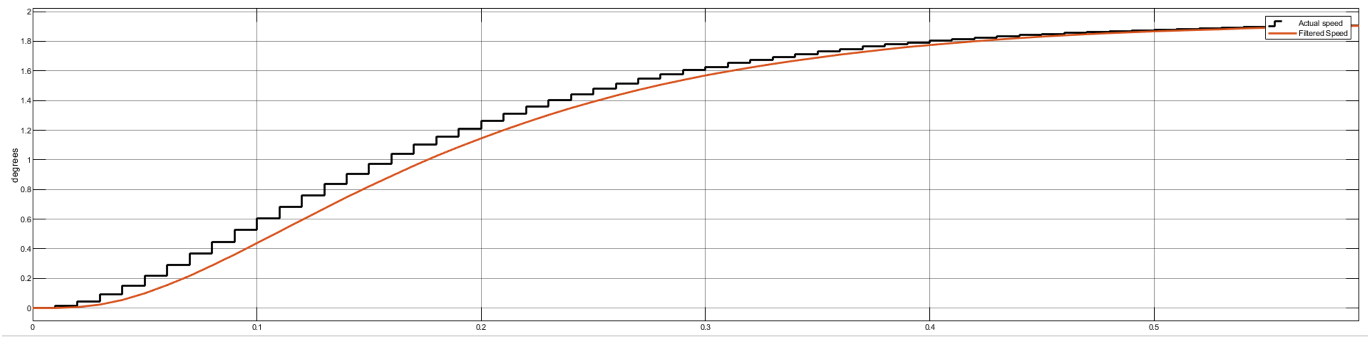

Figure 3: Comparison–actual speed vs. filtered speed

After running the experiment, the raw signal looked steppy and jagged like a staircase, while the filtered signal cut through the center of the steps, creating a smooth line. The tradeoff here is the phase delay introduced by the filter between both lines.

Task 3 (Simscape Motor Model)

Noise was injected into the Simscape model with a Band-Limited White Noise block and a PID controller was designed to have the fastest settling time, and rise time, while having minimal to no overshoot.

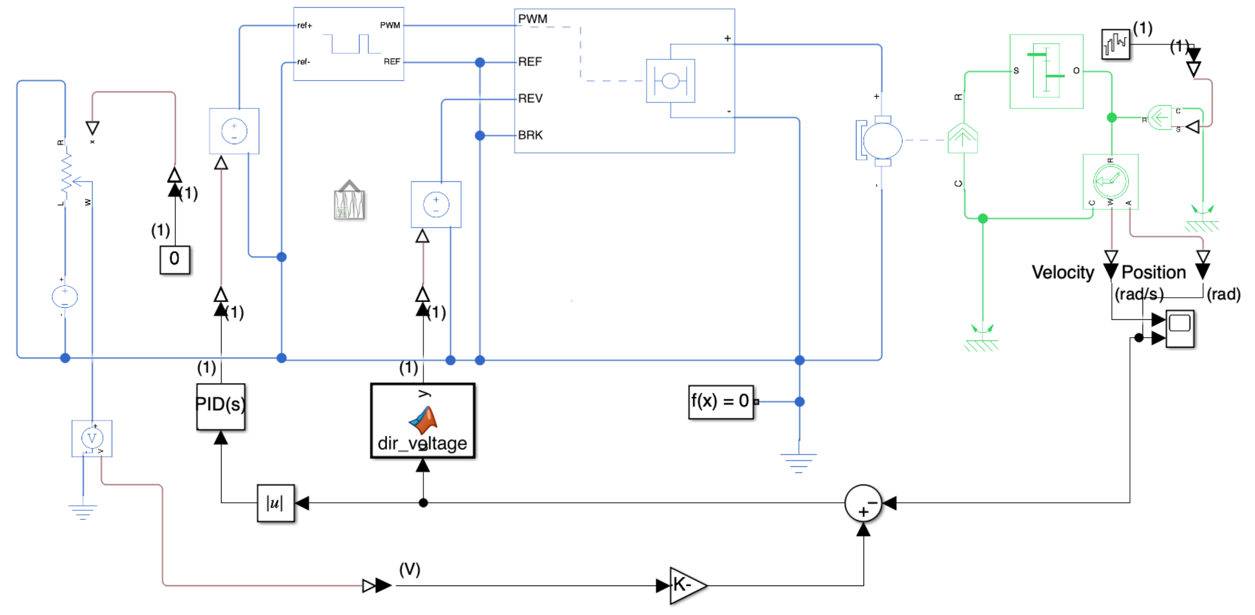

Figure 4: Simscape Model for Motor Position Control w/ Injected Noise

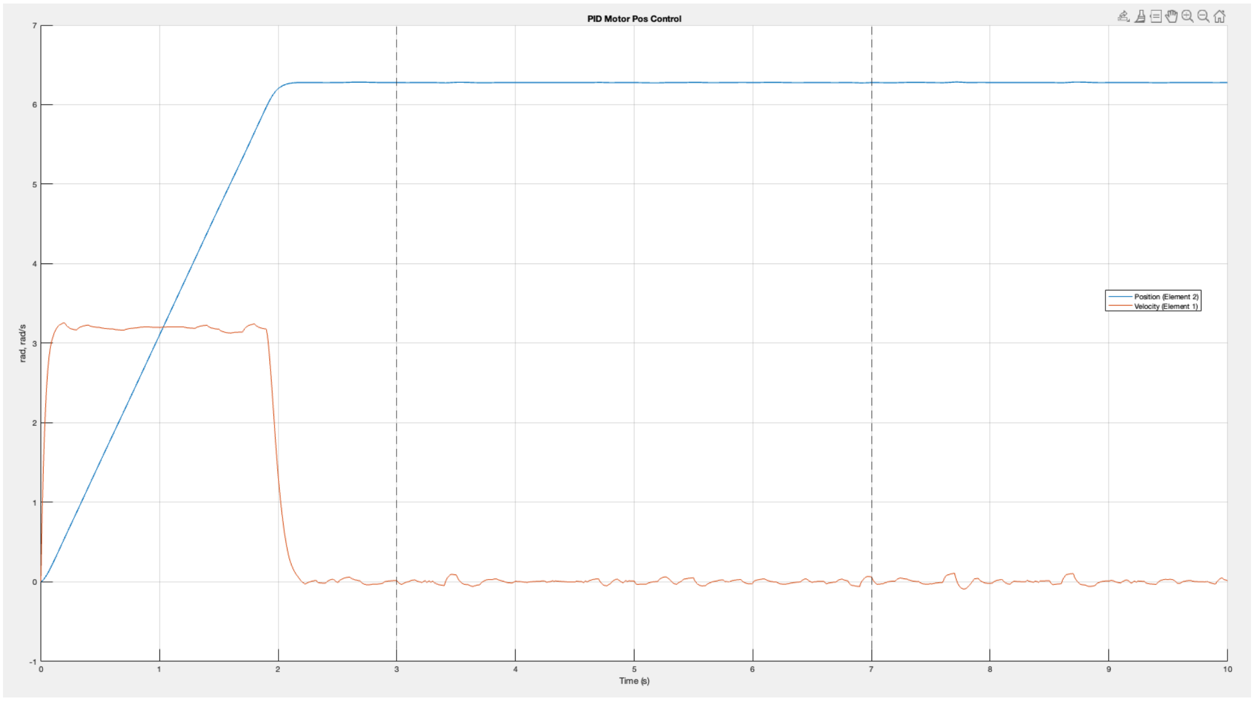

Figure 5: Position Control w/ new PID Controller and Injected Noise

With a **proportional gain of 16**, an **integral gain of 0.0095**, and a derivative gain of 0, the model proved to have the best performance in regards to the criteria designed around. In Figure ##, the following are found for design parameters.

Steady-Error (rad)	0.006
Rise Time (s)	1.57
Settling Time (s)	1.97
Overshoot	0.1 %

Task 4 (New PID into Position Control)

The new PID parameters were placed into the Simulink for hardware testing. The plots below are the results:

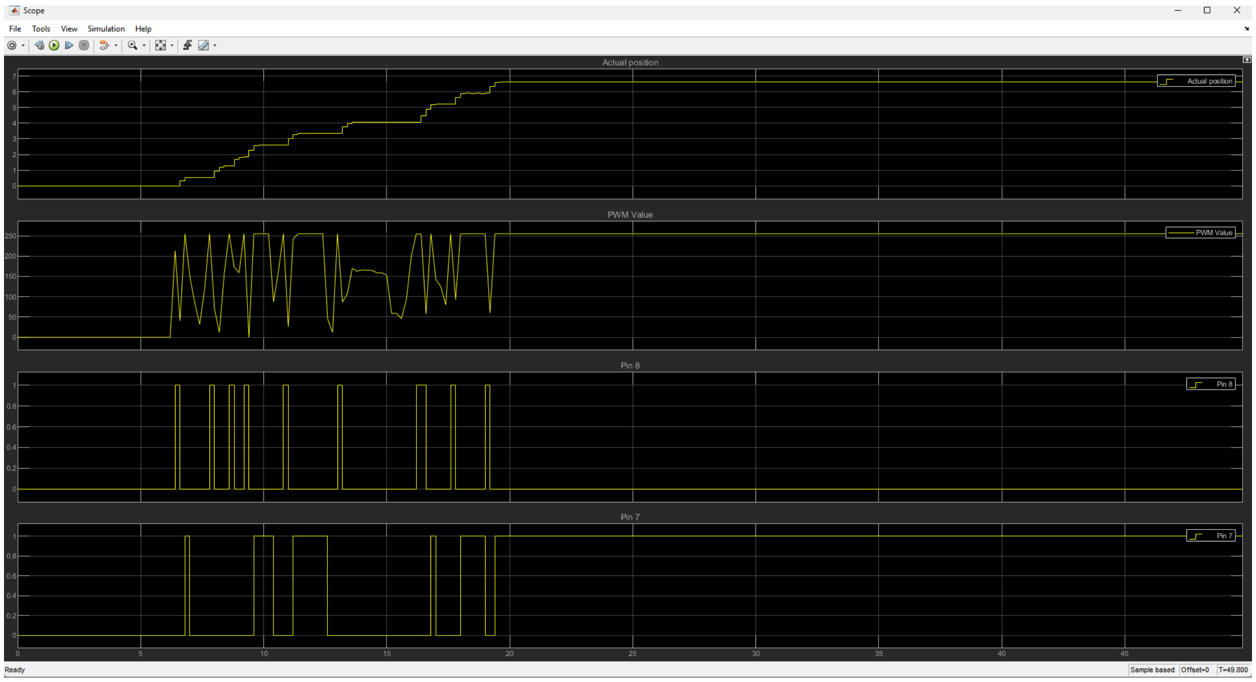

Figure 6: Hardware Testing Behavior from New PID Controller

PWM is much more aggressive than Part C reaction and manages to not overshoot at different positions of the motor. It's even apparent that the position is taking less "steps" to reach setpoint and taking larger steps to achieve setpoint faster.

Task 5 (Arduino Code: Pos CTRL)

MATLAB's Simulink Coder app was utilized to grab a C++ version from the Simulink model that was made for Control Objective (i). From the C++ code, a clean Arduino code was made to work with the hardware while being aware that direction control is still a limitation. The explicit code can be seen in Appendix A.

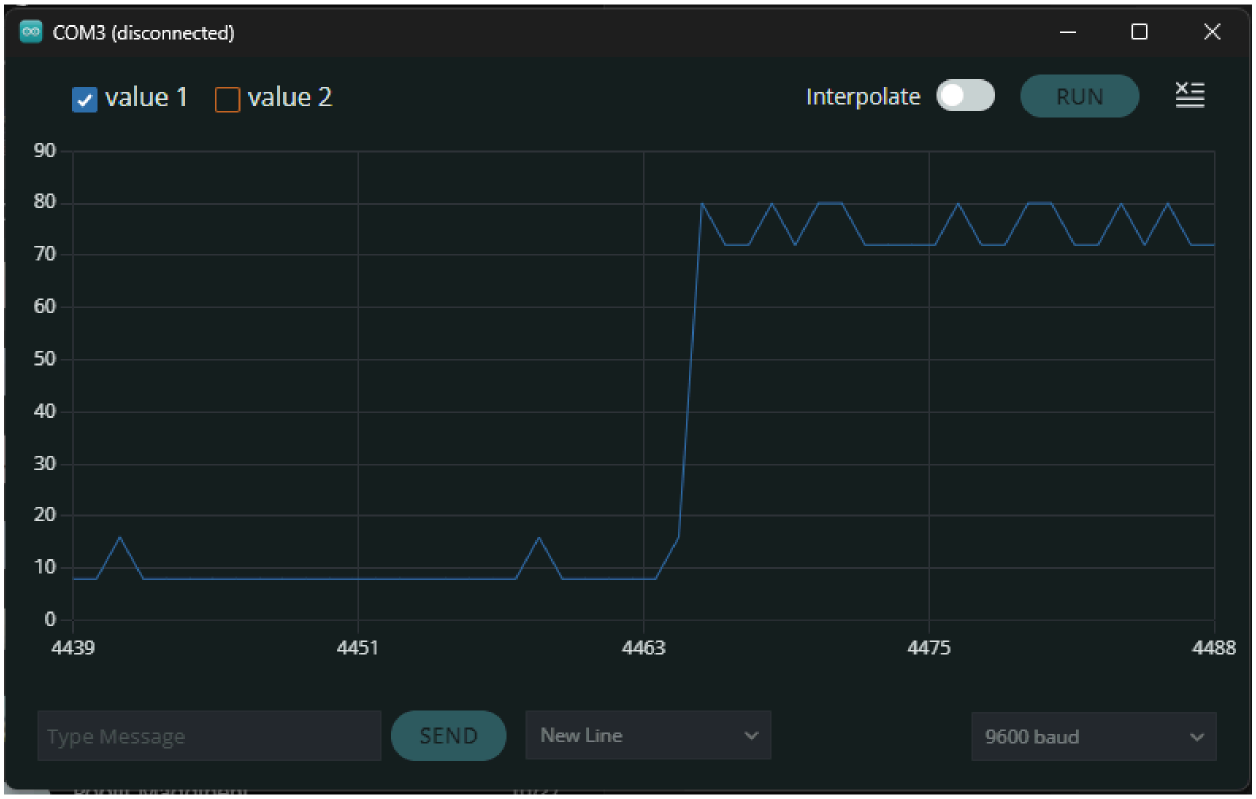

Figure 7: Target Position from Potentiometer using Arduino

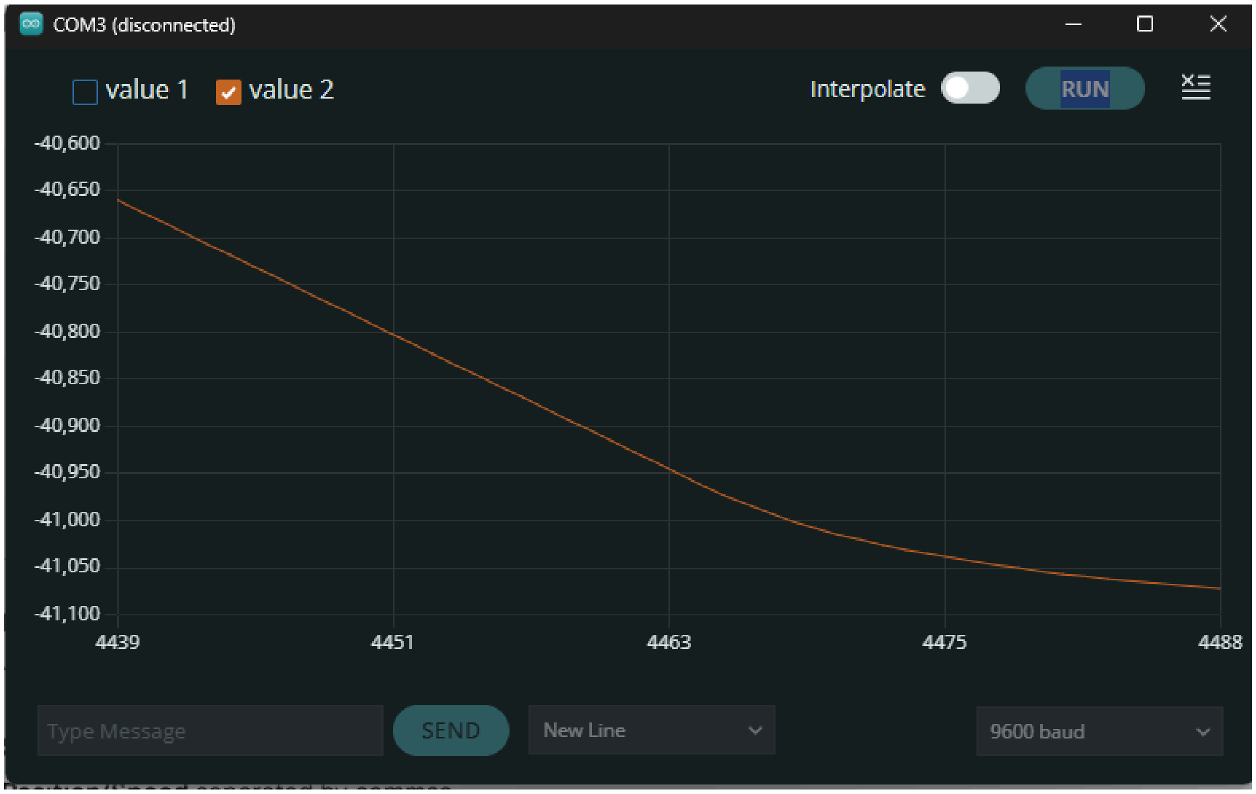

Figure 8: Encoder Position using Arduino

The potentiometer setpoint rises to around 72–80 (scaled command units), and the motor's encoder position responds by moving steadily in the commanded direction, showing that the controller is actively driving the system toward the target. The large negative encoder value simply reflects an accumulated count reference and direction convention, and the smooth, consistent trend indicates stable measurement and continuous corrective action. After the setpoint change, the response settles into a narrow band, which is consistent with the position loop reaching and maintaining the desired position while exhibiting only small fluctuations expected from ADC noise in the potentiometer and quantization in the encoder.

Task 6 (Hardware-Simulation Control Comprision)

Simulink hardware experiment velocity data collected and plotted, which shows the following behavior with the P control.

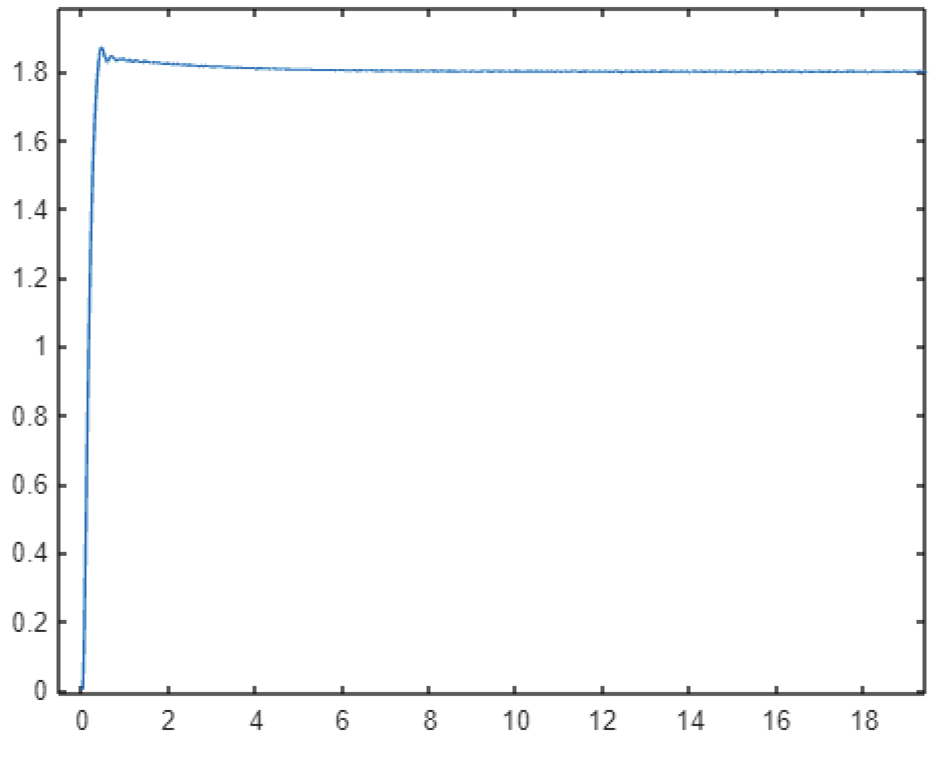

Figure 9: Hardware experiment velocity vs. Time

Same for simscape shows the following behavior

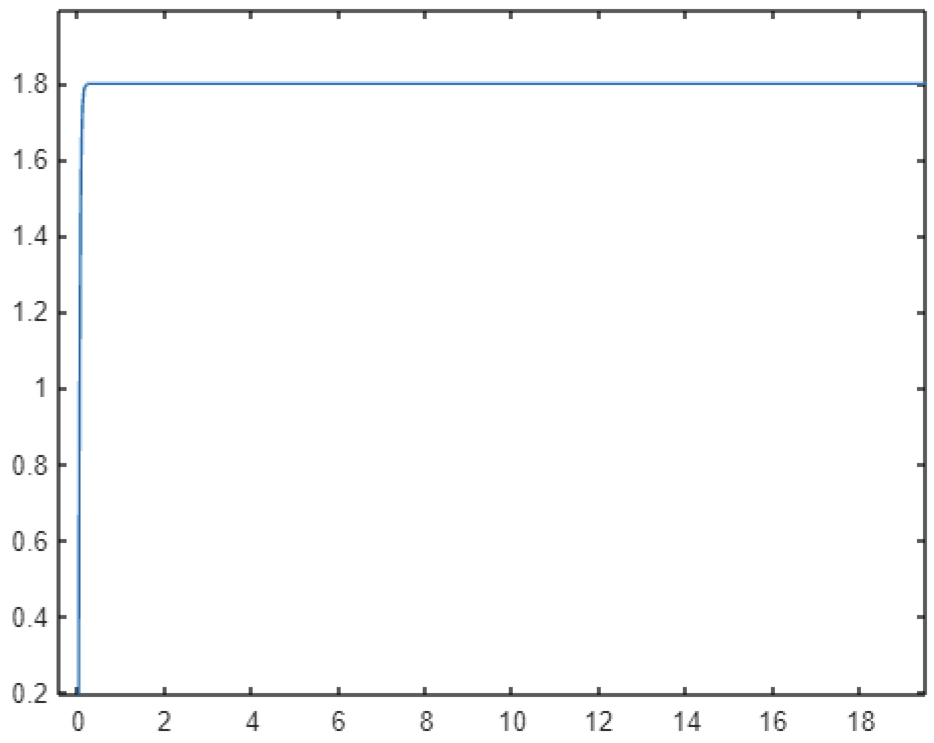

Figure 10: Simscape experiment velocity vs. Time

The control metrics for the both are as mentioned below.

Performance Metrics		
Metric	**Hardware**	**Simscape**
Rise Time (s)	0.2266	0.0569
Settling Time (s)	0.9406	0.1095
Overshoot (%)	3.91%	0.00%

Table 1: Performance Metrics Hardware vs. Simscape

1. Rise Time (t_{rise}) Comparison
   - **Observation:** The hardware rise time ($0.2266s$) is approximately **4x slower** than the Simscape prediction ($0.0569s$).
   - **Justification:**
     - **Voltage Drop:** The Simscape model likely assumes the motor receives the full battery voltage (approx. 7.5V). In reality, the L293D motor driver [1] is a BJT-based H-bridge that incurs a significant internal voltage drop of **1.2V to 1.5V**. This means the physical motor only receives ~6.0V, resulting in significantly less torque and slower acceleration.
     - **Battery Sag:** Under the high current load of acceleration, the AA batteries likely experienced voltage sag, further reducing the drive power compared to the ideal "Constant Voltage Source" used in Simscape.

2. Settling Time (t_{settle}) Comparison
   - **Observation:** The hardware took much longer to settle ($0.9406s$) compared to the simulation ($0.1095s$).
   - **Justification:**
     - **Stiction (Deadzone):** This is a classic symptom of dry friction (Coulomb friction). As the motor approaches the target, the error becomes small, and the Controller reduces the PWM voltage. On the hardware, once the voltage drops below a certain threshold (the "Deadzone"), the motor stalls before fully reaching the target. It may then slowly "inch" or creep into position, drastically extending the calculated settling time.
     - **Simscape Limitation:** The standard Simscape DC motor block uses a **Viscous Friction** model (Linear damping $b \cdot \omega$). It often underestimates the "sticky" nonlinear friction present in real gearboxes.

3. Overshoot ($O.S.\%$) Comparison
   - **Observation:** The hardware exhibited a small overshoot (3.91%), while the simulation showed zero overshoot (0.00%).
   - **Justification:**
     - **Discrete Time Delays:** The Simscape model runs in continuous time (or with a high-fidelity solver). The hardware runs on a discrete loop (Arduino) with communication overhead. This **time delay** in the feedback loop erodes the phase margin, causing the real system to overshoot slightly where the ideal model is perfectly damped.
     - **Inertia Mismatch:** The non-zero overshoot suggests the actual physical inertia of the rotor/gearbox system might be slightly higher than the parameter ($J = 0.00976\ kg \cdot m^2$) used in the simulation, causing the motor to carry more momentum past the setpoint.

Conclusion

The comparison reveals that the physical system is significantly slower ($0.23s$ vs $0.06s$ rise time) and exhibits slight overshoot (3.9%) compared to the ideal Simscape model. These discrepancies are attributed to the unmodeled voltage drop across the L293D driver (reducing effective torque) and communication latencies in the hardware-in-the-loop setup. The extended settling time in hardware highlights the presence of nonlinear Coulomb friction (stiction) in the gearbox, which prevents the clean, rapid settling seen in the linear simulation.

Appendix:

Appendix A: Arduino Code to Control Position

```
//// ---------- Serial monitoring options ----------
#define MONITOR_ENABLED  1
#define SERIAL_PLOTTER   1          // 1 = CSV for Serial Plotter, 0 = labeled text
const unsigned long MONITOR_PERIOD_MS = 20;
const unsigned long MONITOR_BAUD      = 115200;

// ------------------------ Pin mapping (UNO) ------------------------
const byte ENC_A_PIN   = 2;
const byte ENC_B_PIN   = 3;

const byte DIR_FWD_PIN = 8;
const byte DIR_REV_PIN = 7;

const byte PWM_PIN     = 9;
const byte POT_PIN     = A0;

// ------------------------ Model constants ------------------------
const float Kp = 4.0f;
const float Ki = 0.0f;
const float Kd = 0.0f;
const float N  = 100.0f;

const float DEAD_BAND_RAD      = 0.196349540849362077f;
const float ENC_RAD_PER_COUNT  = 0.000761044731974271 56f;
const float FILTER_DEN1        = 0.5f;
const float PWM_SCALE          = 63.75f;
const float PWM_MIN            = 0.0f;
const float PWM_MAX            = 255.0f;

// Pot scaling
const unsigned long POT_GAIN   = 51522UL;
const float POT_SCALE          = 1.1920928955078125e-7f;

// Fixed-step
const float DT                 = 0.001f;
const unsigned long DT_US      = 1000UL;

static inline float absf(float x) { return (x < 0.0f) ? -x : x; }

// ------------------------ Encoder ------------------------
```

```
volatile long g_encoderCount = 0;
volatile byte g_prevAB = 0;

// AB as 2-bit value [A B]
const int8_t QUAD_TABLE[16] = {
  0, -1, +1,  0,
 +1,  0,  0, -1,
 -1,  0,  0, +1,
  0, +1, -1,  0
};

void encoderISR() {
 const byte a = (byte)digitalRead(ENC_A_PIN);
 const byte b = (byte)digitalRead(ENC_B_PIN);
 const byte ab = (a << 1) | b;

 const byte idx = (g_prevAB << 2) | ab;
 g_encoderCount += QUAD_TABLE[idx];
 g_prevAB = ab;
}

// ------------------------ Controller state ------------------------
float g_posFiltState      = 0.0f;
float g_integratorState   = 0.0f;
float g_filterState       = 0.0f;

// ------------------------ Monitoring variables ------------------------
float g_mon_posRad        = 0.0f;
float g_mon_posFiltRad    = 0.0f;
unsigned int g_mon_adc    = 0;
float g_mon_setpointRad   = 0.0f;
float g_mon_error         = 0.0f;
float g_mon_uSat          = 0.0f;
byte  g_mon_duty          = 0;
bool  g_mon_active        = false;
bool  g_mon_fwd           = false;

static inline void derivatives(float absErr,
                              float integratorState,
                              float filterState,
                              float &dIntegrator,
                              float &dFilter) {
```

```
 dIntegrator = Ki * absErr;
 dFilter     = (Kd * absErr - filterState) * N;
}

// ODE3
static inline void ode3Update(float absErr) {
 const float y0 = g_integratorState;
 const float y1 = g_filterState;

 float k10, k11;
 derivatives(absErr, y0, y1, k10, k11);

 const float x0_2 = y0 + DT * 0.5f * k10;
 const float x1_2 = y1 + DT * 0.5f * k11;

 float k20, k21;
 derivatives(absErr, x0_2, x1_2, k20, k21);

 const float x0_3 = y0 + DT * 0.75f * k20;
 const float x1_3 = y1 + DT * 0.75f * k21;

 float k30, k31;
 derivatives(absErr, x0_3, x1_3, k30, k31);

 g_integratorState = y0 + DT * ((2.0f/9.0f)*k10 + (1.0f/3.0f)*k20 + (4.0f/9.0f)*k30);
 g_filterState     = y1 + DT * ((2.0f/9.0f)*k11 + (1.0f/3.0f)*k21 + (4.0f/9.0f)*k31);
}

static void controlStep() {
 long counts;
 noInterrupts();
 counts = g_encoderCount;
 interrupts();

 // Encoder counts -> radians
 const float posRad = ENC_RAD_PER_COUNT * (float)counts;

 // Discrete filter
 const float posFilt = posRad - FILTER_DEN1 * g_posFiltState;
 g_posFiltState = posFilt;

 // Pot -> radians (~0..2*pi)
```

```
  const unsigned int adc = (unsigned int)analogRead(POT_PIN);
  const float setpointRad = ((float)(POT_GAIN * (unsigned long)adc)) * POT_SCALE;

  // Error + magnitude
  const float error  = setpointRad - posFilt;
  const float absErr = absf(error);

  // Deadband + direction selection
  const bool active = (absErr >= DEAD_BAND_RAD);
  const bool fwd    = active && (error >= 0.0f);

  digitalWrite(DIR_FWD_PIN, fwd ? HIGH : LOW);
  digitalWrite(DIR_REV_PIN, (active && !fwd) ? HIGH : LOW);

  // PID on absErr
  const float filterCoeff = (Kd * absErr - g_filterState) * N;
  const float u = (Kp * absErr + g_integratorState + filterCoeff) * PWM_SCALE;

  // Saturate to [0, 255]
  float uSat = u;
  if (uSat > PWM_MAX) uSat = PWM_MAX;
  if (uSat < PWM_MIN) uSat = PWM_MIN;


  const byte duty = (byte)uSat;
  analogWrite(PWM_PIN, duty);

  // Update continuous states
  ode3Update(absErr);

  // --- Update monitoring variables ---
  g_mon_posRad      = posRad;
  g_mon_posFiltRad  = posFilt;
  g_mon_adc         = adc;
  g_mon_setpointRad = setpointRad;
  g_mon_error       = error;
  g_mon_uSat        = uSat;
  g_mon_duty        = duty;
  g_mon_active      = active;
  g_mon_fwd         = fwd;
}
```

```
#if MONITOR_ENABLED
static void monitorPrint() {
 static unsigned long lastMs = 0;
 const unsigned long nowMs = millis();
 if ((unsigned long)(nowMs - lastMs) < MONITOR_PERIOD_MS) return;
 lastMs = nowMs;

 if (SERIAL_PLOTTER) {
   Serial.print(g_mon_posFiltRad, 6); Serial.print(',');
   Serial.print(g_mon_setpointRad, 6); Serial.print(',');
   Serial.print(g_mon_uSat, 2); Serial.print(',');
   Serial.print((int)g_mon_duty); Serial.print(',');
   Serial.println(g_mon_error, 6);
 } else {
   Serial.print("posFilt(rad)=");  Serial.print(g_mon_posFiltRad, 6);
   Serial.print("  setpoint(rad)="); Serial.print(g_mon_setpointRad, 6);
   Serial.print("  potADC=");      Serial.print(g_mon_adc);
   Serial.print("  uSat(0-255)="); Serial.print(g_mon_uSat, 2);
   Serial.print("  duty=");        Serial.print((int)g_mon_duty);
   Serial.print("  err=");         Serial.print(g_mon_error, 6);
   Serial.print("  dir=");         Serial.print(g_mon_active ? (g_mon_fwd ? "FWD" :
"REV") : "STOP");
   Serial.println();
 }
}
#endif

void setup() {
#if MONITOR_ENABLED
 Serial.begin(MONITOR_BAUD);
 delay(50);
#endif

 pinMode(DIR_FWD_PIN, OUTPUT);
 pinMode(DIR_REV_PIN, OUTPUT);
 pinMode(PWM_PIN, OUTPUT);

 pinMode(ENC_A_PIN, INPUT_PULLUP);
 pinMode(ENC_B_PIN, INPUT_PULLUP);

 // Initialize previous AB and attach interrupts
 const byte a = (byte)digitalRead(ENC_A_PIN);
 const byte b = (byte)digitalRead(ENC_B_PIN);
 g_prevAB = (a << 1) | b;

 attachInterrupt(digitalPinToInterrupt(ENC_A_PIN), encoderISR, CHANGE);
 attachInterrupt(digitalPinToInterrupt(ENC_B_PIN), encoderISR, CHANGE);

 // Safe start
 digitalWrite(DIR_FWD_PIN, LOW);
 digitalWrite(DIR_REV_PIN, LOW);
 analogWrite(PWM_PIN, 0);

 noInterrupts();
 g_encoderCount = 0;
 interrupts();

 g_posFiltState = 0.0f;
 g_integratorState = 0.0f;
 g_filterState = 0.0f;
}

void loop() {
 static unsigned long nextTick = 0;

 if (nextTick == 0) {
   nextTick = micros() + DT_US;
   return;
 }

 unsigned long now = micros();
 while ((long)(now - nextTick) >= 0) {
   nextTick += DT_US;
   controlStep();
   now = micros();
 }

#if MONITOR_ENABLED
 monitorPrint();
#endif
}
```

Appendix B: MATLAB code for Performance Metrics Calculation

Task 6 (Simulink - Simscape Comparision)

```matlab
y_exp = squeeze(sim_hardware.Data);

time_vec = sim_hardware.Time;

plot(time_vec, y_exp);
```

```matlab
y_sim = squeeze(out.sim_model.Data);

time_vec_sim = out.sim_model.Time;

plot(time_vec_sim, y_sim);
```

```matlab
figure('Name', 'Model vs Experiment Comparison');

plot(time_vec, y_exp, 'b', 'LineWidth', 1.5); hold on;

plot(time_vec_sim, y_sim, 'r--', 'LineWidth', 1.5);

legend('Hardware Experiment', 'Simscape Model', 'Location',
'SouthEast');

title('Step Response Comparison: Simscape vs. Hardware');

xlabel('Time (s)');

ylabel('Output (Counts or Rad/s)');

grid on;
```

```matlab
% Calculate Metrics using 'stepinfo'

% Note: We crop the data to start AFTER the step for accurate
calculation

info_exp = stepinfo(y_exp(time_vec > 0), time_vec(time_vec > 0), 1.8);

info_sim = stepinfo(y_sim(time_vec_sim > 0), time_vec_sim(time_vec_sim >
0), 1.8);
```

```matlab
% Display Results

fprintf('--- Performance Metrics ---\n');
fprintf('Metric          | Hardware       | Simscape\n');

fprintf('-----------------|----------------|--------------\n');

fprintf('Rise Time (s)   | %.4f        | %.4f\n', info_exp.RiseTime,
info_sim.RiseTime);

fprintf('Settling Time(s)| %.4f        | %.4f\n', info_exp.SettlingTime,
info_sim.SettlingTime);

fprintf('Overshoot (%%)   | %.2f%%        | %.2f%%\n',
info_exp.Overshoot, info_sim.Overshoot);
```